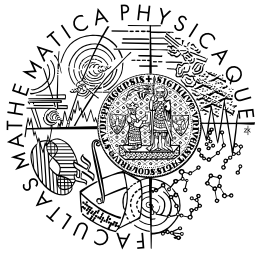


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Daniel Crha

Unified Querying of Multi-Model Data

Department of Software Engineering

Supervisor of the master thesis: doc. RNDr. Irena Holubová, Ph.D.
Study programme: Computer Science
Study branch: Software and Data Engineering

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Most of all, I want to thank my supervisor Irena Holubová for the countless hours she spent on the consultation and review of my thesis. Similarly, I want to thank Pavel Koupil for sharing so much of his knowledge and providing helpful insight. Finally, I want to thank my girlfriend and my mother for supporting me through the many bumps I encountered on the road through university, and for bringing joy to my life when I needed it the most.

Title: Unified Querying of Multi-Model Data

Author: Bc. Daniel Crha

Department: Department of Software Engineering

Supervisor: doc. RNDr. Irena Holubová, Ph.D., Department of Software Engineering

Abstract: The vast majority of current multi-model querying solutions require the user to have intimate knowledge of the specific models involved. There exists a single approach for truly unified multi-model querying, but this approach is not practically usable for most users due to its complexity. In this thesis we present MMQL, a multi-model query language based on category theory, which was designed using SPARQL as a basis. Using MMQL, users can formulate multi-model, multi-database queries without needing to know about the way the data is stored. We also present our proposal for the implementation of MMQL, including the required supporting algorithms. To verify the validity of our proposal, we built the proof-of-concept tool MM-quecat, an implementation of basic MMQL concepts. We then evaluated MM-quecat in a scenario involving PostgreSQL and MongoDB, querying both databases with a single MMQL query. As we present one of the first ever approaches for unified multi-model querying, we also analyze the weaknesses and limitations of the proposed approach, opening the door for future iterations and improvements.

Keywords: multi-model databases, querying, graph representation

Contents

Introduction

Foreword

In recent years, we have seen more and more projects steer away from traditional relational databases, instead opting for NoSQL databases thanks to their many advantages, especially in the world of big data where replication and sharding are requirements, not benefits. Similarly, we are seeing an increase in the usage of multi-model data, whereby different parts of data are stored in their most natural representation, utilizing the specifics of each particular model.

Where there is data, there is a need for querying, and multi-model scenarios are no exception. However, querying multi-model data is exceptionally challenging, as querying data from multiple models in the same query is non-trivial. Many databases today are multi-model to some extent, supporting a couple of selected models, but in general their query languages only support the secondary models as an extension. For this reason, the vast majority of existing multi-model querying approaches require the user to have knowledge of the particular data models when composing queries. This can be highly impractical, especially when the data is stored across multiple different databases, necessitating the knowledge of multiple query languages and the composition and unification of query results.

We believe that an approach allowing the composition of queries over multiple models and databases in a unified, model-agnostic way would be very beneficial, greatly reducing the complexity of multi-model querying. One such approach was proposed in 2021 in the form of MultiCategory [1], allowing unified multi-model querying based on functional folds, supported by a theoretical framework based on category theory [2]. However, while this approach is certainly robust, we question its wide applicability, as the provided query language carries considerable difficulty in forming queries for users who are not intimately familiar with functional programming. In addition, this approach does not take into consideration data redundancy, which is a key feature of multi-model, multi-database environments. For this reason, we believe that a more approachable solution is needed, ideally with a familiar and understandable query language.

In order to query multi-model data in a unified way, we need a unified representation. In the past, several approaches have attempted to use category theory to model data with support for querying, including Spivak et al. [3] with their Categorical Query Language (CQL) for relational data, and CGOOD [4] which focuses on the object-oriented data model with its own query language based on graph pattern matching. However, these approaches do not fully consider multi-model data in its generality. Luckily for us, an approach has been recently proposed for the unified representation of general multi-model data using category theory [5][6]. When presenting their findings, the authors of this unified representation expressed the possibility that their categorical model could be used as a basis for a unified multi-model querying solution. This is where this thesis begins its work, attempting to design a new multi-model querying approach with little in the way of approaches to rely on.

Goals

First of all, a query language is necessary for any kind of querying. Since the unified categorical data representation we will be relying on [5][6] represents data using category theory, our language will need to be a categorical query language. While we could design a query language from the ground up, it is considerably easier to start with another language as a template, making modifications as necessary. Since a category may be visualized in the form of a directed multigraph, the first goal of this thesis is to analyze existing graph query languages, and to adapt one of them to be suitable for categorical data. In this way, we aim to design a categorical multi-model query language.

While the design of such a query language allows the formation of multi-model queries in a unified fashion, a query language is nothing without a concrete plan of how to implement it. The proposal of an approach for the implementation of such a query language is arguably even more complicated than designing the language itself however, as many sparsely-studied problems specific to multi-model data arise along the way. Examples of these problems include, but are not limited to: the construction of multi-model query plans, finding the optimal order of multi-model joins, and evaluating the cost of multi-model query plans. As such, the second goal of this thesis is to propose an approach for the implementation of our query language. As we lack relevant prior approaches to base our work on, we do not aspire for our proposed design to be universal or well-optimized. Instead, we will focus on the clarity and simplicity of proposed algorithms and approaches, to form a solid basis for future work on this subject.

With a multi-model query language and an implementation proposal in hand, we need to verify that our proposal is well-formed. For this reason, the third goal of this thesis is to create a proof-of-concept implementation of our proposed multi-model querying approach, ultimately testing our proposal in a multi-model, multi-database scenario. Due to the sheer amount of work required for the design of the query language and supporting algorithms in such a complex problem domain with so little related work to fall back on, it is not within the scope of this implementation to be a fully-fledged, optimized, user-friendly piece of software. Instead, its purpose is to verify the validity of the concept itself, and to provide a platform for the examination of the strengths and weaknesses of the approach.

Finally, we recognize the fact that regardless of the amount of effort and time spent, no first attempt at anything is ever perfect. For this reason, the fourth and last goal of this thesis is to be as critical as possible when discussing the possible limitations and weaknesses of our proposals. This will ensure that this thesis can become a jump-off point for future related works, which can iterate and improve upon our proposals, moving them closer to real-world applicability.

Thesis Structure

First, we will briefly introduce multi-model data in Chapter 1, and we will showcase some of the most popular data models. Afterwards, we will give a brief overview of the unified categorical representation of multi-model data which we will be using as a basis for our work in Chapter 2. The final piece of related

work we will examine is the set of existing graph query languages, where we will attempt to analyze the existing approaches to decide upon a candidate language to modify for our problem domain. This examination is presented in Chapter 3.

Our original work starts with Chapter 4, where we present MMQL, a unified and declarative multi-model query language which we created, taking inspiration from the SPARQL query language. In addition we will present the algorithms and approaches necessary for the implementation of MMQL in Chapter 5, but we should make it clear that the goal of this chapter is not a concrete implementation, but rather a proposal of the algorithms necessary for such an implementation.

Moving on to the practical part of this thesis, in Chapter 6 we present MM-quecat, our proof-of-concept implementation of MMQL, allowing basic unified querying with PostgreSQL and MongoDB. In Chapter 7, we briefly present our ongoing related work on a user interface for MM-quecat, showing the user experience we are aiming for. Finally, in Chapter 8 we experimentally evaluate our solution, measuring the amount of overhead compared to native queries, as well as analyzing the main performance bottlenecks of our approach in MM-quecat.

1. Multi-Model Data

In the past, applications have tended to use mainly relational databases as their preferred way of storing data. This has been in large part due to the lack of maturity in the multi-model data ecosystem, which has slowly been changing in recent years. While sticking to a single data model (relational for example) has many benefits in terms of uniformity of modeling, access and management, in some use cases, it may be desirable to leverage the unique advantages offered by the usage of multiple different data models. Testament to this is the number of multi-model database technologies on the rise in both academia and industry settings. Among these are polystores [7][8] (database systems consisting of multiple heterogeneous integrated database systems) and multi-model database systems [9][10] (database systems supporting the storage of multiple data models at the same time). While both of the mentioned multi-model options allow multi-model querying, neither provides a unified querying experience which would allow the user to ignore the specifics of the given models. Altogether, we can broadly describe this trend towards using multi-model data as the NoSQL movement [11].

The biggest advantage of using multiple data models within an application is the possibility of modeling the data in the most appropriate way possible, meaning being able to for example model graph data using graph structures, rather than having to reshape the data to conform to a different model. Along with that, this approach can bring performance benefits, as databases suited to a particular model are naturally faster at storing and retrieving data in its native model [12].

This chapter introduces the most popular data models, and showcases why one may want to use them to model their data. It is important to understand the need for multi-model data before we delve deeper into the contents of this thesis, whose usefulness hinges on the fact that multi-model data is omnipresent in today's database ecosystem.

When talking about multi-model data, it can also be useful to have a unified vocabulary, since the terminology can differ between different data models. Such a unification was proposed by Pavel Koupil in his 2022 dissertation [13], and is shown in Table 1.1. There, we can see a set of unifying terms which cover the terminology of various popular models, for example the term *kind* is used to refer to tables in relational data, collections in document data, or column families in columnar data. This thesis uses this unifying terminology in multiple places, so it is recommended that the reader is familiar with these terms before moving on.

1.1 Aggregate-Oriented Models and Aggregate-Ignorant Models

Before delving into the specifics of the various data models, we should note a major divide among how the data models behave. They can be divided into two main camps.

The first one is aggregate-oriented models, which generally store and operate on aggregates – data units with complex and often nested structure. Aggregate-

Table 1.1: Unification of terms in popular models, proposed by Pavel Koupil [13]

Unifying term	**Relational**	**Array**	**Graph**	**RDF**	**Key/Value**	**Document**	**Column**
Kind	Table	Matrix	Label	Set of triples	Bucket	Collection	Column family
Record	Tuple	Cell	Node / edge	Triple	Pair (key, value)	Document	Row
Property	Attribute	Attribute	Property	Predicate	Value	JSON Field / XML element or attribute	Column
Array	–	–	Array	–	Array	JSON array / repeating XML elements	Array
Structure	–	–	–	–	Set / ZSet / Hash	Nested document	Super column
Domain	Data type	Data type	Data type	IRI / literal / blank node	–	Data type	Data type
Value	Value	Value	Value	Object	Value	Value	Value
Identifier	Key	Coordinates	Identifier	Subject	Key	JSON identifier / XML ID or key	Row key
Reference	Foreign key	–	–	–	–	JSON reference / XML keyref	–

oriented models generally focus on manipulating aggregates with various operations. Database systems working with aggregates can use the aggregate boundaries to determine which bits of data will be manipulated together, which can be useful for sharding or replication. As a result, cross-aggregate operations may be inefficient, or in some cases, totally impossible while maintaining Atomicity, Consistency, Isolation and Durability (ACID). Examples of aggregate-oriented models include document, column-family, and key-value models.

In contrast, aggregate-ignorant models do not have a universal strategy of drawing aggregate boundaries, as these boundaries depend on how we manipulate the data. This can be an advantage in the absence of a primary structure for accessing and manipulating the data, as it allows one to easily work with the data in different ways. Examples of aggregate-ignorant models include relational and graph models.

1.2 Relational Model

Relational data is arguably the oldest data model which is widely used today [14]. Whenever one is designing a system which requires some sort of data persistence, relational databases are usually among the first options to be examined. This is in large part due to their great track record of being solid, tested and well-understood by many.

Data in the relational model is organized into tables, with each table having a rigid schema defined by a set of columns, and data being stored in the form of table rows. The relational model generally contains some kind of key or identifier for each table row, be it a simple or compound one. An example of relational data can be seen in Figure 1.1, where it is shown in the form of an ER diagram [15] on the left, and in its native representation on the right.

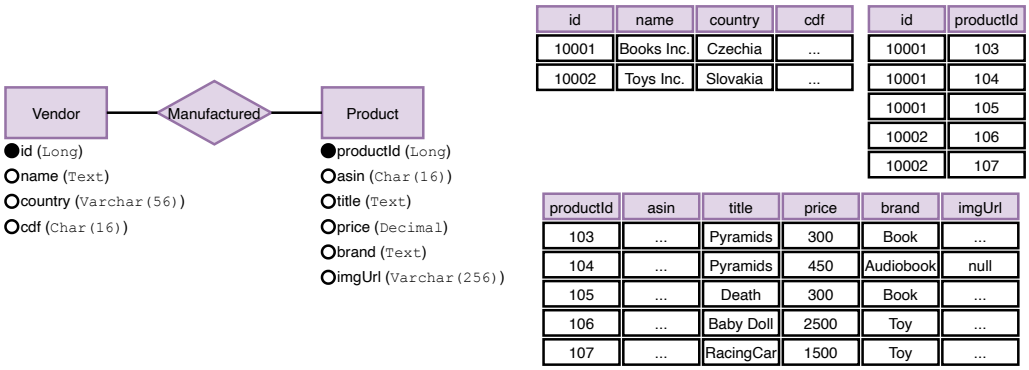


Figure 1.1: An example of relational data [16].

When querying relational data (typically using SQL), the largest amount of effort is generally spent on joining data together using database joins, in order to extract the data from the normalized form that it is stored in. This is a tradeoff which many are happy to make, considering relational databases generally offer great support for Online Transactional Processing (OLTP) workloads in the form of transactions – bundles of database commands which are executed with ”all or none” semantics. Relational systems also tend to have very good support of database integrity management tools, allowing database administrators to have full confidence in the continued integrity of their data.

However, the NoSQL movement has brought with it a desire for database *horizontal scalability*, as use cases arise where simply vertically scaling a relational database system by improving its hardware is not always enough [17]. In such use cases brought on by the advent of Big Data, many are seeking to leverage the benefits of traditional relational databases in the context of scalable and highly available data. This has spawned efforts which can be described as the NewSQL movement [18]. As examples, one can look at databases like CockroachDB[1] or VoltDB[2], which offer relational databases with the speed and scalability of NoSQL systems.

For the purposes of this thesis, it is necessary to point out a few representative database system for each data model, for the purposes of developing a universal querying approach. It would not be reasonably feasible to develop such an approach with respect to every database from each data model, therefore we simply select a couple of popular representative systems which feature the main characteristics of its corresponding data model. For relational databases, we put forward PostgreSQL[3] and MySQL[4] as some of the most popular options.

[1] `https://www.cockroachlabs.com/product/`
[2] `https://www.voltactivedata.com/`
[3] `https://www.postgresql.org/`
[4] `https://www.mysql.com/`

1.3 Document Model

One of the main critiques often leveraged at relational databases, especially by developers who highly value speed of development, is the fact that they can have a prohibitively high start-up cost in terms of data modeling and database management. If speed of development and ease of reasoning about the data model are of high importance, one may soon find themselves considering document oriented databases [19]. These databases are aggregate-oriented, with the aggregate boundaries being drawn as the boundaries of *documents* – self-contained pieces of data with a potentially complex structure. We can think of documents in a document-oriented database in terms of objects in a programming language – each has a structure and carries some data. Objects in the document-oriented model generally do not need to adhere to a schema, which makes document databases popular where ease of development is paramount. An example of document data may be seen in Figure 1.2.

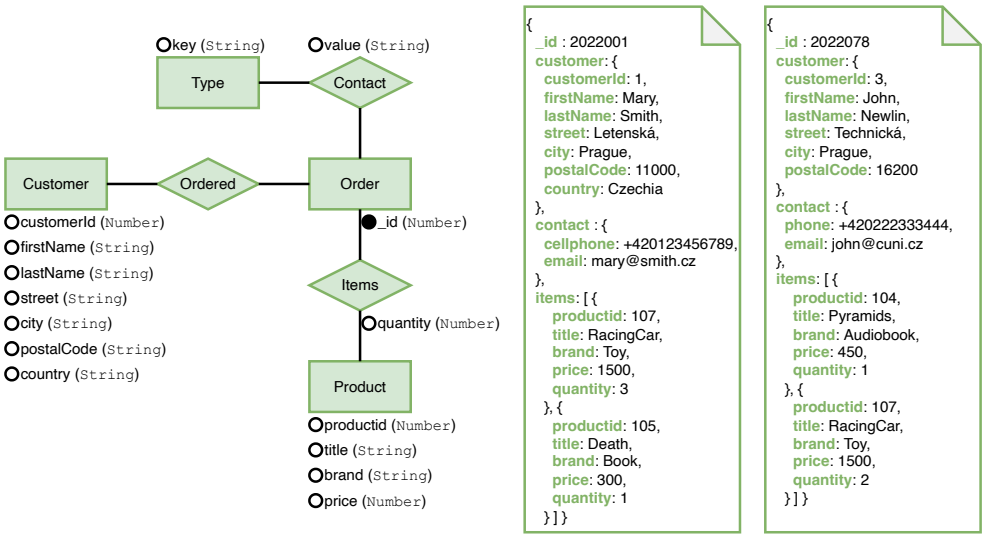


Figure 1.2: An example of document data in the JSON format [16].

Choosing a representative of the document model is not difficult, as MongoDB[5] is a very prominent and popular document database, which is very widely used in practice. MongoDB organizes documents into *collections*, which are sets of related documents which may be queried together. Working with MongoDB can have a relatively user-friendly learning curve, as one can store objects from most popular programming languages into the database in JSON[6] form without too many modifications, unlike the relational model where Object-Relational Mappers (ORM) are often required. However, in more complex use cases where working with multiple documents in a single transaction is required, developers may find the complexity resurfacing.

[5] `https://www.mongodb.com/`

[6] `https://www.json.org/json-en.html`

1.4 Graph Model

Graph databases model data in terms of objects (nodes) and relationships between those objects. There exist two main kinds of graph models – edge-labeled graphs and property graphs. Both of these graph models are discussed at greater length in Section 3.1. To see an example of data in the graph model, we refer the reader to Figure 1.3.

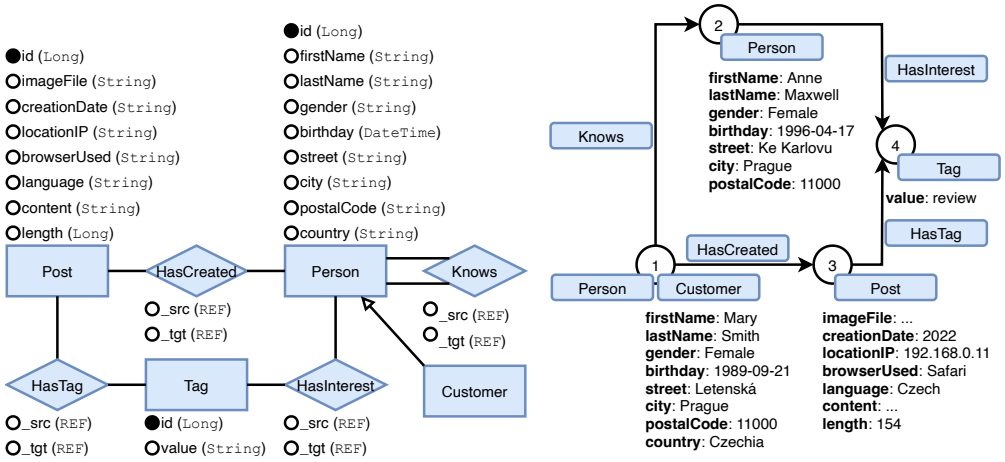


Figure 1.3: An example of graph data [16].

In general, graph databases are best suited for use cases where the graph nature of the data is important. These systems excel at querying for relationships between objects, and at finding patterns in the data. Unlike relational data, relationships in graph data models are first-class entities, and may often be given properties. In theory, graph databases provide a performance advantage compared to relational databases when considering graph-oriented workloads, however, this is not a clear-cut point of consensus in the academic community [20].

Regardless of performance implications however, it is undeniable that when presented with graph-like data, it is most convenient to model that data in a graph database. As such, a number of graph databases have reached wide adoption today. One of the foremost representatives of graph databases is Neo4j[7]. Although the rest of this thesis considers Neo4j as the main representative of graph databases, a special mention should be made to RDF data which is briefly discussed in Section 3.1.1, as RDF is also a graph data format, and is often used in the semantic web community.

1.5 Wide-Column Model

Wide-column data may seem very similar to relational data on the surface – both have concepts like rows and columns, storing data in table-like structures. However, the major difference from the relational model is the fact that the names and format of the columns may differ between rows of the same table (or column family, as it is sometimes called). Column-oriented databases store data

[7] `https://neo4j.com/`

in column-major order rather than in row-major order, meaning all values for a single column are stored contiguously, rather than contiguously storing all values for a single row. This confers performance benefits in many cases, such as for workloads where reading all data related to a single record is rare. An example of columnar data is shown in Figure 1.4.

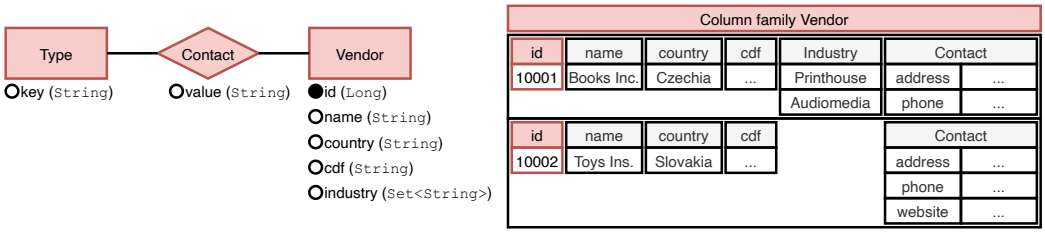


Figure 1.4: An example of columnar data [16].

When picking a representative for wide-column databases, Apache Cassandra[8] is a clear candidate, as it is open-source and widely used.

1.6 Key-Value Model

A key-value database or key-value store may be thought of as a hash table, which is designed to quickly store and retrieve opaque values. As such, key-value databases are very well suited to workloads like caching, often times being used as an in-memory cache of a slower, on-disk database. Naturally, the weakness of key-value databases is the fact that they are not generally designed to examine or query the actual values stored in any way beyond retrieving them based on the key. Therefore, they may not be ideal for more complex transactional or analytical workloads. See Figure 1.5 for an example of key-value data.

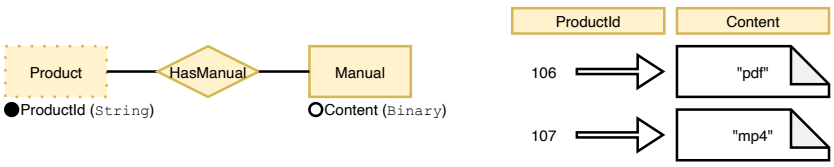


Figure 1.5: An example of key-value data [16].

We will consider Redis[9] and Riak KV[10] as representatives of a key-value stores for the purposes of this thesis.

[8] https://cassandra.apache.org/_/index.html
[9] https://redis.io/
[10] https://riak.com/products/riak-kv/

2. Categorical Data Representation

In the previous chapter, we examined the various data models which are widely in use today. As we saw, there is great variety between the models, and the same thing is true for the query languages designed for their respective models.

In this chapter, we will explore a concrete unified representation [6][5] of multi-model data using category theory, a powerful branch of mathematics which studies mathematical structures and relations between them. We will also discuss the benefits of such a representation, and we will then utilise it to build a universal, multi-model query language.

2.1 Benefits of a Unified Representation

As discussed in Chapter 1, there already exist two approaches to encompassing multiple data models in a single database system – polystores [7][8] and multi-model database systems [9][10]. However, neither approach provides a true, seamless, multi-model experience in terms of data modeling, querying and management, or this experience is only limited to two or three select models [10]. Naturally, it would be beneficial to be able to work with all popular models in such a unified way. Ideally, such a unified representation allows us to do the following [6]:

1. capture all the existing models, preferably in the same and definitely in a standard way;
2. query across multiple interconnected models efficiently;
3. perform correct and complete evolution management, i.e., propagation of changes;
4. enable data migration without complex reorganisations; and
5. permit integration of new data models.

Specifically, for the purposes of this thesis, having such a unified representation of multi-model data would form the perfect base for designing a multi-model query language. This is why we will spend a short while examining the unified representation of multi-model data proposed by Martin Svoboda, Pavel Koupil and Irena Holubová [6][5]. To demonstrate the concepts that we will explain in this chapter, we present a sample multi-model scenario in Figure 2.1, with a corresponding Entity Relationship (ER) schema shown in Figure 2.2.

2.2 Basics of Category Theory

As mentioned earlier, category theory [21] is a branch of mathematics which studies mathematical structures and relations between them. A category $\mathbf{C} = (\mathcal{O}, \mathcal{M}, \circ)$ consists of three entities:

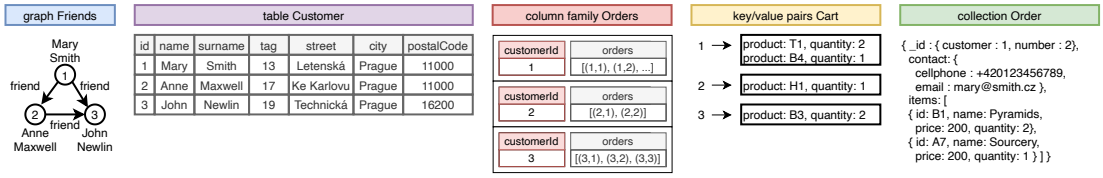


Figure 2.1: A sample multi-model data scenario [6].

- A class $\mathcal{O}$, whose elements are called objects. This class may also alternatively be referred to as $Obj(\mathbf{C})$.
- A class $\mathcal{M}$, whose elements are called morphisms, each of which has a target object and a source object. This class may also alternatively be referred to as $Hom(\mathbf{C})$.
- A binary operation $\circ$ called the composition over morphisms.

A useful visualization form of a category is the form of a *multigraph*, with category objects forming its set of vertices and schema morphisms forming directed edges.

We represent morphisms $f \in \mathcal{M} : A \to B$, as arrows, with $A, B \in \mathcal{O}$. We refer to object A as the *domain* of morphism f and to object B as its *codomain*[1]. Let $f : A \to B, g : B \to C \in \mathcal{M}$ be morphisms, then the composite morphism $g \circ f \in \mathcal{M}$ is also part of the set of morphisms in the same category, also called the *transitivity* property. The morphism composition operation is not only transitive, but it is also *associative*, i.e., $h \circ (g \circ f) = (h \circ g) \circ f$ for any morphisms $f, g, h \in \mathcal{M}$ such that $f : A \to B, g : B \to C$, and $h : C \to D$. Finally, for every object $A \in \mathcal{O}$, an *identity* morphism 1_A must exist, where $f \circ 1_A = f = 1_B \circ f$ for any $f : A \to B$. With respect to the composition operation, this identity morphism therefore acts as a unit element.

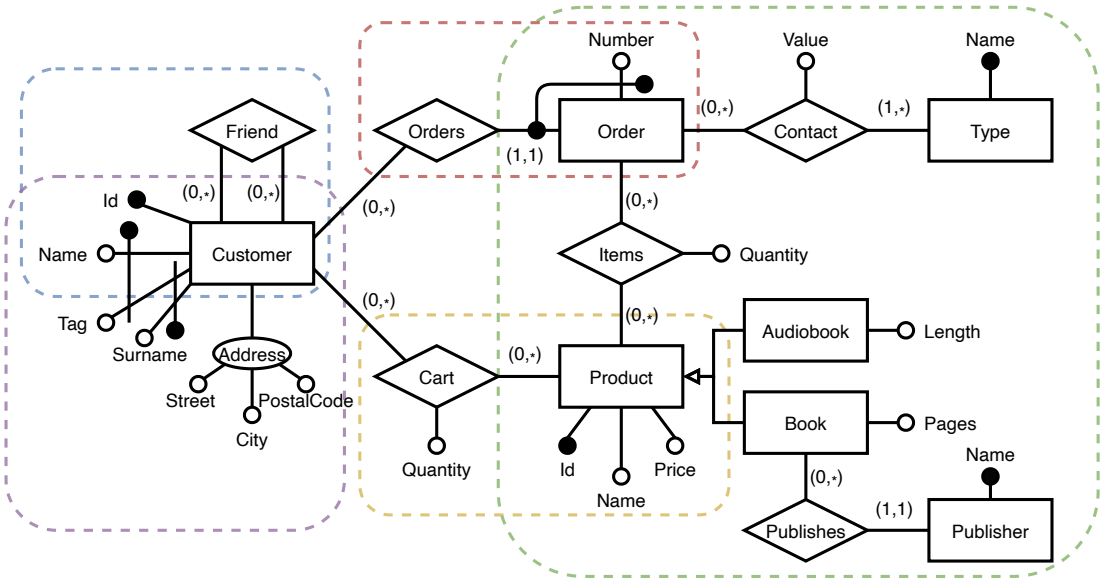


Figure 2.2: ER schema of the sample multi-model scenario in Figure 2.1 [6].

[1]The domain and codomain may also be referred to by *dom* and *cod*.

With these basic concepts from category theory, we can introduce the unified model [5][6] which we will be using in the rest of this thesis. We will introduce three main concepts which together form the basis of this model – the **schema category** describing the schema of the data in question, the **instance category** describing actual data conforming to its corresponding schema category, and the **mappings** which describe how objects from the schema category are stored in the underlying databases. Note that the introduction of the following concepts is somewhat informal and does not cover every single aspect of the categorical model, and for this reason we refer the reader to the original sources on the matter [5][6] for a more comprehensive definition.

2.3 Schema Category

We define a *schema category* [5][6] as a tuple $\mathbf{S} := (\mathcal{O}_\mathbf{S}, \mathcal{M}_\mathbf{S}, \circ_\mathbf{S})$, with $\mathcal{O}$ being the set of *schema objects*, $\mathcal{M}_\mathbf{S}$ being the set of schema morphisms, and $\circ$ being the morphism composition operation. Schema morphisms in $\mathcal{M}_\mathbf{S}$ connect pairs of objects from $\mathcal{O}_\mathbf{S}$. We distinguish two types of morphisms in $\mathcal{M}_\mathbf{S}$:

- *Base* morphisms are morphisms which are explicitly defined; and
- *Composite* morphisms which are obtained via the composition of base or composite morphisms via the composition operation $\circ$.

We define each schema object $o := (key, label, superid, ids) \in \mathcal{O}_\mathbf{S}$ to be a tuple. Let $\mathbb{O} \subseteq \mathbb{N}$ be a set of numbers forming the set of object identifiers. Then $key \in \mathbb{O}$ is an automatically assigned object identifier, unique for each schema object. We can optionally define names for schema objects using the *label* property, which is defined as $\perp$ when missing. Naturally, we also need our schema to contain the model of the data it represents. For this reason, each object contains a set of attributes $superid \neq \emptyset$, where each attribute corresponds to a morphism signature, introduced in the following paragraph. The set *superid* models the data each object is expected to contain. Finally, the set $ids \neq \emptyset \subseteq \mathcal{P}(superid)$ is a set of identifiers, each being a set of attributes from *superid*. This allows us to uniquely distinguish individual data instances using their attributes.

Each morphism $m := (signature, dom, cod, min, max) \in \mathcal{M}_\mathbf{S}$ is also defined as tuple, regardless of whether the morphism is base or composite. Firstly, let us define the alphabet $\mathbb{M}$, and the set of possible strings $\mathbb{M}^*$ over this alphabet (including ε which denotes an empty string). These strings are ordered lists of symbols from $\mathbb{M}$, concatenated using the $\cdot$ operator (for example 1 for a string containing a single character, 1.2.3 for a string with three characters, and ε for an empty string). For a given morphism m, $signature \in \mathbb{M}^*$ uniquely identifies this morphism, save for all identity morphisms which share the ε signature. If m is a base morphism, its signature will be a single alphabet symbol, i.e. $signature \in \mathbb{M}$. If m is a composite morphism however, its signature will be a string $signature \in \mathbb{M}^* \setminus (\mathbb{M} \cup \{\varepsilon\})$, as this allows us to decompose a composite morphism into its constituent base morphisms according to the $\circ$ operation. The domain and codomain of m are represented by *dom* and *cod* respectively, with both referring to schema objects in $\mathcal{O}$. In order to model the cardinality of each morphism, we also have the properties $min \in \{0, 1\}$ and $max \in \{1, *\}$,

which specify the minimal and maximal number of occurrences of each specific morphism in the instance category respectively.

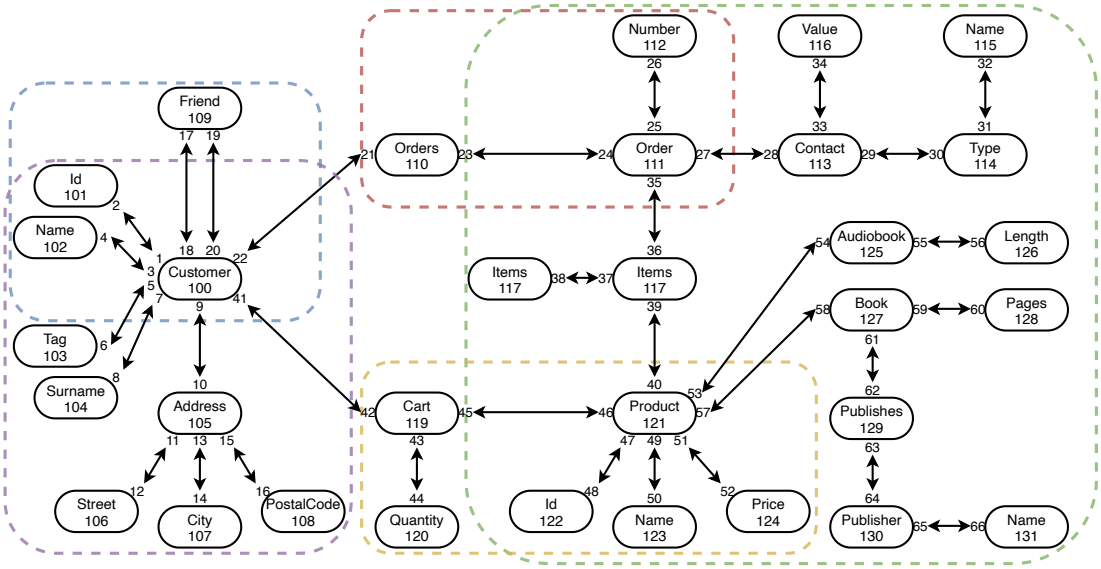


Figure 2.3: Schema category which was extracted from the sample ER schema in Figure 2.2 [6].

When it comes to morphism composition, we define the signature of the composite morphism $m_2 \circ m_1$ to be $signature := signature_2 \cdot signature_1$, unless either morphism is an identity morphism, in which case the resulting signature is the signature of the other morphism. The domain and codomain are composed naturally, resulting in the composite morphism having the domain of m_1 and codomain of m_2. Finally, we define the composite cardinalities to be $min = \min(min_1, min_2)$ and $max = \max(max_1, max_2)$. With the necessary definitions out of the way, we present an example of a schema category in Figure 2.3, where we can see the schema category corresponding to the sample multi-model scenario presented earlier in Figure 2.1.

2.4 Instance Category

As we mentioned earlier, the schema category $\mathbf{S}$ only describes the schema of the data, not data instances themselves. For this, we need to introduce the notion of an *instance category* [5][6]. An instance category $\mathbf{I} = (\mathcal{O}_\mathbf{I}, \mathcal{M}_\mathbf{I}, \circ_\mathbf{I})$ has the same structure as the corresponding schema category $\mathbf{S}$, meaning that for each schema object and morphism in $\mathbf{S}$, there exists a corresponding instance object or morphism in $\mathbf{I}$ and vice versa. A particular instance category $\mathbf{I}$ describes the data stored in a set of databases at a particular point in time, being essentially a snapshot of the entire data set. Whenever the underlying data changes, this will naturally induce a new instance category, with the relevant changes reflected.

Even though the sets of objects $\mathcal{O}_\mathbf{S}$ and $\mathcal{O}_\mathbf{I}$ have the same structure, just like $\mathcal{M}_\mathbf{S}$ and $\mathcal{M}_\mathbf{I}$, their representation must necessarily differ in order to be able to model the data instance. Let $\mathbb{V}$ be a set of all possible primitive values within the data instance we are describing. Then each instance object $o_\mathbf{I} := \{t_1, t_2, \ldots, t_n\} \in$

$\mathcal{O}_\mathbf{I}$ is a set of tuples $t_i : superid \to \mathbb{V}$ for all $i \in \mathbb{N}, 0 < i \leq n$, specifying the concrete values this instance object has in the current data instance. The specific set of tuples defined for a particular data instance for an instance object $o_\mathbf{I} \in \mathcal{O}_\mathbf{I}$ is called the *active domain* of this instance object, meaning the set of data currently bound to it.

Recall that each schema object also contains a set of identifiers *ids*, each of which uniquely identifies a particular object instance. In an instance category, this unique identification property must hold for each $id \in ids$ for each $o_\mathbf{I} \in \mathcal{O}_\mathbf{I}$. This means that if we project the active domain of this object to any particular identifier of this object, the number of unique tuples in the active domain must not change.

Now that we have explained the notion of instance objects when compared to schema objects, let us also introduce *instance morphisms*, which act as binary relations between instance object active domains. Let us consider an instance morphism $m_\mathbf{I} \in \mathcal{M}_\mathbf{I}$, $m_\mathbf{I} : o_1 \to o_2$ for some objects $o_1, o_2 \in \mathcal{O}_\mathbf{I}$. Then $m_\mathbf{I} \subseteq o_1 \times o_2$, meaning that the morphism is a subset of the relation induced by the Cartesian product of o_1 and o_2 (meaning the product of their active domains). Note that even though we define morphisms to be directed, we can traverse them in the opposite direction as they are defined as relations, which will be useful to us later.

2.5 Mapping Data to the Categorical Representation

In the previous sections, we defined the notion of a schema category representing the data schema, and an instance category representing the entire set of data at a particular point in time. However, in order to be able to automatically transition between the native data model and the categorical model, we also need to know how they map to each other. For this reason, we introduce the notion of *mappings* [6], which specify how data for one or more schema objects and morphisms are stored within a particular kind (recall Table 1.1) for a particular database. For example, this could mean describing the structure of rows in a table in the relational model, the structure of documents in a collection in the document model, or the structure of objects and their relationships in the graph model and so on. As it is likely that the entire schema will not fit within a single kind of a single database, it is expected that there will be multiple mappings for any given schema category, possibly even with some degree of overlap between the mappings if data redundancy is present. These mappings may be created manually by the user using a tool like MM-evocat [22] which we will be working with later in this thesis, but there are also approaches which attempt to infer the mapping from the databases themselves like MM-infer [23]. Note that sometimes, we may use the terms mapping and kind interchangeably in this thesis (especially during algorithmic descriptions), but in those cases we are always referring to the mapping which maps a given kind to its categorical representation.

Each of these mappings consists of a root object, denoting the root of the context of this kind within the schema category, the name of the kind which this mapping corresponds to, the source database of the kind, and finally an *access*

path, which describes the internal structure of the kind, and its mapping to the categorical representation. The access path for any given kind has a tree-like structure rooted in the mapping's root object, and recursively specifies the shape of the kind. We generally use a JSON-like representation of the access path, as shown in Figure 2.4.

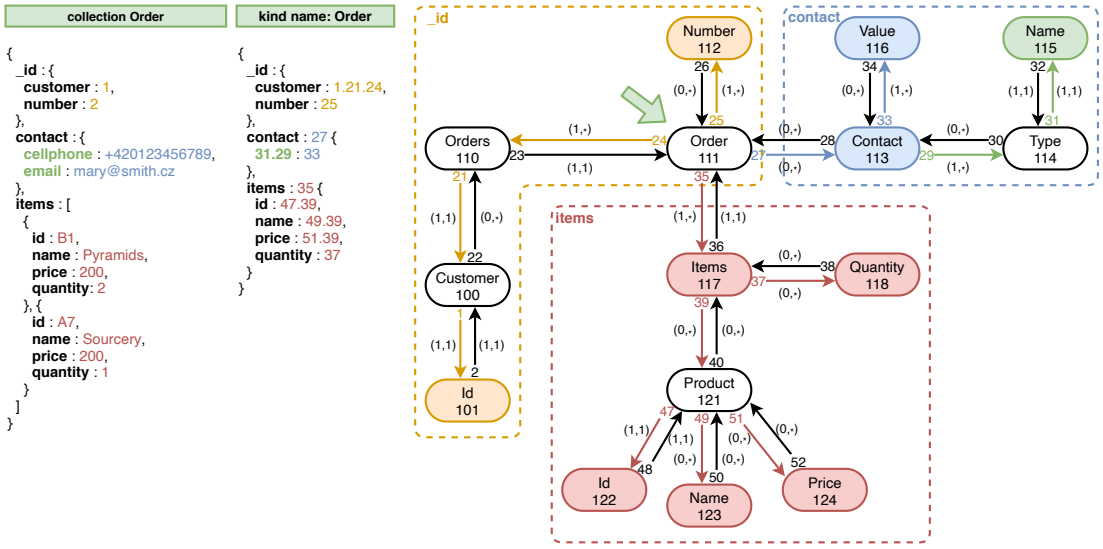


Figure 2.4: Collection *Order*, an access path for kind *Order*, and the corresponding part of schema category **S** [6]

When defining a property within the access path, we always define its name and structure, where the structure is constrained by the particular database model (for instance, mappings for kinds in a purely relational database are always flat with no nested properties, and their property names must be unique). This structure in the form of an access path must always cover at least one identifier of the root object, as without it, it would not be possible to distinguish different object instances. We will point out that not every schema object needs to be the root object of a mapping, as mappings generally map the values of many schema objects which are specified within a single mapping (for example the set of customer names, surnames and ids from a single relational table).

We distinguish three possibilities for the kinds of child properties in the access path for a given mapping:

- The child property is a direct neighbour of its parent in the schema category, meaning it is accessible via a base morphism.
- The child property is *inlined* from a more distant position within the schema category, meaning it is accessible from its parent via a composite morphism. Note that multiple paths can exist in the schema category between any given objects, which is why the exact composition of the composite morphism is important.
- The child property is *auxiliary*, meaning no corresponding object exists in the schema category. The purpose of such properties is purely structural.

The aforementioned associated morphisms are also called the property *context*, as they determine which schema object each property maps to (if any). Properties can have other kinds of names than simple static names, specifically we distinguish the following possibilities:

- *Static* names are user-defined as required by the structure of the underlying kind;
- *Dynamic* names are derived from instances of particular schema objects (for example types of contact like phone or email within a customer contact object); and
- *Anonymous* properties which do not have a name, or their naming is not permitted within the given model (for example array elements in JSON).

When it comes to the values of properties, we distinguish only two types of values:

- *Simple* properties which only contain primitive values; and
- *Complex* properties containing a set or list of child properties, like a nested array or nested object in JSON.

Note that the explanation of the concept of mappings was considerably less formal than the ones for schema and instance categories, as the formal definition is more complex in the case of mappings. For a more formal definition of the concept of a mapping, please refer to the paper by Pavel Koupil and Irena Holubová, where an exact definition is given [6]. While reading the original proposal of these ideas is not strictly necessary for readers of this thesis, it will certainly make the rest of it more digestible, as we will be expecting a certain level of familiarity with these concepts.

2.6 The Need for a Categorical Query Language

Now that we have examined the basics of category theory and the unified model we will be considering in the rest of this thesis, let us consider the *why* of building a query language using this unified model. As discussed earlier, there is no standard truly multi-model query language, which would be able to uniformly encompass all currently popular data models. The utility of such a query language should be fairly apparent – one does not need to bother with a different query language for each subset of their data which happens to be stored in a database with a different paradigm. On the contrary, queries across the entirety of one's data may easily be expressed in the same query language.

Even in the situation where one is not particularly perturbed by the need to know multiple query languages, there are still a few more problems on the horizon. Namely, there is an issue with queries which cross the boundaries of multiple database systems. For example, let us consider a scenario where we store a table of user information in PostgreSQL, and we store the order information in a collection in MongoDB. In such a case, expressing a query which crosses

the boundaries of both databases is impossible without sharing the same query language.

As the reader now hopefully shares our enthusiasm for the existence of such a language, we must still formalize the requirements for such a language. This language should:

- be able to encompass the particulars of all popular data models,
- be able to express queries crossing model boundaries,
- be expressive and readable,
- leverage the power of category theory,
- be intuitive and familiar to users of existing query languages,
- have the capability of being nearly as performant as native queries where possible.

Considering we may look at a category as a multigraph, we may not need to create a categorical query language from scratch. We can attempt to take inspiration in existing graph query languages, which is the focus of the following chapter.

3. Existing Graph Query Languages

Before engaging in the endeavor of designing a brand new graph query language for the categorical representation described in the previous chapter, it is prudent to first analyze other existing graph query languages, and consider their features, advantages and disadvantages. The most popular graph query languages are SPARQL [24], Cypher [25], Gremlin [26] and PGQL [27]. Another language worth mentioning is G-CORE [28], a graph query language designed by a team of academics and industry professionals, although it does not currently have an implementation. This chapter explores the properties of these languages and attempts to make a comparison of their capabilities.

3.1 Graph Data Models

Graph data can be modeled in a variety of ways, and each data model will naturally require a different kind of query language, despite containing only graph data. As such, a proper analysis of existing graph query languages necessitates a formalization of these data models. Naturally, a number of different graph data models have been tried in practice, but two have emerged as the most commonly used: *edge-labeled graphs* and *property graphs*, as described by Angles et al. [29] All of the aforementioned graph query languages operate on one of these two data models.

This section does not provide a rigorous definition of these data models, as their details may vary slightly between databases and query languages. Instead, it aims to illustrate the concepts of each data model and to showcase how they model data. As an aside, all graphs are implied to be directed multigraphs consisting of a set of vertices and a set of edges, unless explicitly specified otherwise.

3.1.1 Edge-Labeled Graphs

Edge-labeled graphs are graphs which assign a label to each edge from a set of labels. Each edge has exactly one label, while vertices are unlabeled.

This data model is quite simple yet strong. A vertex with a property can be modeled by introducing a new vertex containing the property value and connecting it with the original vertex via an edge labeled with the property name. Similarly, it is possible to model labeled vertices by introducing a new vertex containing the vertex label value, and connecting it with the original vertex via an edge labeled with an arbitrarily defined label, selected to denote vertex labels. The edge-labeled graph model is also capable of representing edge properties. This can be achieved by reification - materializing the edge into an extra vertex connected with new edges to both ends of the original edge. The extra vertex can then be linked with additional edges to other vertices representing the property values. An example of data modeled as an edge-labeled graph may be seen in Figure 3.1.

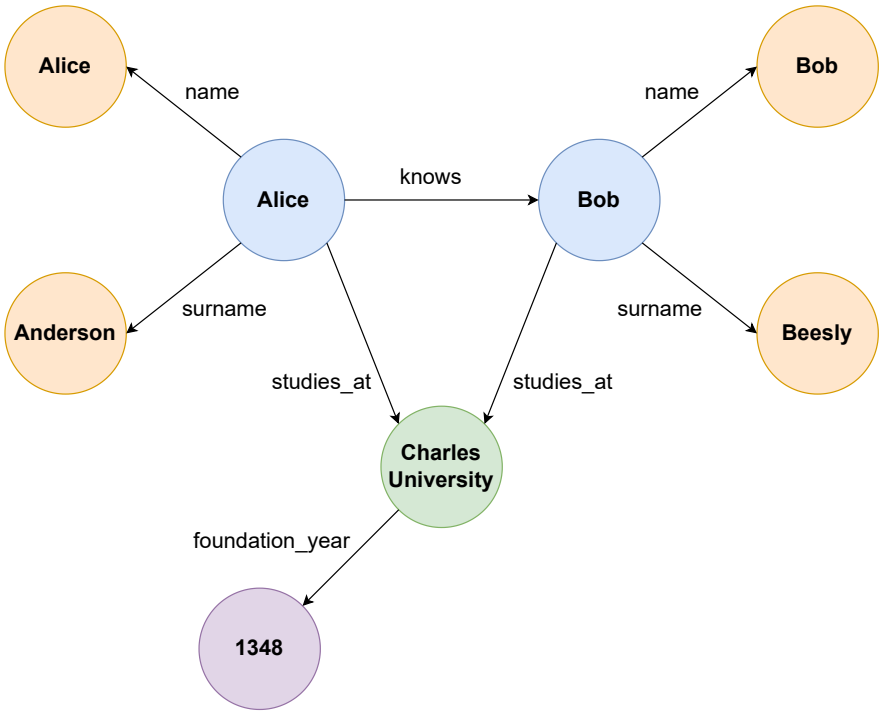


Figure 3.1: Data modeled as an edge-labeled graph.

Edge-labeled graphs are the data model for the Resource Description Framework (RDF) [30], for which SPARQL [24] acts as the most popular query language. RDF is a framework for representing information on the Web, and it models information as triples in the form *subject predicate object*. RDF graphs may contain three types of nodes: Internationalized Resource Identifiers (IRIs), literals (value types like strings and numbers) and blank nodes (vertices without a globally persistent IRI). Predicates are always IRIs, which can be equated to edges with labels from edge-labeled graphs.

3.1.2 Property Graphs

A graph data model used more widely in practice by graph-oriented databases is the property graph. A property graph can be seen as an extension of edge-labeled graphs, where both vertices and edges may be labeled. In addition, each vertex and edge may be associated with a set of key-value pairs called *attributes*, which store additional information about the vertex or edge. An example of data modeled as a property graph can be seen in Figure 3.2.

These additional constructs do not give property graphs a higher expressive power compared to edge-labeled graphs, but they do carry other benefits. First of all, the structure of the data in property graphs may be easier for users to work with, since data related to a particular vertex or edge is associated directly with that vertex or edge, rather than residing in a different vertex. Additionally, imagine a scenario where we have an edge-labeled graph, and we need to add a property to an edge which previously did not have any properties. The solution to this problem is to use reification and create a new in-between vertex holding the new property. However, such a change is disruptive to the schema of the data, and will necessarily break existing queries. Using the property graph model, such

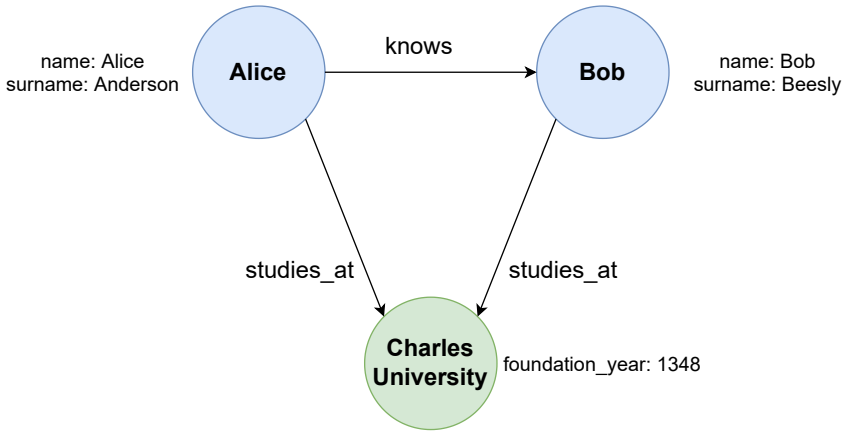


Figure 3.2: Data modeled as a property graph.

a change can be done without significantly disrupting the schema.

The property graph model forms the basis for the majority of modern graph data platforms. The foremost among them are Neo4j[1] and Apache TinkerPop[2], which primarily allow querying with Cypher [25] and Gremlin [26] respectively.

3.2 Language Features

With the graph data models clarified, we analyze the specifics of each selected graph query language. The presented languages are a sample of the most popular graph query languages which cover a variety of concepts and problem domains. Specifically, this section focuses on the querying capabilities of the given languages, and only briefly covers their mutation functionality.

3.2.1 SPARQL

As SPARQL [24] is primarily used as the query language for RDF [30], it is different from the other selected query languages in that it operates on an edge-labeled graph. It operates in terms of triple patterns – whitespace-separated lists consisting of a subject, predicate, and object. Any part of the triple pattern may consist of a constant (IRI or RDF literals like strings and numbers) or a variable. These triple patterns can then be used to match parts of the queried graph. SPARQL itself only supports querying, but SPARQL 1.1 Update [31], which is an update language for RDF graphs, allows for updates with a syntax very similar to SPARQL.

A simple SPARQL query which retrieves a list of books released in 2015 and their respective authors (ordered by birth year) may be seen in Figure 3.3. It can be seen that the SPARQL query structure is in some ways inspired by SQL – we have a `SELECT` clause specifying what the query should return, as well as a `WHERE` clause specifying the conditions for the returned variables, and an `ORDER BY` clause. However, these similarities end upon inspection of the graph querying capabilities of SPARQL. In order for a set of variable bindings to be

[1] `https://neo4j.com/`
[2] `https://tinkerpop.apache.org/`

```
PREFIX lib: <http://example.org/library#>

SELECT ?author ?book
WHERE
{
    ?author lib:authored    ?book .
    ?book   lib:releaseYear 2015  .
    ?author lib:birthYear   ?authorBirthYear .
}
ORDER BY ASC(?authorBirthYear)
```

Figure 3.3: Basic SPARQL query example.

returned by the query, their substitution into the `WHERE` clause must induce a valid subgraph of the queried graph. This `WHERE` clause utilizes the aforementioned triple patterns to this end. In the sample query, the first triple pattern specifies that the query should match triples in the form of `?author lib:authored ?book .`, where the dot at the end signifies the end of the triple. The second triple specifies that the matched book should be released in 2015. Specifying both triples after one another in the `WHERE` clause functions as their conjunction - both must match in order for the entire `WHERE` clause to match. Lastly, the `PREFIX` clause simply specifies the prefix `lib:` to be equivalent to the namespace IRI `http://example.org/library#`. This does not increase the expressive power of SPARQL, but it makes queries significantly more readable compared to queries which use the full namespace IRI at every occurrence.

SPARQL also supports more complex graph patterns such as joins, unions, differences, optional matching, and filtering. Joins are implicit both between triple patterns and graph patterns – they act as a natural join on the set of shared variables between the two patterns whenever they are specified within the same graph pattern. The `MINUS` and `NOT EXISTS` expressions allow the elimination of matches depending on the evaluation of a pattern and the existence of a pattern respectively. Using the `OPTIONAL` keyword, the query may specify that parts of the graph pattern are optional, meaning that when they cannot be successfully matched for a given solution, that solution is not discarded, but rather any unmatched variables remain unbound. Filtering matches are possible using the `FILTER` keyword, which specifies a condition and filters all matches which do not satisfy this condition.

More complex graph pattern matching can be seen in Figure 3.4. This query returns the list of authors and co-authors of books which have been released in 2015 or later, and additionally returns the book rating if it is available. If the rating is not present in the data, that book is still returned, with the `?rating` variable remaining unbound. The `UNION` construct in the query ensures that results will be returned for both authors and co-authors of any given book.

When it comes to navigational queries (i.e. queries which navigate the graph), SPARQL uses the concept of *property paths*. These are constructs not unlike regular expressions, which specify possible routes through a graph between two graph nodes. Property paths may contain RDF terms or variables at both ends,

```
PREFIX lib: <http://example.org/library#>

SELECT ?author ?book ?rating
WHERE
{
    { ?author lib:authored ?book . }
    UNION
    { ?author lib:coauthored ?book .}

    ?book   lib:releaseYear ?releaseYear .
    OPTIONAL { ?book lib:rating ?rating . }

    FILTER(?releaseYear >= 2015)
}
```

Figure 3.4: More complex graph pattern matching in a SPARQL query.

but they cannot contain variables as part of the path itself. An example of a property path is `lib:TheHobbit lib:sequel+ ?book .`, which will bind the `?book` variable to any book which follows The Hobbit as a direct or indirect sequel. A path like `lib:TheHobbit lib:sequel/lib:rating ?rating .` would bind `?rating` to the rating of The Hobbit's direct sequel. Additional property path features include inverse paths from object to subject, alternative paths or negated property sets which match any except the specified IRIs in the path.

Figure 3.5 shows a query which returns the list of all authors who co-authored a book with J. R. R. Tolkien. This query uses a property path to find all books co-authored by Tolkien, and then uses an inverse path from these books to their co-authors. Note that it is necessary to filter Tolkien himself from the results of the query, as the `?author` variable would otherwise match Tolkien himself, as he is naturally a co-author for all books he co-authored.

```
PREFIX lib: <http://example.org/library#>

SELECT DISTINCT ?author
WHERE
{
    lib:JRRTolkien lib:coauthored/^lib:coauthored ?author .
    FILTER(?author != lib:JRRTolkien) .
}
```

Figure 3.5: Graph navigation in a SPARQL query.

SPARQL is most notably supported by the Virtuoso[3], Amazon Neptune[4],

[3] `https://virtuoso.openlinksw.com/`
[4] `https://aws.amazon.com/neptune/`

Stardog[5], Apache Jena Fuseki[6] or GraphDB[7] graph databases.

3.2.2 Cypher

Cypher [25] is a declarative graph query language originally developed by Neo4j. It was open-sourced in 2015 and contributed to the openCypher [25] project, which is now responsible for maintaining the language. Cypher operates on a property graph data model, referring to the vertices of the graph as *nodes* and the edges as *relationships*. Its structure is inspired by SQL, as queries are composed of various clauses. Its pattern matching is also inspired by SPARQL [24]. Cypher may be used for both querying and updating graphs.

Clauses in a Cypher query are chained together, and the results of each clause become the inputs for the next clause. Regarding basic graph pattern matching, Cypher notation tries to be very intuitive when it comes to the underlying graph data model. It models relationships as an arrow between two nodes: `(a)-[r]->(b)`. Such a pattern describes a directed relationship `r` from the node `a` to the node `b`. Note that both nodes and relationships do not have to be named, they may simply use the `()` and `[]` notation respectively. The pattern `(a)-[]-(b)` will match the relationship in either direction. These patterns are used in the `MATCH` clause, which matches the patterns to the underlying data and binds any variables accordingly.

It should be noted that unlike SPARQL, Cypher does not return matches where the same graph relationship is found multiple times in a single pattern. What this means is that if we specify a pattern like `(a)-[]->(b)<-[]-(c)`, Cypher implicitly makes sure that the node `a` is distinct from the node `c`. If allowing the return of the same node in different parts of the pattern is desirable, multiple consecutive `MATCH` clauses must be used, matching parts of the pattern separately and using some common variables to join them together.

```
MATCH (author:Author) -[:AUTHORED]-> (book:Book {releaseYear: 2015})
RETURN author, book
ORDER BY author.birthYear ASC
```

Figure 3.6: Basic Cypher query example.

The last information needed for composing a simple query is knowing how to specify labels and properties. The construct `(a:Author surname: "Tolkien") -[r:AUTHORED]-> (b)` will match `a` to nodes labeled `Author` and with the property `surname` equal to "Tolkien", and `r` to relationships typed `AUTHORED`. The query shown in Figure 3.6 uses these principles to return a list of books released in 2015 and their respective authors (ordered by birth year). The `RETURN` clause is used for projection on the output variables, and it is mandatory in queries which do not mutate the underlying dataset. The `ORDER BY` clause works similarly to SQL.

[5] `https://www.stardog.com/`
[6] `https://jena.apache.org/documentation/fuseki2/`
[7] `https://graphdb.ontotext.com/`

Much like SPARQL pattern matching, joins happen implicitly when we specify multiple `MATCH` clauses in succession. The `NOT` operator acts as a difference – its usage in a `WHERE` clause after a `MATCH` clause will eliminate matching results from the results of the `MATCH` clause. Optional pattern matching is also supported with the `OPTIONAL MATCH` clause. Filtering is accomplished with the `WHERE` clause.

```
MATCH (author:Author) -[:AUTHORED]-> (book:Book)
OPTIONAL MATCH (book) -[:RATING]-> (rating:Rating)
WHERE book.releaseYear >= 2015
RETURN author, book, rating.average AS rating

UNION

MATCH (author:Author) -[:COAUTHORED]-> (book:Book)
OPTIONAL MATCH (book) -[:RATING]-> (rating:Rating)
WHERE book.releaseYear >= 2015
RETURN author, book, rating.average AS rating
```

Figure 3.7: More complex graph pattern matching in a Cypher query.

Figure 3.7 shows a more complex example of graph pattern matching using some of the features mentioned above. The depicted query returns the list of authors and co-authors of books released in the year 2015 or later, and additionally returns the book rating if it is available (for simplicity's sake, we assume that the rating is stored in a node connected to a book via the `RATING` relationship). It should be noted that Cypher does not have an easy way of specifying alternate matches like SPARQL does, and the `UNION` clause works at the level of query results. Therefore we need to run both queries independently, and then `UNION` their results. Cypher also does not yet support post-union processing, so if we wanted to order the results of the union, we would need to resort to other workarounds.

As far as navigational queries go, the strength of Cypher is considerably lower than that of SPARQL. Traversal of a path is limited to a single relationship type (i.e. label), and if multiple labels are desired, the traversal has to be broken up into multiple parts. The same applies to properties of the relationship – path pattern matching is only able to match the same set of properties and property values for each relationship in the path. Variable-length pattern matching is signified by the usage of the `*` operator. The pattern `(a)-[*2]->(b)` is equivalent to the pattern `(a)-[]->()-[]->(b)`, i.e. a path of length 2. The length of this path can be variable: `(a)-[*2..4]->(b)` means that the path is at least of length 2 but no more than length 4. Either of these bounds can be omitted, meaning *length X or more* or *length X or less*. If both bounds are omitted, the path can be of any length: `(a)-[*]->(b)`, as the lower and upper bounds default to 1 and infinity respectively. The ability to specify a range for the path length present in Cypher is not present in SPARQL.

What Cypher lacks in terms of path traversal specification, it makes up for in other ways of working with paths. It features paths as a first-class citizen of the language, offering the ability to save paths to variables or to return them from queries. It also has a number of interesting functions like searching for one or all

shortest paths between two nodes in the graph. Cypher also has native support for list primitives, something which is missing from SPARQL (and RDF).

```
MATCH path = (:Book {title: "The Hobbit"}) -[*:SEQUEL]-> (book:Book)
RETURN book.title AS title, length(path) + 1 AS seriesNumber
```

Figure 3.8: Graph navigation in a Cypher query.

Figure 3.8 demonstrates a query which returns the list of all sequels of the book The Hobbit transitively. The query also returns for each book its number in the series, meaning The Hobbit's direct sequel will have the number 2, its sequel the number 3 and so on.

The most notable databases using Cypher are Neo4j[8], Amazon Neptune[9], Memgraph[10], Katana Graph[11] and RedisGraph[12].

3.2.3 Gremlin

Gremlin [26] is a graph traversal language operating on property graphs. It was developed by Apache TinkerPop of the Apache Software Foundation, and is available under the Apache License 2.0. Gremlin is a functional language which operates in terms of a data flow. A traversal in Gremlin consists of a sequence of steps, each of which performs an operation on the underlying stream of data. This data is functionally passed between three kinds of steps: map steps which transform the data, filter steps which filter, and remove some of the data, and side effect steps which can compute statistics about the data stream.

```
g.V().
    hasLabel('Book').
    has('releaseYear', 2015).
    as('book').

    in('authored').

    hasLabel('Author').
    as('author').

    select('book', 'author').
        by('title').
        by('name')
```

Figure 3.9: Basic Gremlin query example with imperative traversal.

[8] https://neo4j.com/
[9] https://aws.amazon.com/neptune/
[10] https://memgraph.com/
[11] https://katanagraph.com/
[12] https://redis.io/docs/stack/graph/

Figure 3.9 shows a simple Gremlin query which retrieves the list of books released in 2015 and their respective authors, returning their titles and names. The `g.V()` step returns a list of all vertices in the graph. Using `hasLabel('Book')`, only vertices with the Book label are selected. Similarly, the `has()` step filters out books which were not released in 2015. The `as()` step is not a real step, but rather a step modulator which assigns a label to the previous step, making it accessible by later steps. It can be thought of as referencing to that step with a variable. Edge traversal is performed with the `in()` step, which in the example traverses incoming edges with the `authored` label. Other edge traversal steps exist which traverse edges in a specified direction, or both. The `select()` step takes data from previously labeled steps and collects them, additionally projecting the data to the book title and author name.

Gremlin differs from the previously mentioned graph query languages by allowing imperative graph traversal as shown in Figure 3.9, but it also supports declarative traversals, as well as allowing mixing of imperative and declarative traversals. An example of a declarative traversal can be seen in Figure 3.10. While declarative pattern matching is possible, it is considerably less succinct than its SPARQL or Cypher equivalents, as Gremlin focuses primarily on graph traversal. Gremlin also supports filtering and unions, but other more complex operation like set differences and optional matching require extra effort.

```
g.V().
    match(
        as('tolkien').
            hasLabel('Author').
            has('name', 'J. R. R. Tolkien'),
        as('tolkien').
            out('coauthored').
            as('book'),
        as('book').
            in('coauthored').
            as('coauthor'),
        where('coauthor', neq('tolkien'))
    ).
    select('coauthor').by('name')
```

Figure 3.10: Declarative graph pattern matching in a Gremlin query.

Gremlin supports variable-length paths via the `repeat()` step, which offers both do-while and while-do semantics. The `repeat()` step may repeat any traversal, meaning the repeated traversal may itself consist of multiple steps. Figure 3.11 showcases this in a query which performs a full traversal of the graph by iteratively following edges. This traversal starts with the author J. R. R. Tolkien, and follows outgoing edges until it gets to end nodes which have no outgoing edges. It uses the `simplePath()` step to eliminate any potential cycles in the graph during traversal. It also contains another stop condition – the maximum traversal depth of 10 set with `loops().is(10)`. Gremlin is also able to return entire paths from a query.

```
g.V().
    hasLabel('Author').
    has('name', 'J. R. R. Tolkien').

    repeat(out().simplePath()).
        until(outE().count().is(0).
            or().loops().is(10))
```

Figure 3.11: Graph navigation in a Gremlin query.

Gremlin can be used for both Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP), meaning a Gremlin traversal can be executed as either a real-time database query, or as a batch analytics query. Notable graph databases implementing Gremlin are Amazon Neptune[13], OrientDB[14], Blazegraph[15] and Titan[16].

[13] https://aws.amazon.com/neptune/
[14] https://orientdb.org/
[15] https://blazegraph.com/
[16] https://titan.thinkaurelius.com/

4. MMQL – A Categorical Query Language

So far, we have examined the particulars of multi-model data, introduced a categorical model for representing said data, and we have formulated the need for a query language to query this model. Based on requirements specified in Section 2.6, this chapter introduces the **Multi-Model Query Language (MMQL)**, a query language created specifically to query the aforementioned categorical representation. Before we introduce MMQL, the reader may get a taste of the language by examining the example query shown in Figure 4.1.

```
SELECT {
    ?customer ordered ?productName ;
        name ?customerName .
}
WHERE {
    ?product 49 ?productName ;
        -39/36 ?order .

    ?order -23/21 ?customer .
    ?customer 3 ?customerName .

    FILTER(?productName = "Lord of the Rings")
}
```

Figure 4.1: An example query in MMQL, retrieving customers who ordered the Lord of the Rings book. The corresponding schema category is shown in Figure 2.3.

MMQL is a query language for categorical data which operates on a schema category. Its purpose is to query data in a unified way using the categorical representation, again utilizing categorical data representation to return the data in a unified representation. As such, if data stored across a variety of different databases is modeled using a schema category, one can query all of that data using a single, unified query language. Under the covers, this singular query is translated into potentially many subqueries in the relevant databases' own query languages, and the results of those queries are joined to create the final query result. We should note that MMQL is a querying-only language, it does not aspire to handle any data modifications.

Designing a query language is no trivial task, therefore this chapter pays special attention to the design process behind MMQL, and various key decisions made during its design. This chapter explores the major concepts in MMQL, but its full grammar is available in Attachment A.1. A comparison of constructs supported in MMQL is shown in Table 4.1, where we show a feature comparison with selected popular query languages, which are used in some of the most popular databases in their respective models (as we discussed in Chapter 1).

Table 4.1: Comparison of constructs supported in MMQL and in single-model query languages [32].

MMQL	**PostgreSQL (SQL)**	**Neo4j (Cypher)**	**MongoDB**	**Cassandra (CQL)**
FROM	FROM	-	db.collection	FROM
SELECT	SELECT	RETURN	$project	SELECT
WHERE	WHERE	WHERE	$match	WHERE
FILTER	condition(s)	condition(s)	condition(s)	condition(s)
COUNT, MIN, MAX, AVG	GROUP BY ... HAVING	COUNT, MIN, MAX, AVG	aggregate(...)	GROUP BY
graph pattern	JOIN	MATCH	$lookup	-
OPTIONAL	OUTER JOIN	OPTIONAL MATCH	-	-
UNION	UNION	UNION	$unionWith	-
EXCEPT	EXCEPT	WHERE NOT	$setDifference	-
ORDER BY	ORDER BY	ORDER BY	sort	ORDER BY
OFFSET	OFFSET	SKIP	skip	-
LIMIT	LIMIT	LIMIT	limit	LIMIT
DISTINCT	DISTINCT	DISTINCT	distinct	DISTINCT
AS	AS	AS	”alias” : ”$field”	AS
{ SELECT ... }	(SELECT ...)	CALL MATCH	-	-

4.1 Taking Inspiration from SPARQL

In Chapter 3, we have discussed existing graph query languages which may be suitable for the purpose of querying categorical data. Naturally, it is easier to modify and adapt an existing query language, than to design one from the ground up. Not only is designing a complete grammar a complicated process, but one must pay close attention to making sure that the language has the desired expressive power, while maintaining ease of use, readability, and ease of implementation. As such, MMQL is based on SPARQL [24], a query language for RDF [30] data discussed in Section 3.2.1.

As a reminder, SPARQL queries operate on the concept of RDF triples - tuples in the form of *subject-predicate-object*. As RDF expresses data in the form of IRIs, all three members of said triples may be IRIs. This is actually the catalyst of why SPARQL was chosen as the base language for MMQL - having a language which works with global identifiers transitions easily into working with globally-identified schema objects and morphisms. For example, in SPARQL,

we may use the triple `?customer <http://xmlns.com/foaf/0.1/name> "Alice Anderson"` to describe a customer variable having the name Alice Anderson. An important observation is that a schema category with its objects and morphisms is reminiscent of an RDF graph. The basic idea of MMQL is then to replace IRIs in SPARQL subjects and objects with references to schema category objects, and to replace IRIs in SPARQL properties with references to schema category morphisms. If we have a schema morphism with the signature of 42 which refers to the relationship between a customer and their name, we can express an equivalent triple in MMQL like so: `?customer 42 "Alice Anderson"`.

While it is not immediately clear that such a modification to SPARQL will yield a reasonable categorical query language, the suggestion itself is not so far-fetched. However, it is not yet obvious why SPARQL was chosen as the base language for MMQL. The first reason for that has already been touched on slightly, which is the fact that modifying an existing language is simply easier from a development point of view. However, the more important reason is the reason of usability - SPARQL is very good at expressing complex graph patterns, including constructs like paths in a graph. When working with a more complex schema category, it is imperative that compound morphisms are easy to express in the query language, allowing greater readability and better expressiveness of the language. SPARQL with its property paths [33] allows the chaining of properties into paths in the property graph, allowing the user to express a "transitive friendship", i.e. friends of friends recursively, using the simple form `?customer <http://xmlns.com/foaf/0.1/knows>+ ?friend`, meaning that the `http://xmlns.com/foaf/0.1/knows` occurs in the path 1 or more times. Similarly, it is possible to chain together different properties, allowing us to retrieve the name of a friend: `?customer <http://xmlns.com/foaf/0.1/knows> / <http://xmlns.com/foaf/0.1/name> ?friendName`. It is this capability that proves very important when working with categorical data, because it maps perfectly to compound morphisms. As such, if the respective morphisms would have the signatures 60 and 42, we could use the MMQL notation `?customer 60/42 ?friendName`. This method of graph traversal is very intuitive and allows MMQL to be very expressive when working with the schema category.

4.2 Design Process

Before we start introducing the various concepts in MMQL, we will shortly describe the process used to arrive at the current iteration of MMQL, as well as the supporting algorithms presented in Chapter 5. Firstly, we evaluated the graph query languages discussed in Chapter 3, and selected SPARQL as the best candidate for adaption to a categorical domain, for reasons specified in the previous section. Afterwards, we used an iterative process where we incrementally took concepts from SPARQL one-by-one and attempted to transition them to work with our categorical data model. This was a very lengthy and cumbersome process, however it was aided by a single key fact, which is the power of the categorical model. Since the categorical model we are using has the capability of representing multi-model data in a unified way, we only need to worry about supporting all concepts of the categorical model itself, at which point we will transitively support the relevant features of the various data models. Even so,

the design process was arduous, which is why it was aided by a set of approximately 20 sample scenarios, which the author of this thesis constructed to aid in the iterative process. With each new iteration of MMQL or its supporting algorithms, they were manually evaluated in these scenarios on paper, and notes were made of possible issues arising from the current iteration. In this way, we arrived at the form in which MMQL and its supporting algorithms are presented in this thesis. These scenarios changed dramatically over time as the proposed approach evolved, and because they are not particularly relevant to presenting the final iteration of MMQL and its supporting algorithms, they are omitted from this thesis.

4.3 Basic Concepts of MMQL

To see the bare bones of MMQL at work, let us look at the query shown in Figure 4.2. Just like SPARQL, each query consists of a few main clauses, the mandatory ones being `SELECT` and `WHERE`. We will start by introducing the `WHERE` clause, since it is responsible for the selection of the data, with the `SELECT` clause being responsible for projection to the desired form.

```
SELECT {
    ?customer name ?name .
}
WHERE {
    ?customer 42 ?name .
}
```

Figure 4.2: A basic MMQL query selecting customer names.

4.3.1 WHERE Clause

The `WHERE` clause defines a graph pattern, using triples of the form *subject-predicate-object*. Each such triple must also end with a period, marking its end. This graph pattern defines the data which should be matched by the query, as per the schema category. A **variable** is defined using the syntax `?varName`, with each variable matching a specific schema object. Variables are strongly typed, meaning it is disallowed to use the same variable in positions implying different schema objects - if the variable `?customer` is used in the position of a schema object with a key of 10, it must always be used in that position. Unlike SPARQL, variables in MMQL may only be placed in the subject and object positions, they may never be in the predicate position. The reason for this decision is the fact that placing variables in the subject position (and therefore querying schema morphisms) has the semantics of querying the schema category itself and not the underlying data. These semantics are not desired for MMQL, as the schema category is simply a model of the data, and MMQL focuses on querying the data only. Looking at Figure 4.3, we can see examples of statements one may use in the `WHERE` clause.

```
WHERE {
    // Simply traversing a morphism
    ?customer 42 ?name .
    // Morphism traversal in the opposite direction
    ?name -42 ?customer .
    // Chaining morphisms
    ?customer 55/31 ?orderNumber .
    // Repeating subjects - syntactic sugar for graph patterns
    ?customer 42 ?name ;
        55/31 ?orderNumber .
    // Filtering data
    FILTER(?orderNumber >= 15)
    // Using aggregations
    FILTER(?orderNumber = MAX(?orderNumber))
    // Optional pattern - variables will be null if not found
    OPTIONAL {
        ?customer 23 ?address .
    }
}
```

Figure 4.3: A showcase of WHERE clause contents.

The job of the `WHERE` clause is to specify the data to query and bind variables to the matched data. Semantically, the `WHERE` clause induces a schema category, where each variable has its own schema object, such that there exists a homomorphism between this induced schema category and the original schema category. Such a homomorphism maps all schema objects corresponding to variables to the schema objects queried by those variables. As a result, we may think about the data matched by the `WHERE` clause as forming an instance category conforming to the aforementioned induced schema category, with other clauses of the query operating on this instance category. However, a more practical approach to reasoning about this is to imagine that the `WHERE` clause is matching graph patterns, and its result is a set of matching graph patterns, where each pattern contains a set of variable bindings to their real values. The other clauses would then operate on these graph patterns found within the data, performing tasks like projection or ordering.

4.3.2 SELECT Clause

At first glance, the `SELECT` clause looks a lot like the `WHERE` clause, which is because both clauses specify some kind of graph pattern. However, their semantics are very different. It is the job of the `SELECT` clause to project the matched data to a shape which is desired for the output of the query. Just like in SQL one might want to only query certain properties, or to return them in a given order, in MMQL, one may choose the schema to which the query results will adhere.

The `SELECT` clause uses new morphism definitions, not the ones from the schema category. This allows the user to not just select parts of existing data in the same shape it is already in, but change the shape of the data altogether.

```
SELECT {
    // Define new morphisms using alphanumeric identifiers
    ?customer name ?customerName .
    // Form more complex graph structures
    ?customer ordered ?product .
    ?product name ?productName .
    // Syntactic sugar also applies here
    ?customer name ?customerName ;
        ordered ?product .
    // Syntax for a blank node - node without data
    // This is useful for creating new shapes in the graph
    ?customer details _:customerDetails .
    _:customerDetails name ?customerName .
    // Aliasing and aggregations
    ?customer orderedNum COUNT(DISTINCT ?productName) AS ?numProds .
}
```

Figure 4.4: A showcase of SELECT clause contents.

Morphism definitions having user-defined names is useful, because for example if the user then takes the query results and converts them into JSON[1], the JSON document will not only have a meaningful structure, but its properties will be aptly named. Please refer to Figure 4.4 for examples of statements possible within the `SELECT` clause.

Again, the `SELECT` clause has some specific semantics in terms of category theory. The `SELECT` clause induces a new schema category, and the results of the query form an instance category adhering to this schema category (which is different from the schema category used to model the data).

Observant readers who are already familiar with SPARQL may note that the MMQL `SELECT` clause looks similar to the SPARQL `CONSTRUCT` clause, which is not a coincidence. Both clauses specify a graph pattern rather than a list of variables to return. However, because MMQL is a categorical query language, it also returns data in the categorical representation. It is naturally expected that a user may want to get the data in a more practical representation (such as JSON or RDF), however, this is not part of the query language, but rather is something that is handled by the tooling around MMQL. The topic of transforming data outputted by the query is expanded on further in Section 5.2.8.

It is also worth mentioning the optional `FROM` clause, which may be present between the `SELECT` and `WHERE` clause together with a schema category identifier. In the case that a particular MMQL query engine contains multiple defined schema categories, this gives the user the option to specify the one they wish to query. If there are multiple schema categories but the `FROM` clause is omitted, it is up to the query engine to decide which schema category is the default one to use in such cases.

[1] `https://www.json.org/json-en.html`

4.4 Advanced Concepts of MMQL

So far, we have presented a sufficient amount of information for a reader to pick up MMQL at a very basic level. However, eventually, users will need more advanced constructs present in other query languages, like those pertaining to ordering or aggregating data. This chapter expands on those concepts, in no particular order.

4.4.1 ORDER BY Clause

Data ordering is a necessity for any query language, and MMQL naturally does include it. Figure 4.5 shows the usage of ordering in MMQL - let us consider some items for which users are able to submit reviews with their ratings. As shown, we can use MMQL to calculate the average user rating for each item, and order those items by their average rating in descending order.

```
SELECT {
    ?item averageRating AVG(?rating) AS ?avg .
}
WHERE {
    ?review 40 ?item ;
        42 ?rating .
}
ORDER BY ?avg DESC
LIMIT 10
OFFSET 20
```

Figure 4.5: A showcase of the ORDER BY, LIMIT and OFFSET clauses.

Semantically, if we consider the `SELECT` clause to return graph instances, then the `ORDER BY` clause introduces a higher-order graph, which dictates the relative ordering of those graph instances. In other words, we divide up the instance category corresponding to the `SELECT` clause into maximally connected components, and assign ordering to those components based on the ordering parameters.

4.4.2 LIMIT and OFFSET Clauses

To facilitate the incremental querying of a large amount of data, MMQL includes the `LIMIT` and `OFFSET` clauses. These clauses respectively limit the number of graphs matching the `SELECT` clause to return, and skip a specific number of them with respect to their ordering.

It only makes sense to use the `LIMIT` and `OFFSET` clauses when grouped together with `ORDER BY`, as without it, the order of the results is undefined. As such, iteratively using `OFFSET` and `LIMIT` without `ORDER BY` may yield duplicate or missing records, depending on the implementation of MMQL and the state of underlying data in the data stores.

4.4.3 Advanced Graph Manipulation

While simple graph patterns introduced so far may be sufficient to get the job done in most scenarios, the advantage of MMQL is its great expressiveness when it comes to graph patterns. This subsection introduces a few helpful concepts in MMQL which will help the user write more expressive and readable queries.

Morphism traversal and chaining was introduced in Figure 4.3, but let us examine its semantics in more detail. When specifying a graph pattern of the form `?customer 55/31 ?orderNumber`, that is semantically equivalent to specifying the triples `?customer 55 ?order` and `?order 31 ?orderNumber`. However, it may still be useful to use morphism chaining whenever possible, as depending on the MMQL implementation, the query engine may optimize the query execution by discarding data which is not actually needed in the query, even if it is part of the queried subgraph. Morphism chaining may include the following modifiers: `|` to introduce alternative paths, `?`, `*` and `+` with the same semantics as in regular expressions (optional, repeated any number of times and repeated at least once, respectively), together with brackets `()` to control precedence. However, do note that in order for a repeated morphism to be valid, it must form a cycle in the schema category, otherwise such repetitions would not be possible.

```
SELECT {
    _:sameOrder customer ?customerA ;
        customer ?customerB ;
        product ?productName .
}
WHERE {
    ?customerA 55/36/12 ?productName .
    ?customerB 55/36/12 ?productName .

    FILTER(?customerA != ?customerB)
}
```

Figure 4.6: A query selecting two different customers who ordered an item with the same name.

The inquisitive reader may have started to wonder what happens if the same variable is used multiple times in the query. By the principle of least surprise, this notation (showcased in Figure 4.6) means that all occurrences of the same variable refer to the same value. This is useful in cases similar to the one shown in Figure 4.6, where we want to use a graph pattern to specify that multiple sources point to the same thing. This would have been also possible to express without this feature, simply using different variables together with `FILTER`, however this feature greatly aids the expressiveness of MMQL.

4.4.4 Data Types

MMQL considers a few primitive data types, aside from categorical data: strings, integers, and booleans. This set of data types may however be freely extended. SPARQL supports working with floating-point numbers, dates or language-tagged

strings, and in principle, there is nothing preventing MMQL from being extended to do so as well. In addition, the set of filtering operations in MMQL may also be freely extended together with the set of data types, allowing the future inclusion of features like regular expression matching.

4.4.5 Filtering and Aggregation

`FILTER` clauses may be used to introduce selection into the query, with the valid operators being = (equality) and ! = (inequality), as well as four other comparison operators: <, <=, >, >=.

In addition to `FILTER` clauses, MMQL also offers `VALUES` clauses, which constrain a variable to a set of possible values, specified in the form of `VALUES ?var { "value1", "value2" }`.

Aggregations naturally may be used as one of the operands of a comparison expression, as well as elsewhere in the object position in a triple. The available aggregation functions are `COUNT`, `SUM`, `AVG`, `MIN` and `MAX`.

4.4.6 Set Operations

As shown in Table 4.1, MMQL also supports set operations in the capacity that one would expect a fully-fledged query language to support them.

```
SELECT {
    ?customer boughtOrReviewed ?productName .
}
WHERE {
    {
        ?customer 60/34 ?productName .
    }
    UNION
    {
        ?customer 76/34 ?productName .
    }
}
```

Figure 4.7: Using UNION in MMQL.

The `UNION` statement combines the result sets from two graph patterns in the `WHERE` clause. Such a case is shown in Figure 4.7, where we can see a query selecting a list of product names for each customer, that the customer bought or reviewed. While both patterns combined by `UNION` do not necessarily need to have an identical structure, all variables used in `SELECT` must have their value definitely bound in all `UNION` variants. This rule applies in other situations as well - the only time when variables are allowed to be unassigned is when they are part of an `OPTIONAL` statement, which accepts a graph pattern and marks it as optional, leaving its variables unbound if its pattern is not present in the data.

The `MINUS` statement again takes two graph patterns, and results in solutions which are compatible with the first graph pattern, but not the second one. This is

```
SELECT {
    ?customer bought ?productName .
}
WHERE {
    ?customer 60 ?product .
    ?product 34 ?productName .
    MINUS {
        ?product 34 "Lord of the Rings" .
    }
}
```

Figure 4.8: Using MINUS in MMQL.

showcased in Figure 4.8, where we can see a query returning customers and their purchased products, but not returning any customers who happened to order the Lord of the Rings book.

The reader may notice that while query languages like SQL often also possess an `INTERSECT` clause representing set intersection, MMQL does not. This is by design, as modeling set intersection in terms of graph patterns simply means merging the patterns into a single one, returning matches for both patterns.

4.4.7 Subqueries

Naturally, the expressive power of MMQL is not sufficient for some queries as defined so far. For example, let us consider the situation where we want to find out the highest rating given to any item by any user, and we want to select all items having received such a rating. This is not achievable with a single query in MMQL, since we need to perform a calculation across all instances of a particular kind, and then continue working with that value.

Therefore it is necessary to use query nesting - MMQL supports writing queries inside the `WHERE` clause of higher-level queries. In such a case, variables from the `SELECT` clause of the nested query are bound into the context of the `WHERE` clause of the containing query. We can see this happening in Figure 4.9, where an inner query computes the maximal rating given to any product, with the instance category for the inner query `SELECT` clause just containing a single active domain row for the `maxRating` object, having the value of the maximal product rating. As a result, we can bind the variable into the context of the containing query, and filter only products which have received the highest available rating in any review.

It should be noted that joining the inner `SELECT` variables into the outer `WHERE` clause only works when there is a common variable to use as a join point. The reason why this join point is not required in Figure 4.9 is that it can be inferred from the shape of the inner `SELECT` clause that it will only contain a single active domain row, and can therefore be used as a constant in the outer query.

There is a tradeoff to take note of here in connection to the way nested queries are joined to parent queries. As presented, it is not valid to form subqueries in which the schema permits multiple active domain rows without a join point.

```
SELECT {
    ?product hasMaxRating ?rating ;
        name ?productName .
}
WHERE {
    ?review 40 ?product ;
        42 ?rating .
    ?product 34 ?productName .

    {
        SELECT {
            _:maxRating rating MAX(?oneRating) as ?maxRating .
        }
        WHERE {
            ?review 40 ?product ;
                42 ?oneRating .
        }
    }

    FILTER(?rating = ?maxRating)
}
```

Figure 4.9: Nested query in MMQL.

However, there may be queries which will always return a single active domain row despite their schema allowing them to return multiple (for example if we filter the set of customers by their ID). In these instances, it may be desirable to defer the validation of the subquery join point until the subquery has already been executed, and to allow the joining of queries which only contain a single active domain row regardless of their result schema. However, this would mean that the validity of such a query cannot be verified without executing it. As such, we posit that both approaches are valid, and we chose the former approach due to its user-friendliness.

4.4.8 Grouping

The perceptive reader familiar with SPARQL may have noticed that we did not mention the inclusion of the equivalent of SPARQL's `GROUP BY` in MMQL. This is not an accident, but rather a conscious decision, which was made in light of the fact that MMQL already implicitly supports grouping in another way. In the context of grouping without aggregation, there is not much to consider, as this can be achieved using graph patterns.

However, in the context of aggregations across grouped sets of solutions, things get more interesting. Aggregations in MMQL are evaluated based on the dependence of variables on other variables, which is best demonstrated using the query shown in Figure 4.10 as an example. Let us examine the `SELECT` clause in this query, where we can see that the value of the `?rating` variable is dependent

```
SELECT {
    ?product name ?productName ;
        avgRating AVG(?rating) AS ?avgRating .
}
WHERE {
    ?review 40 ?product ;
        42 ?rating .
    ?product 34 ?productName .
}
```

Figure 4.10: MMQL query containing implicit grouping.

on the value of the `product` variable[2]. For this reason, the aggregation of the `?rating` variable is also dependent on the value of the `?product` variable, effectively meaning that it will be evaluated separately for each product. The equivalent PostgreSQL query which could be generated from the MMQL query in Figure 4.10 is shown in Figure 4.11.

```
SELECT product.name, AVG(review.rating)
FROM review INNER JOIN product ON review.productId = product.id
GROUP BY review.productId
```

Figure 4.11: SQL query equivalent to the MMQL query shown in Figure 4.10.

For completeness, we will also mention that aggregations across an entire kind are also possible in MMQL. An example of this was already shown in the subquery of the query shown in Figure 4.9. This is achieved by simply making the aggregation not depend on any variables other than the aggregated variable itself.

[2]This is achieved using morphism contractions, which are explained later in Section 5.2.7.

5. Algorithms for Implementing MMQL

Having proposed a multi-model query language called MMQL in Chapter 4, in theory we could now start writing queries encompassing multiple data models. However, designing such a language is only half the work. Naturally, for a query language to be useful, it must also be executable. Designing the supporting algorithms which are necessary for the implementation of a query language is far from trivial, which is why we will spend a considerable amount of effort describing them in this chapter. A very rough basis of an approach for unified multi-model querying based on category theory was first outlined by Pavel Koupil and Irena Holubová [6] as part of the potential future applications of their proposed unified categorical representation of multi-model data. This chapter builds on their initial ideas, developing them into a set of fully-fledged algorithms which support the implementation of our proposed query language – MMQL.

As the implementation effort itself is also non-trivial, we will separate it into a separate chapter altogether, and in this chapter, we will only focus on the design of the algorithms themselves. The algorithms presented in this chapter form the basis of *MM-quecat*, a proof-of-concept implementation of MMQL described in Chapter 6.

5.1 Proposed approach

Since we want to reuse existing database systems as they are, our querying approach must necessarily rely on the translation of MMQL queries into queries in the databases' native query languages, subsequently transforming the retrieved data into our categorical representation. However, to enable us to reason better about the whole approach, let us first describe it from a very high level view. As such, we can divide our proposed approach into the following steps:

1. *Create a set of query plans.* Analyze the query and prepare a plan of which data from which databases should be used during the query execution. In the case of data redundancy, multiple alternative query plans are created.

2. *Create a join plan for each query plan.* Given a query plan, joining data from different databases may be required. The point of a join plan is to define the join points for the query plan, as well as the data which will be necessary from both ends of the join point to perform the actual join.

3. *Select the best query plan.* If there are multiple possible query plans due to data redundancy, the best query plan must be selected by the query planner, if a specific plan was not explicitly requested by the user. Note that this step may need to come after the query part translation step instead if the query planner needs to use the generated native queries to make an estimation of the cost of the entire query.

4. *Process individual query parts.* If we consider a query part to be a self-contained unit of execution, we need to translate it into a database-native

query. We define query parts in such a way that it is always possible to translate a single query part into a single native query.

5. *Merge query part results.* When we have the native database query for each query part, we need to execute it, transform the result into the categorical representation, and merge the retrieved data together.

6. *Perform projection on the result.* After merging all query part results into a single one (corresponding to the contents of the entire `WHERE` clause of the query), we need to project the results to the desired final representation as described by the `SELECT` clause, before returning them to the user. Also, there may be some MMQL statements which cross database boundaries, which makes them impossible to execute fully within the context of a single query part. Therefore we need to execute these deferred statements in this step.

7. *Transform the result into the desired format.* Since an instance category may not be the desired output format for many use cases, this optional step lets users specify which format the query results should be returned in, like JSON or RDF.

An example of a query being processed may be seen in Figure 5.1. The steps as shown in the diagram do not correspond one-to-one to the steps outlined in this chapter, but the overall flow is the same. Figure 5.1 comes from a conference demo paper which the author of this thesis coauthored with Pavel Koupil and Irena Holubová, but which is not yet published at the time of writing of this thesis [32].

As we will see later on in this chapter, a number of tradeoffs were involved in the formation of this version of the proposed approach. Since MMQL and its supporting algorithms are attempting to break new ground in a sparsely studied area, our main goals for this approach were *simplicity* and *comprehensibility*, with performance not being a large focus point. Considering we are proposing one of the first ever unified querying solutions for the multi-model, multi-database environment, we are aware that any solution we propose at this stage will have limitations and weaknesses, which is why we focus on designing a simple solution first, analyzing its limitations, and pointing out what may be improved by future works on this topic.

5.1.1 Database Wrappers

As we divide the MMQL query into separate query parts, the idea is for each query part to be a self-contained unit of execution which compiles down to a single native database query, regardless of the specific database. We also wish to design our algorithms to be *database agnostic*, as the spectrum of multi-model databases is vast, and we do not want to confine ourselves to a specific set of them. Both of these facts motivate the concept of *database wrappers*, which encapsulate the specifics of each database and its native query language behind a generic interface. As the general idea of the wrappers is encapsulation of database-specific logic, we have two main requirements for the design of this generic interface:

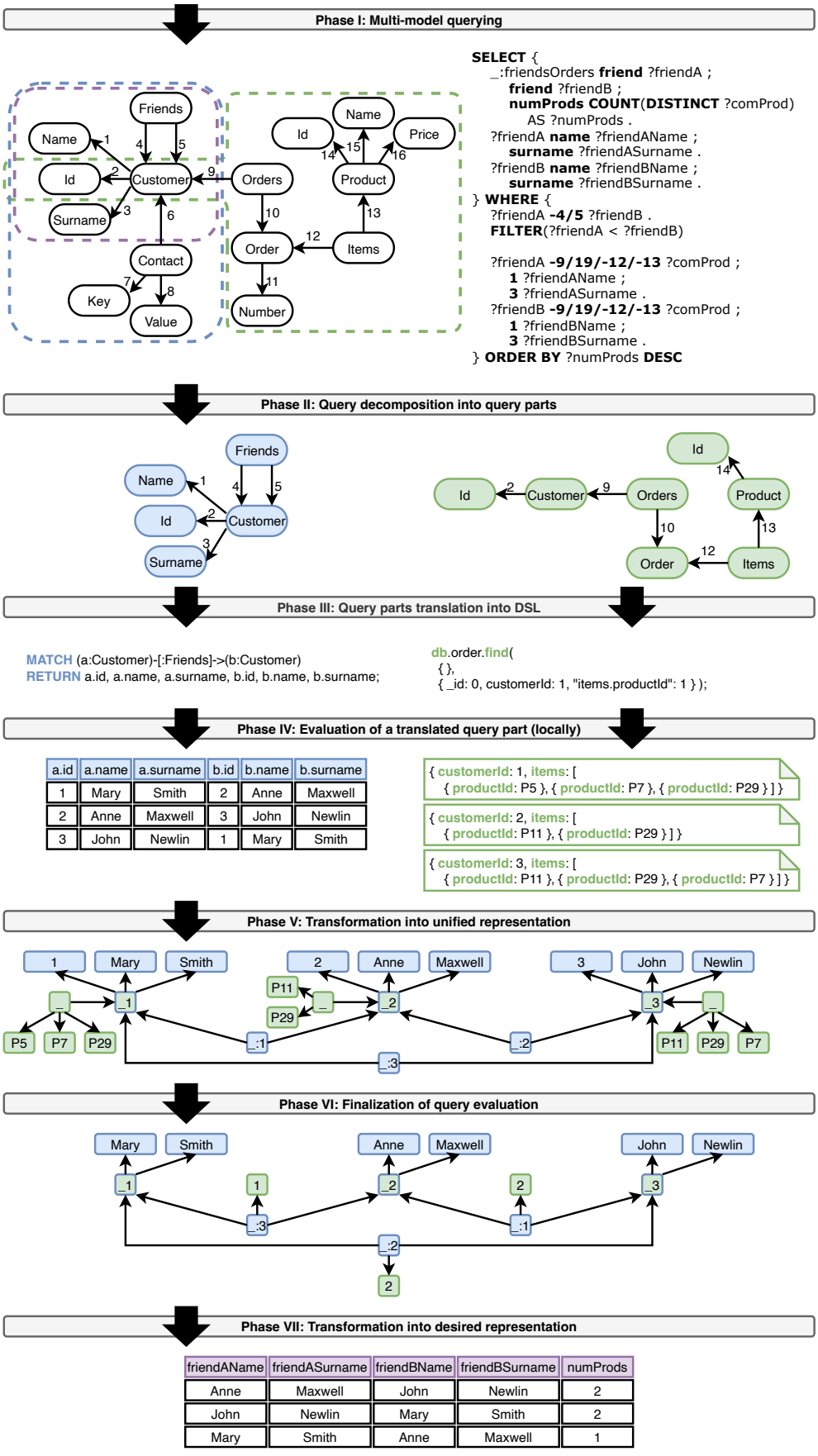


Figure 5.1: A diagram showing the full querying workflow [32].

- The interface should be general enough to allow the addition of any database system which we may reasonably expect to support;
- The interface should abstract away all categorical concepts from the wrappers' implementations where possible to make supporting new databases easier; and
- The order in which wrapper interface methods are called should be irrelevant.

Perhaps the last goal is worth elaborating on – since MMQL allows the definition of statements in any order, it is also desirable for our query translation algorithm to be able to process these statements in any order. For this reason, all methods in the wrapper interface will make the wrapper simply "remember" this operation, and finally, when the entire query is processed, a finalization method will be called on the wrapper, building the native query. In this way, the database wrappers operate similarly to a builder object in the builder design pattern [34]. With these goals in mind, we propose the following methods for the generic database wrapper interface:

- `defineKind(`$kindId$`,` $kindName$`)` which is an initialization function, assigning the identifier $kindId$ to the kind name $kindName$. The kind name refers to for example table names in PostgreSQL or collection names in MongoDB. Note that a single kind name can be registered multiple times under different identifiers, in which case its data is retrieved independently multiple times.
- `addProjection(`$propertyPath$`,` $kindId$`,` $variableId$`)` which adds projection of a specific property to a query, i.e. given a specific kind, this method will add the specified property of this kind to the native query output.
- `addFilter(`$variableId$`,` $filterOp$`,` $constant$`)` which constrains the value of a property identified by $variableId$ to a logical relation specified by $filterOp$ (such as equality or inequality), using $constant$ as the second operand. Note that the variable $variableId$ does not need to exist before this method is called, as the validation is delayed until the moment of generation of the native query. In addition, we also consider an overload `addFilter(`$propertyPath$`,` $kindId$`,` $filterOp$`,` $constant$`)` which does not require its filtered property to be a part of the projection, instead specifying the path directly. This overload is necessary because of triples which contain constant literals.
- `addFilter(`$variableId_1$`,` $filterOp$`,` $variableId_2$`)` which creates a relationship between the values of two variables, eliminating solutions where this relationship is not satisfied. This can be used together with aggregations, whose results have their own variable identifiers.
- `addValueSet(`$variableId$`,` $constants$`)` which constrains the value of a property identified by $variableId$ to the set of values provided in $constants$.

- `addJoin(`$kindId_1$`, `$kindId_2$`, `$joinProperties$`)` which performs a join operation on the two kinds provided. Note that we call it a join operation, but for example, in the case of the graph model, this entails a graph traversal rather than a join in the relational sense.

- `addRecursiveJoin(`$recursiveJoinPath$`)` which performs a series of recursive joins between various kinds in the query part (if recursion is supported). If we have a morphism with repetition, its domain must necessarily be the same as its codomain, otherwise it would not be possible to repeat it. This kind of recursive joining is useful in cases of transitive relationship traversal (for example finding all superiors of a given company employee), and it is generally supported in the relational model with recursive common table expressions[1], and in the graph model with variable length paths[2].

- `addAggregation(`$aggregationType$`, `$aggregationRootKindId$`, `$rootIdPaths$`, `$kindId$`, `$propertyPath$`, `$variableId$`)` which performs an aggregation of the specified type on the specified property of the specified kind. The aggregation may have multiple possible roots (for example averaging the item price for a set of customers, each of whom has made multiple orders with multiple items - should we average the items per order, or per customer?), and for this reason, the aggregation root is also specified, along with one of its identifiers. The result of this aggregation is stored in the variable identified by $variableId$.

- `buildQuery()` which takes the information collected from all method calls on this wrapper and generates the native database query based on it. This method contains the majority of the actual wrapper logic, with the other methods mostly just storing the arguments for processing by this method. The result of this method is a tuple ($nativeQuery$, $variableMap$), where $nativeQuery$ is the built native query in a string representation, and $variableMap$ contains a map of variable identifiers included in the projection to property paths within the native query result.

Naturally, not all databases support all of these operations, for example Cassandra does not support joins[3]. For this reason, each database wrapper also exposes the following properties about the functionality the underlying database has:

- `isJoinSupported` which specifies whether the database supports (inner) joins. For example, most relational databases do, as do graph databases in the form of graph traversals. Document databases may support joins, but for example MongoDB only supports left outer joins. Naturally, the inner join operation can be emulated using the left join operation, but the wrapper implementer may not want to support this due to the involved complexity. Finally, some databases like Cassandra do not support joins of any kind.

[1] `https://www.postgresql.org/docs/current/queries-with.html`

[2] `https://neo4j.com/docs/cypher-manual/current/syntax/patterns/#cypher-pattern-varlength`

[3] We only consider the built-in capabilities of the given databases, not user-defined procedures.

- `isOptionalJoinSupported` which specifies whether the database supports optional (left outer) joins. This is necessary because of the `OPTIONAL` clause in MMQL. In the situation that optional joins are not supported, the joins must be performed at the categorical level.

- `isNonIdFilterSupported` which specifies whether the database supports filtering of properties outside of the identifier set of the root object of a given kind. For example, many key-value databases treat stored values as opaque objects, not allowing filtering based on their value, but only their retrieval.

- `isCountAggregationSupported` which specifies whether the database supports the `COUNT` aggregation. We also define a corresponding property for each type of aggregation supported in MMQL.

- `isRecursionSupported` which specifies whether the database supports recursive queries, which may necessitate the recursive joining of kinds. Such joins are generally supported in relational and graph databases, and allow the traversal of tree or graph structures in a single query.

For each interface method mentioned, we will also define an optional counterpart which means that this operation should be included in the query, but with optional semantics. In some cases, the implementations will be identical (for example in SQL, adding projection on an optional column will return `NULL` if that column is not set for a particular row). However, in some cases, there may be differences (for example in SQL, we would generate an inner join for a non-optional join operation, whereas we would generate a left outer join in the optional case). When defining the interface methods, we used a set of arguments which must also be explained in detail:

- *propertyPath* is an ordered list of property names, forming a traversal from the kind root to a leaf property. This property path, along with name information (for example, whether the name is dynamic), also carries the property type at each level, in order to allow database wrappers to correctly deal with nested objects and arrays. The argument *rootIdPaths* is a list of property paths.

- *kindId* is the identifier of the kind to which this operation applies. This identifier can be mapped to the kind name configured in the database wrapper at initialization time. The inclusion of an identifier instead of a name allows us to query the same kind multiple times within a query part, for example joining customers together with other customers based on the equality of their names. Similarly, *kindName* refers to the name of the kind identified by *kindId* in its respective database.

- *variableId* identifies the output field of the native query. We cannot generally make assumptions about how the query output may be structured, as this decision is ultimately up to the database wrapper. For this reason, whenever defining a property which should be part of the native query output, we assign it a variable identifier. When the native query is compiled,

the database wrapper returns a mapping of variable identifiers to concrete property paths within the query output, allowing us to parse this output into a categorical representation correctly.

- *joinProperties* contains a set of tuples, with each tuple containing a property path within the corresponding kind. These tuples define join points for the join operation. Note that a limitation of our approach is that we only support joins on *equality*. In theory though, joins with more complicated conditions could also be supported.

- *aggregationType* specifies the type of aggregation which should be performed, like counting or averaging.

- *recursiveJoinPath* is a list of join path segments, where each join path segment may either be a tuple ($kindId_1$, $kindId_2$, *joinProperties*) symbolizing a join between two kinds, a path segment with a repetition specifier (?, $*$ or $+$ as in regular expressions), or multiple path segments chained together. In this way, we can represent arbitrary morphism paths with repetitions as sequences of joins between kinds, with some subsequences optionally repeated as necessary. Note that whenever a path segment is repeated, its first kind must be the same as its last kind in order for the repetition to be valid.

With the database wrapper interface specified, we can see that in its simplest form, implementing a database wrapper is actually rather straightforward. In order to support a new database in the most minimal form possible, we can simply implement the function `addProjection`, at which point we will be able to query any data from this database, even though we would not be able to filter this data or perform joins and aggregations at the database level, forcing them to be executed in a slower way at the instance category level.

5.2 Algorithms

In Section 5.1, we described the high-level steps involved in our proposed approach for the implementation of MMQL. In this section, we will examine the outlined steps in greater detail, specifying a concrete algorithm for each, as well as discussing the presented algorithms and their weaknesses.

Before we start discussing the particulars of the individual algorithms, we need to discuss one key characteristic of our proposed approach. As we attempted to divide the whole approach into a number of well-defined, easily digestible algorithms, we were faced with a decision. On one hand, we could try to propose an algorithm which attempts to translate the query as a whole into native database queries, taking into account all parts of the MMQL query such as projection and ordering. However, as we discovered during the design process, such an approach would result in a monolithic, very complex algorithm which is hard to reason about, making its modifications or analysis impractical. For this reason, we opted for a simplification in this matter – when creating a query plan and dividing the query into query parts, we only consider the contents of the query's

`WHERE` clause, as this clause is actually the only truly necessary part for the retrieval of data from databases. In this way, we are able to limit the query part translation algorithm to a set of graph patterns and a few other concepts. This has some implications for the possible performance of our approach, as all query elements outside the scope of the `WHERE` clause must be handled by the MMQL query engine, which is undoubtedly much less efficient than using native database queries where possible. However, as a whole, this allowed us to greatly simplify and modularize the proposed algorithms.

In general, we will discuss performance optimization possibilities when presenting the algorithms, but we will not include these optimizations in the algorithms themselves for the sake of simplicity.

5.2.1 Query Preprocessing

As mentioned in the introduction to this chapter, when constructing native database queries, we will only consider the `WHERE` clause of the MMQL query. However, before we proceed with the main parts of the querying process, we need to perform a couple of additional steps first. One thing which needs to be done is general adjustments to the query structure to make its processing easier in the following steps. However, as the `WHERE` clause has the semantics of inducing a schema category, we will also need to construct this schema category for further usage. In addition to constructing this schema category, we will also need to construct the corresponding mappings. Lastly, query validation should be a part of this process, like the validation of variable types. We will not present any validation algorithms as they would be relatively straightforward, and for the rest of this chapter, we will assume that input queries are well formed and valid. The full query preprocessing algorithm is shown in Algorithm 5.1, and we will shortly discuss its constituent parts in greater detail.

Query Modifications

In Chapter 4 during the introduction of MMQL, we mentioned its power in the form of graph patterns, including constructs like ending triples with a semicolon instead of a period to repeat the same subject in multiple triples. However, as these constructs are simply syntactic sugar, we convert them to sets of triples with an explicitly specified subject to simplify their further processing (lines 1-9 in Algorithm 5.1).

The most interesting part of the query preprocessing step is *compound morphism decomposition* (lines 10-20 in Algorithm 5.1). Recall that in MMQL, we may specify compound morphisms using the syntax `55/31`, meaning the traversal of morphism with signature `55`, followed by the traversal of morphism with signature `31`. Again, as we attempt to make the query processing simple, we decompose all compound morphisms into base morphisms[4], inserting technical temporary variables with unique names in between the base morphisms. An example of this can be seen in Figure 5.2, where we can see a triple containing a compound morphism, and the decomposed representation of this triple. As far

[4]For the purposes of the algorithms in this chapter, we will consider base morphisms to be traversals of a base morphism in any direction.

Algorithm 5.1: Query Preprocessing Algorithm.

```
   Input: SELECT – query SELECT clause
          WHERE – query WHERE clause
 1 foreach graph pattern P in SELECT ∪ WHERE do
 2     if P has repeated subject subject and morphism morphism then
 3         delete P from query
 4         foreach object in P do
 5             add triple subject morphism object to query
 6     else if P has repeated subject subject then
 7         delete P from query
 8         foreach morphism, object in P do
 9             add triple subject morphism object to query
10 foreach triple subject, morphism, object in WHERE do
11     if morphism is compound and not repeated then
12         b := getBaseMorphisms(morphism)
13         delete triple subject morphism object from WHERE
14         prevVar := subject
15         foreach baseMorphism in b do
16             if baseMorphism is the last element in b then
17                 newVar := object
18             else
19                 newVar := getUniqueVarName()
20             add triple prevVar baseMorphism newVar to WHERE
21 foreach triple subject, morphism, object in WHERE do
22     if morphism is base then
23         if morphism is −oppositeMorphism then
24             delete triple subject morphism object
25             add triple object oppositeMorphism subject
```

as MMQL is concerned, both of these constructs are semantically equivalent. Finally, for each morphism, we can also traverse it in the opposite direction with the MMQL unary minus operator. Without loss of generality, we can reverse the direction of traversal for base morphisms along with swapping the subject and object, which simplifies further processing of these opposite-direction traversals (lines 21-25 in Algorithm 5.1).

```
WHERE {
    // Compound morphism before preprocessing
    ?customer 55/31 ?orderNumber .

    // Decomposed compound morphism with inserted variable
    ?customer 55 ?51d747110d95 .
    ?51d747110d95 31 ?orderNumber .
}
```

Figure 5.2: Compound morphism decomposition.

Note that compound morphism decomposition only applies to compound morphisms without repetition. When morphisms with repetition are encountered, they are left intact, as it may be necessary to form recursive queries in this instance, and it is not possible to decompose morphisms with infinite upper bound repetition.

The inquisitive reader may raise the question of optimization – is it possible for the query engine to omit "useless" data along the path of the compound morphism, only retrieving the data necessary for the query variables? However, recall Chapter 2 where we introduced the notion of the schema and instance category. The instance category must conform to its corresponding schema category, meaning that if our schema category has base morphisms `55` and `31`, these morphisms must also exist as instance morphisms, making this kind of optimization impossible if we intend to stick with the categorical model throughout the process. Not all hope is lost though, as there is a way to make this optimization work. If the query engine notices that a particular set of morphisms is only ever traversed together in a compound morphism, it could perform a contraction of these morphisms in the schema category, defining a new schema category where this compound morphism is transformed into a single base morphism. In this fashion, the instance category would now contain the formerly compound morphism as a base morphism, allowing the query engine to omit irrelevant data from the query plan.

Constructing the Schema Category

As we mentioned in the introduction of this subsection, before we may continue with the rest of the process, we must first construct the schema category induced by the `WHERE` clause, which will be used in the following subsections. The original schema category which the query was made with is therefore only used as the input for the algorithm for the construction of the schema category induced by the `WHERE` clause, which we will introduce shortly. The same applies to the

original mappings, which must also be adjusted. However, before we explain *how* this is done, we should mention *why* it needs to be done. In fact, if we have a query which contains a maximum of one variable per schema object, this step is totally unnecessary. The reason for the existence of this step is queries which have multiple variables corresponding to the same schema object (for example, recall Figure 4.6, showing a query selecting two different customers who purchased an item with the same name). In this case, we need a way to separate the values of both variables, as their possible sets of values may not be the same.

To solve the issue of multiple variables per schema object, we propose a modification to the original schema category. We will not provide an explicit algorithm for this, as its creation from the following definitions is straightforward. Let $\mathbf{S} = (\mathcal{O}_\mathbf{S}, \mathcal{M}_\mathbf{S}, \circ_\mathbf{S})$ be the original schema category (recall Section 2.3), and let $\mathbf{Var}_{WHERE}$ be the set of variables in the query `WHERE` clause. We will define $\mathbf{S}_{WHERE} := (\mathcal{O}_{\mathbf{S}_W}, \mathcal{M}_{\mathbf{S}_W}, \circ_{\mathbf{S}_W})$ For each $var \in \mathbf{Var}_{WHERE}$ and $o \in \mathcal{O}_\mathbf{S}$, we define a schema object $o_{var} \in \mathcal{O}_{\mathbf{S}_W}$ whose instances will contain the values of variable var. Additionally, as the query does not need to contain all (or any) identifiers of any particular schema object which is part of the query, we also add all schema objects necessary for all identifiers of all objects $o_{var} \in \mathcal{O}_{\mathbf{S}_W}$ to $\mathcal{O}_{\mathbf{S}_W}$. These additional objects are inserted separately for each $o_{var} \in \mathcal{O}_{\mathbf{S}_W}$, they are not shared in the case that multiple variables refer to the same schema object in $\mathbf{S}$.

For each $o_{var1}, o'_{var2} \in \mathcal{O}_{\mathbf{S}_W}$ and $m = (s, o, o', min, max) \in \mathcal{M}_\mathbf{S}$, if the triple `?var1 s ?var2` exists in the `WHERE` clause, then we define
$m_{var12} := (s_{var12}, o_{var12}, o'_{var12}, min, max) \in \mathcal{M}_{\mathbf{S}_W}$. In addition, just as we did with schema objects, we will define the required morphisms to satisfy all identifiers of all $o_{var} \in \mathcal{O}_{\mathbf{S}_W}$ to be part of $\mathcal{M}_{\mathbf{S}_W}$. Using these definitions, we will also consider morphism signatures s to be equivalent to s_{var12} when used in triples of the form `?var1 s ?var2` in the rest of this chapter when working with $\mathbf{S}_{WHERE}$. This allows us to reason more simply about the query processing algorithms, as we can keep using signatures specified in the query to refer to corresponding signatures from $\mathbf{S}_{WHERE}$, since given a triple, this mapping is always unambiguous.

Finally, we define $\circ_{\mathbf{S}_W}$ to be the natural extension of $\circ_\mathbf{S}$ over $\mathcal{M}_{\mathbf{S}_W}$. We show an example of the construction of this induced schema category using a schema category shown in Figure 5.3 and a query shown in Figure 5.4, with the schema category induced by the query's `WHERE` clause being shown in Figure 5.5. Note that the `Surname` schema object is missing, as it is not part of the query's `WHERE` clause, nor is it part of any object's *ids*.

We need to mention that the `WHERE` clause may contain the `OPTIONAL`, `UNION` and `MINUS` clauses in addition to graph patterns and filters. As far as the generation of the new schema category goes, `OPTIONAL` is quite simple, as we simply consider its contents the same way as non-optional graph patterns, save for modifying the morphism cardinality at join points to have a minimum of zero, as the optional parts may or may not exist. It is remarkably similar for `MINUS`, where we can treat the forbidden pattern as an optional pattern, using it to filter the final result set. In the case of `UNION`, we simply consider both of its operands to be optional as far as the schema category is concerned.

As for nested queries, they are evaluated recursively from the most nested to the least nested, and their results are joined to the containing query's `WHERE` clause. This means that at the level of the schema category, we simply need to

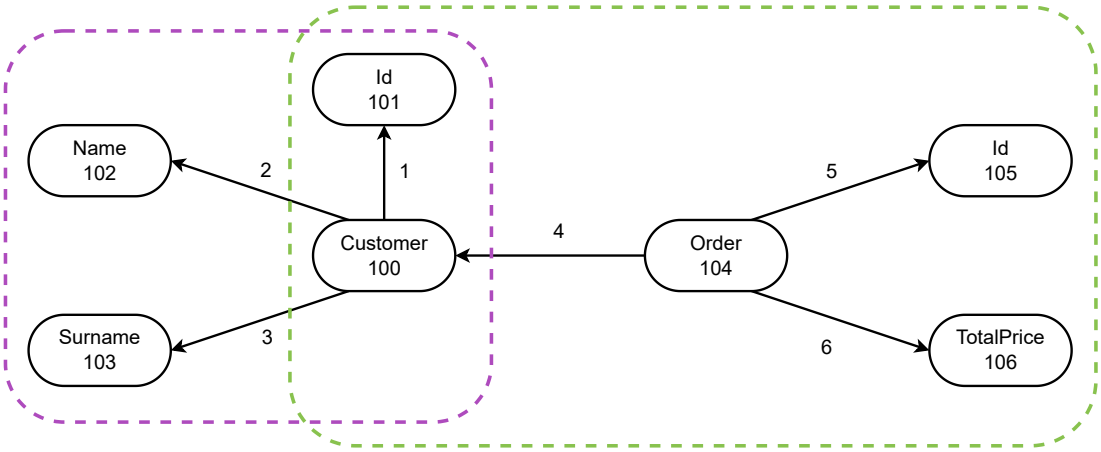


Figure 5.3: A sample schema category showing a set of customers in a relational database, and a set of orders in a document database, with each order containing the identifier of the ordering customer.

```
SELECT {
    _:shared name ?sharedName ;
        price ?sharedPrice .
}
WHERE {
    ?customer1 2 ?sharedName ;
        -4 ?order1 .

    ?customer2 2 ?sharedName ;
        -4 ?order2.

    ?order1 6 ?sharedPrice .
    ?order2 6 ?sharedPrice .
}
```

Figure 5.4: A query returning two customers with the same name who placed an order with the same total price, corresponding to the schema category shown in Figure 5.3.

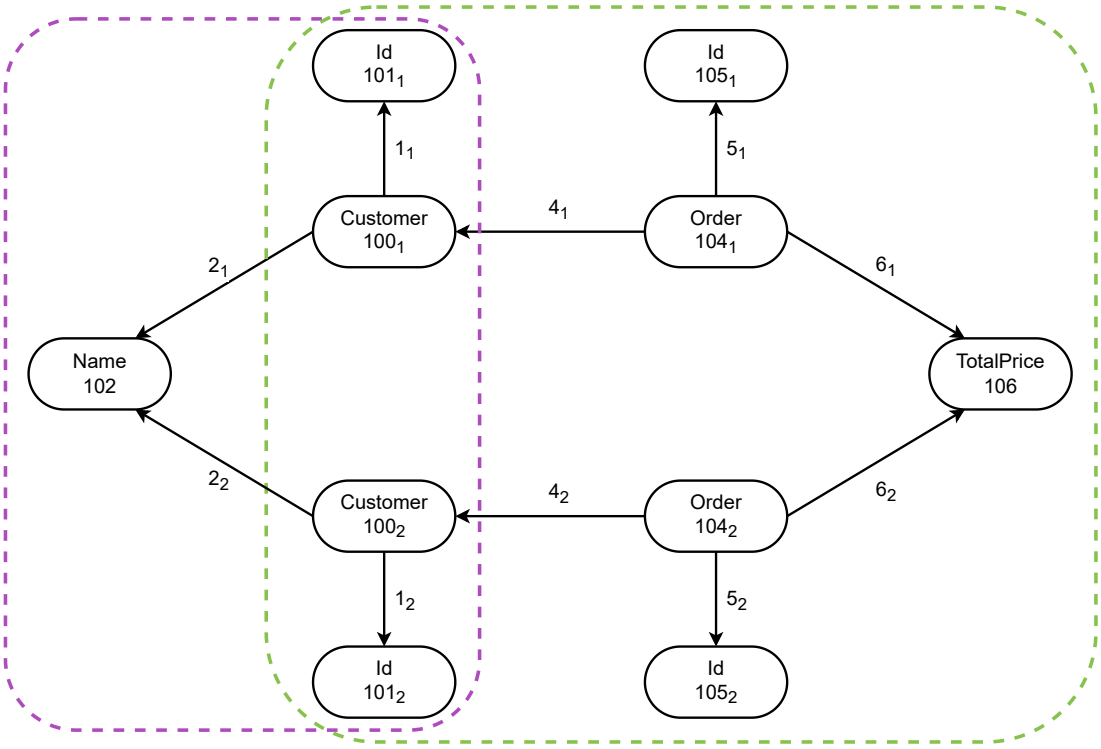


Figure 5.5: Schema category induced by the WHERE clause of the query shown in Figure 5.4.

define the appropriate schema objects corresponding to the results of the nested query.

Constructing New Mappings

With the new schema category $\mathbf{S}_{WHERE}$ in hand, we naturally also need to adjust the set of mappings $\mathfrak{M}$ corresponding to $\mathbf{S}$ to be valid for $\mathbf{S}_{WHERE}$. This is necessary because these modified mappings $\mathfrak{M}_{WHERE}$ will be used for the generation of query plans and the translation of query parts to native database queries, both of which operate on the $\mathbf{S}_{WHERE}$ schema category. Before continuing, it is recommended that the reader is intimately familiar with the concepts introduced in Section 2.5. It is also worth remembering that for queries with at most one variable per schema object, this operation is effectively a no-op, and its purpose is to correctly handle queries with multiple variables per one schema object.

As we mentioned earlier, we restrict our approach to queries for which every property selection from a kind must necessarily contain the entire property path from the root. For this reason, it is sufficient to examine only kinds whose root objects are present in the query.

Given $\mathfrak{M}$, let us define $\mathfrak{M}_{WHERE}$ in the following way. For each mapping $\mathfrak{m} \in \mathfrak{M}$ with root object o, discard it if its root object is not part of the query. Otherwise, for each $o_{var} \in \mathcal{O}_{\mathbf{S}_W}$, we define $\mathfrak{m}_{var} \in \mathfrak{M}_{WHERE}$. For each property $\phi \in \mathfrak{m}$, we define the corresponding property $\phi' \in \mathfrak{m}_{var}$ recursively by replacing occurrences of signatures $s \in \mathbf{S}$ with signatures $s_{varxy} \in \mathbf{S}_{WHERE}$ for the corresponding objects $o_{varx}, o_{vary} \in \mathbf{S}_{WHERE}$. This formulation seems complicated, but when explained with plain words, it is actually rather straightforward – for

each schema object in $\mathbf{S}_{WHERE}$ which is the root of some kind's mapping, we are simply constructing an equivalent mapping, but using the new variable-specific objects and morphisms from $\mathbf{S}_{WHERE}$. For example, let us consider a query selecting two different customers, with the customer kind having its own mapping. This definition would create two instances of this mapping, each for one of the customer variables, in such a way that when querying the customers' data from the database, the correct data ends up in the correct customers' instance objects, taking into account that some may be shared due to the shape of the query. We can see an example of such mappings in Figure 5.6, which correspond to the schema category shown in Figure 5.5.

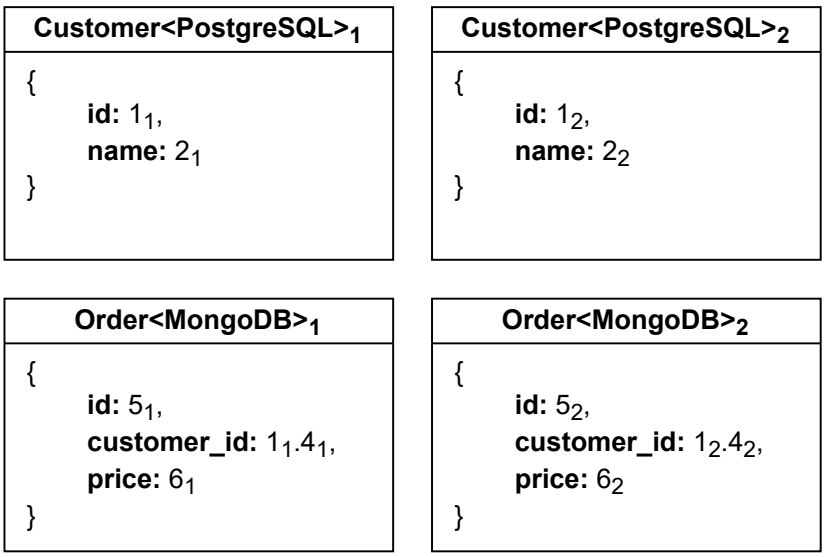


Figure 5.6: Mappings corresponding to the induced schema category shown in Figure 5.5.

Note that it may be necessary for a single property ϕ to produce multiple properties ϕ', ϕ'' and so on. This can happen in the specific cases of nested arrays in the mapping, as we are able to bind multiple distinct variables to different elements of the array. We do not need to give this scenario any special consideration, as long as we presume that given ϕ' or ϕ'', we are able to determine ϕ, allowing us to correctly construct the appropriate database query during the translation of query parts.

5.2.2 Query Plan

Now that the query has been preprocessed, the query engine must first create a set of query plans dictating how the query will be executed (recall Phase II in Figure 5.1). In simple terms, we consider a query plan to be a plan of which data will be retrieved from which database, and in what way. When we consider the situation of no data redundancy, we only need to consider a single query plan on the level of MMQL, as there is only one database from which we can retrieve any given kind. Naturally, the databases' query planners will come into play under the covers, and during the execution of native database queries, the database planners will certainly consider a number of different plans. However, in

the context of MMQL, the query planner only needs to decide where to retrieve the data from, and the optimization of native queries is left up to the databases themselves. When we consider data redundancy, there will be multiple query plans generated on the MMQL side.

More formally, we consider a query plan to be a set of query parts, such that each distinct base morphism which is part of the MMQL query `WHERE` clause, is assigned to exactly one query part. A query part is a set of mappings, assigning base morphisms to kinds defined in mappings. This means that for each morphism in the query, we make a decision on which kind this morphism should be retrieved from. The algorithm for the creation of query plans is shown in Algorithm 5.2. In simple terms, for each base morphism in the query, it retrieves the set of kinds of which this morphism is a part (lines 1-7), and then generates a set of query plans by Cartesian product of all morphism-kind assignments (lines 8-12).

Algorithm 5.2: Query Plan Generation Algorithm.

Input: **WHERE** – query WHERE clause
$\mathfrak{M}_{WHERE}$ – set of mappings adjusted for the schema category induced by the WHERE clause

```
// Map listing the set of possible source kinds for each morphism
1  morphismKindMap := {}
2  queryPlans := [ ]
3  foreach triple subject morphism object do
4      foreach mapping m in M_WHERE do
5          namePath := getPropertyPath(morphism, m)
6          if namePath is not NULL then
7              morphismKindMap.append(morphism, m)
8  kindAssignments := cartesianProduct(morphismKindMap)
9  foreach kindAssignment in kindAssignments do
10     queryPlan := createQueryPlan(kindAssignments)
11     if queryPlan is not NULL then
12         queryPlans.append(queryPlan)
```

There are a couple interesting functions being called in this algorithm which are worth mentioning. The first of them is `getPropertyPath`, which given a morphism and a kind's mapping, returns an ordered list of properties which form a traversal from the kind root to the morphism within the kind. If the morphism does not exist in the kind, this function returns `NULL`, which is why we can see this function being used to check whether a morphism lies in a given kind.

The next interesting function is `createQueryPlan`, which may be seen in Algorithm 5.3. This function accepts a mapping of morphisms to their source kinds, and returns a query plan for this mapping. This function's job is to divide the query into query parts, forming the query plan. It also validates the query plan, as it only makes sense to assign a morphism to be selected from a kind when the entire property path to this morphism is also being selected from the same kind. This has a very important consequence for the set of queries which our approach is able to process, as given this constraint, any MMQL query which is only operating on a subforest of a kind's access path which does not include the

kind root object is not valid. For example, given the schema category and access path shown in Figure 2.4, a query only selecting order items and their names, but not containing any variable for the `Order` schema object, would produce zero valid query plans as proposed. This is fixable by adding special cases for such queries to the algorithm, which might insert the necessary variables for a valid query plan to be generated. However, we omitted these special cases for the sake of simplicity.

Algorithm 5.3: Function `createQueryPlan` from Algorithm 5.2.

```
1 function createQueryPlan(kindAssignments):
      Input: kindAssignments – the source kind for each morphism
2     foreach morphism, kind in kindAssignment do
3         namePath := getNamePath(morphism, kind)
4         isPlanValid := isPathInKind(kindAssignment, kind)
5         if NOT isPlanValid then
6             return NULL
7     queryParts := getQueryParts(kindAssignments)
      // A query plan is a list of query parts
8     return queryParts
```

Additionally, we have the `getQueryParts` function, which performs the actual division of the query into query parts. We consider a query part to be a maximal connected component of kinds from the same database in the query in the case that the database's wrapper supports joins, or a single kind in the case that joins are not supported. The reason for this is that we need each query part to be convertible to a single native database query, and if joins are not supported at the database level (like in Cassandra, a columnar database), each kind must be separated into its own query part. This function is shown in Algorithm 5.4. There we can see a few more functions worth explaining being used, specifically `getConnectedComponents` which returns kinds contained within a given query part grouped by reachability, i.e. if there are multiple kinds within this query part which are not directly connected. This can happen if we are selecting multiple kinds from the same database, but in between them is a kind from a different database. There is also the `getDBWrapper` function, which returns a database wrapper for a given query part, as described in Section 5.1.1.

The perceptive reader may have noticed that the presented algorithm does not take into account morphisms with repetition, which is intentional. When we are dealing with morphisms with repetitions, things get considerably more complex, therefore they were omitted from the algorithms for the sake of simplicity. Repeated morphisms may require special support on the database level, which is why the database wrappers contain the `isRecursionSupported` property, telling us whether we can form recursive queries in this database. For example, this is possible in most graph databases with their path matching capabilities. If recursion is not supported on a database level or the repeated morphism is a compound morphism crossing database boundaries, the MMQL query engine would have no choice but to emulate this functionality by itself, and perform the required joins at the level of the instance category. This is not only extremely inefficient, but

Algorithm 5.4: Function `getQueryParts` from Algorithm 5.3.

```
1  function getQueryParts(kindAssignments):
       Input: kindAssignments – the source kind for each morphism
       // Initial query parts are kinds grouped by their database
2      queryParts := groupBy(kindAssignments, x => x.database)
       // If the kinds for a given query part do not form a single
          connected component, split this query part such that each
          new query part forms a connected component
3      foreach queryPart in queryParts do
4          disjointKindSets := getConnectedComponents(queryPart)
5          queryParts.remove(queryPart)
6          queryParts.extend(disjointKindSets)
       // If the database for this query part does not support joins,
          separate each kind into its own query part
7      foreach queryPart in queryParts do
8          W := getDBWrapper(queryPart)
9          if not W.isJoinSupported then
10             queryParts.remove(queryPart)
               // This is a simplified way of expressing that the
                  morphism mappings for each kind in queryPart are
                  split into a separate query part
11             foreach kind in queryPart do
12                 queryParts.append(kind)
13     return queryParts
```

also needlessly complicated, which is why we do not present a full algorithm for this scenario.

Another thing missing from the presented algorithm is handling of `OPTIONAL`, `UNION` and `MINUS` clauses, which is intentional. When constructing a query plan, we consider optional graph patterns as if they were not optional, assigning source kinds to morphisms. Similarly, both `UNION` operands are considered to be optional by the query planner, as either one of them may or may not exist in any given instance. Finally, the second operand of `MINUS` clauses is treated as if it was optional, attempting to match the pattern in the data, and removing the matching solutions.

As an example, recall the query from Figure 5.4, whose `WHERE` clause induced the schema category presented in Figure 5.5. We can see the division of the triples from this query's `WHERE` clause into query parts in Figure 5.7. Note that because PostgreSQL supports inner joins, both customers' data can be queried together in a single query part, but because MongoDB does not support inner joins natively[5], the queries for the two orders need to be separated into two query parts. The reader may have also noticed that there are some extra triples selecting the customers' identifiers, the addition of which we will discuss in Section 5.2.3.

[5]MongoDB does support the left join operation, which makes it possible to implement an inner join as well, but for the purposes of this example, we consider MongoDB as not supporting inner joins.

```
Query Part 1 (Customer<PostgreSQL>1, Customer<PostgreSQL>2) {
    ?customer1 2_1 ?sharedName .
    ?customer1 1_1 ?customerId1 .

    ?customer2 2_2 ?sharedName .
    ?customer2 1_2 ?customerId2 .
}
Query Part 2 (Order<MongoDB>1) {
    ?order1 4_1 ?customer1 .
    ?customer1 1_1 ?customerId1 .
    ?order1 6_1 ?sharedPrice .
}
Query Part 3 (Order<MongoDB>2) {
    ?order2 4_2 ?customer2 .
    ?customer2 1_2 ?customerId2 .
    ?order2 6_2 ?sharedPrice .
}
```

Figure 5.7: Decomposition of triples into query parts based on the query shown in Figure 5.4.

5.2.3 Join Plan

With a set of query plans generated by the algorithm presented in the previous subsection, one may think that we are ready to select the best query plan and continue with query execution. However, there is one more thing that needs to be done beforehand, as it may have an impact on the choice of the best query plan. We are talking about *join plans* – since we are dealing with multi-model data, it is likely that we will have a query with multiple query parts, with the necessity of joining their results together to get the full query result.

In general, join order optimization is a known hard problem [35], with multi-model join optimization is even harder with relatively little related work to fall back on, which is why attempting to solve it would be outside the scope of this thesis. For this reason, in this subsection, we will discuss join plans in a slightly different fashion. For our purposes, a join plan is a set of join points in the query, where each join point consists of neighboring kinds, as well as the *join identifier*. The join identifier is one of the *ids* (recall Section 2.3) of the schema object which the given kinds need to be joined on.

In general, the properties which are a part of the *ids* of any given schema object do not need to be a part of the query, even if the object itself is a part of it. However, if this object happens to be the join point between two kinds, this would pose an issue, because we need one of that object's identifiers in order to determine how to join the data from both kinds. For this reason, if none of the object's identifiers are a part of the query, we will need to insert the selection of at least one such identifier to make the join possible. Note that there can even be multiple objects which are part of a single join point between two query parts, which is why we need to consider all of them. Therefore, in this subsection, we present Algorithm 5.5, where we show our proposed approach of finding all join

points for a given query plan, and the selection of the necessary identifiers.

The join plan algorithm identifies join points by looking for specific patterns in the query plan (line 5 in Algorithm 5.5). These patterns involve two neighboring query parts, having a triple from each query part, with both triples sharing a schema object. This shared schema object which forms the intersection of the query parts must be part of both triples, but the direction of the triples is irrelevant, i.e. the intersection object may be either the subject or object of the given triples. Once the intersection object is found, its identifiers are examined, and an identifier is selected which may be queried using both kinds at the join point. Then, the selection of this identifier from the intersection object is added to both query parts in the form of triples with unique variable names (if not already present).

A special case to consider is when the intersection object has a signature of ϵ, meaning the object is identified only by its value. This means that we are joining two query parts by the value of an attribute, like joining two different customers on the equality of their name. In this case, no extra work is needed, as the value necessary for the join is already being selected.

Algorithm 5.5: Join Plan Algorithm.

```
   Input: queryPlan – the query plan for which we need to create a join plan
 1 foreach neighboring query parts q1, q2 in queryPlan do
 2     foreach triple t1 in q1, triple t2 in q2 do
 3         m1 := t1.morphism
 4         m2 := t2.morphism
           // The parentheses in this pattern mean schema objects, and
              the labeled dashes represent morphisms.  We match the
              pattern regardless of the direction of traversal of the
              morphism, which is why they are depicted as undirected.
 5         if t1 and t2 match the pattern () -m1- (I) -m2- () then
 6             ids := getIds(I)
 7             k1 := getKind(t1)
 8             k2 := getKind(t2)
 9             foreach id in ids do
10                 if id is ϵ then
11                     queryPlan.joinPoints.add((q1, q2, I, id))
12                     break
13                 if id can be selected from k1 and k2 then
14                     r1 := getRootObject(k1)
15                     r2 := getRootObject(k2)
16                     foreach prop in id do
17                         if prop is not selected from I in q1 then
18                             add triples selecting prop from I to q1
19                         if prop is not selected from I in q2 then
20                             add triples selecting prop from I to q2
21                     queryPlan.joinPoints.add((q1, q2, I, id))
22                     break
```

Finally, we mentioned earlier in this subsection that we do not attempt to solve the hard problem of multi-model join order optimization. In our implementation of the core parts of these algorithms (described in Chapter 6), we will mention that we are using software called MM-evocat [22], which provides us with the functionality of merging instance categories together. For this reason, we do not need to worry about coming up with a basic replacement for a join order optimizer, because if we send query parts to MM-evocat to process in parallel, MM-evocat decides the order of the joins, relieving us of the responsibility.

5.2.4 Picking the Best Query Plan

As a result of our work in the last subsection, we now have a set of query plans, each of which has an associated join plan (which does not define the order of joins, but rather the set of joins itself). The next thing we need to do is we need to pick a query plan. However, this turns out to be much more complicated than it sounds. Single-model query planning and optimization are already known to be a hard problem [36]. When we move into the world of multi-model data, polystores like BigDAWG[6] include multi-model query planners and optimizers, but these do not fit neatly to our problem domain due to the number of extra considerations owing to our categorical data representation. For this reason, proposing an approach for the problem of multi-model query planning and optimization is outside the scope of this thesis. However, for the benefit of future work on this subject, we will discuss the possible inputs which could be used for the development of a suitable multi-model query planner for MMQL.

When it comes to the information we already have, the query planner could take into account various features of the query plans themselves. For example, the number of databases used in the query may be relevant, as it is likely that a query which utilizes 4 databases to retrieve the same data as a query utilizing only 2 databases may be less performant. The specific databases used may also be relevant, as the query planner could use information about the current load on the databases to select a plan which utilizes databases which are not overloaded. Another piece of information for a multi-model query planner to consider is the amount of work necessary to perform additional work on the MMQL query engine side, like projection, ordering, or deferred statements which will be introduced shortly in Section 5.2.5. The cost of these tasks is prohibitively high, especially for large amounts of data, which is why they should be avoided at all costs.

In addition, a multi-model query planner working with multiple databases can rely on the single-model query planners native to those databases. Most databases expose some kind of functionality which allows users to access information about the query planner. For example, SQL has the `EXPLAIN` keyword, which when used together with a query, will return information like the total plan cost (generally expressed in abstract units, but sometimes also in units like the approximate number of disk accesses necessary), information about which indexes the query is using, or the approximate number of rows which may be returned by the query. In this way, the query planner might avoid slow queries which do not have indexes available, and prefer faster queries which are able to use indexes. Similarly, the Neo4j query planner can estimate the number of rows returned by various stages of

[6] `https://bigdawg.mit.edu/`

the query. Examples of the outputs of the PostgreSQL and Neo4j query planners can be seen in Attachment A.2 and Attachment A.4 respectively. Note that in order to make use of the generated native queries, the query planner would need to defer the choice of best plan one step further in the process, meaning query translation would need to happen for each generated query plan, and only then would the best plan be selected.

For some databases, the information returned by their query planner may be less useful however. For example in MongoDB, the query planner is able to say whether a given query will use an index without actually executing the query, but does not give any kind of cost estimate. To get the cost associated with a given query, one must execute the query using an explain command, which runs the query and only then returns the cost. We can see examples of both modes of operation of the MongoDB query planner in Attachment A.3. In general, a multi-model query planner should ideally take into account all of this information to make an educated guess about the cost of each query plan, selecting the plan with the lowest cost for execution. Similarly, such a query planner should expose information about its decision making process to allow users to better understand the query plans, and potentially add indexes to the relevant databases to make queries more performant.

5.2.5 Translation

Now that we have constructed and selected a query plan consisting of query parts, we need to translate each query part into a corresponding native database query (recall Phase III in Figure 5.1). Even though we already mentioned this, it is such a crucial fact that we will remind the reader of this again – the query plan only concerns the `WHERE` clause of the query, describing a way to get to the result of this clause. This is an important fact to keep in mind throughout this subsection, as we will be operating with the newly defined schema category $\mathbf{S}_{WHERE}$, however this schema category *does not* represent the result of the query, but the result of only the query's `WHERE` clause. The following subsections will then transform the result of the `WHERE` clause into the result of the entire query. We also recommend that the reader is familiar with the concept of *database wrappers*, introduced in Section 5.1.1.

The query part translation algorithm is relatively simple at its core – for each query part in the selected query plan, iterate over the set of statements within this query part, and call the necessary function on the query part's corresponding database wrapper to produce a native database query. We should note that we have not defined what it means for a statement to be within a query part, as a query part is defined as a set of morphism-kind mappings. If the statement is a triple, we simply mean that the triple's morphism (base or compound) is located entirely within that query part. If a triple cannot be assigned cleanly to a single query part as it crosses query part boundaries, we consider this triple to be part of a set of *deferred statements* to be executed at a later point, which are described in more detail in Section 5.2.7. Similarly, `FILTER` statements may also cross query part boundaries, leading to them not being part of any single query part, instead being deferred for later execution.

Along with building the native query via the database wrapper, a mapping

builder is also used to construct the mapping which specifies how the result of this query maps to the categorical representation, specifically to the $\mathbf{S}_{WHERE}$ schema category described earlier in this chapter. The base algorithm for the processing of query parts is shown in Algorithm 5.6, where we can see the enumeration of all possible statements. The statements are processed one-by-one in an arbitrary order, incrementally giving the database wrapper and mapping builder commands about the things which the native database query needs to accomplish.

Algorithm 5.6: Query Part Translation Algorithm.

```
   Input: queryPlan – the query plan for which we need to translate query
          parts into native queries
 1 foreach queryPart in queryPlan do
 2     W := getDBWrapper(queryPart)
 3     MB := MappingBuilder()
 4     foreach kind in queryPart do
 5         W.defineKind(getId(kind), kind.name)
 6     processProjectionTriples(queryPart, W, MB)
 7     processJoinTriples(queryPart, W)
 8     processValuesStatements(queryPart, W)
 9     processFilterStatements(queryPart, W)
10
11     queryPart.nativeQuery, variableMap := W.buildQuery()
12     queryPart.mapping := MB.buildMapping(variableMap)
```

Translating Graph Patterns

As we can see in Algorithm 5.6, the core of the algorithm itself is not that complicated. However, the most crucial work is hidden in the functions called by this algorithm, with `processProjectionTriples` and `processJoinTriples` being the most important of them all. Without other clauses, MMQL would still function as a unified multi-model query language, albeit limited in certain aspects. For this reason, we will examine the `processProjectionTriples` and `processJoinTriples` functions in more detail. Note that earlier, we specified that a triple only belongs to a query part if all of its constituent base morphisms do. For this reason, we do not need to worry about triples which must be deferred in these functions.

In Algorithm 5.7, we can see the details of the `processProjectionTriples` function. This function processes the set of triples within a single query part, transforming graph patterns into database wrapper function calls. Recall that in Section 5.2.1, we preprocessed triples with base morphisms to always traverse their morphisms in-order, meaning we do not have to worry about reverse traversals outside repeated compound morphisms. This means that we can be confident that for any given triple, its subject is always a variable, and the object is either a variable or a constant. We also do not consider triples whose object is non-terminal, meaning it is either a constant or a variable with a primitive value, as these triples are later processed during the processing of join triples. We recognize three main situations when it comes to the set of triples within a query part:

1. The triple's *morphism* contains repetitions, but the entirety of *morphism* lies within a single query part. In this case, *morphism* needs to be translated into a combination of recursive joins and a projection, or it needs to be deferred if recursion is not supported. This would be implemented by the `processRepeatedMorphism` function, although we omit this function from the algorithms shown, as its implementation would be too long.

2. The triple's *object* is a constant c, in which case we need to add a filter to the triple's containing kind, forcing the specified property value to be equal to c. In case that non-identifier filtering is not supported (such as in key-value databases), this filtering must be deferred.

3. The triple's *object* is a variable, in which case we need to add projection of this variable to the corresponding kind.

Algorithm 5.7: Function `processProjectionTriples` from Algorithm 5.6.

```
1  function processProjectionTriples(queryPart, W, MB):
       Input: queryPart – the query part being translated
              W – database wrapper for this query part
              MB – mapping builder for this query part
2      foreach triple subject morphism object in queryPart do
3          if morphism is base then
4              if object is not terminal then
5                  continue
6              kind := getKind(morphism)
7              path := getPropertyPath(morphism, kind.mapping)
8              if object is constant then
9                  pathMorphism := getPathMorphism(path)
10                 if not W.isNonIdFilterSupported and pathMorphism not in
                     getRootObject(kind).ids then
11                     defer triple as a deferred statement
12                 else
13                     W.addFilter(path, getId(kind), =, object)
14             else
                   // object is a variable
15                 varId := getVarId(object)
16                 kindId := getId(kind)
17                 W.addProjection(path, kindId, varId)
18                 MB.defineVariable(varId, path)
19         else
20             processRepeatedMorphism(triple, queryPart, W, MB)
```

We also mentioned the `processJoinTriples` function, which we can see in Algorithm 5.8. This function also processes the set of triples in the query part,

just like `processProjectionTriples`, but instead of looking for patterns requiring projection, it looks for the required join points in this query part. Recall that in Section 5.2.3, we introduced the join plan generation algorithm, which looked for join points between query parts. The algorithm within the `processJoinTriples` function is remarkably similar in its structure, but with one key difference. At the level of joining query parts, we solved the join problem by selecting the required *ids* from each query part, relying on the algorithm which converts query part results into an instance category to do the joining using these identifiers. In this case, the join is happening at the native database level, which means that rather than selecting the correct object identifiers, we need to actually generate the database join. For this reason, we find all the join points between two kinds in the query part, and for each join point, we create joins using the correct identifiers. Also note that we do not need to use the mapping builder in any way in this function, as we are only performing the work required to join the kinds together at the native query level, but we are not modifying the results of the query.

Algorithm 5.8: Function `processJoinTriples` from Algorithm 5.6.

```
1  function processJoinTriples(queryPart, W):
       Input: queryPart – the query part being translated
              W – database wrapper for this query part
2      foreach triples t1, t2 in queryPart do
3          m1 := t1.morphism
4          m2 := t2.morphism
5          if t1 and t2 match the pattern () -m1- (I) -m2- () then
6              ids := getIds(I)
7              k1 := getKind(t1)
8              k2 := getKind(t2)
9              if k1 = k2 then
10                 continue
11             foreach id in ids do
12                 if id is ϵ then
13                     path_k1 := getPropertyPath(t1, k1.mapping)
14                     path_k2 := getPropertyPath(t2, k2.mapping)
15                     W.addJoin(getId(k1), getId(k2), (path_k1, path_k2))
16                     break
17                 if id can be selected from k1 and k2 then
18                     joinProperties := [ ]
19                     foreach prop in id do
20                         path_k1 := getPropertyPath(prop, k1.mapping)
21                         path_k2 := getPropertyPath(prop, k2.mapping)
22                         joinProperties.add((path_k1, path_k2))
23                     W.addJoin(getId(k1), getId(k2), joinProperties)
24                     break
```

Finally, even though we do not show their handling in the algorithm for simplicity, we need to discuss paths containing morphism repetitions using the ?, $*$

and + MMQL operators. In the case that such a morphism spans multiple query parts, we have no choice but to defer its processing to the instance category level, retrieving all instances of all kinds along the morphism path using native queries, and creating the corresponding instance morphisms manually. However, it is entirely possible for a morphism path with repetition to fit within a single query part – imagine an example scenario where we have a set of employees, and each employee also has a superior identifier containing the identifier of that employee's direct superior. In such a scenario, we may want to retrieve the set of all superiors for a given employee, transitively, which is certainly possible natively if the employees are stored in a relational table or a graph. For these cases, the database wrapper contains the `isRecursionSupported` field, which specifies whether we can process recursive queries at a native level. If so, then we can define recursive queries using the `addRecursiveJoin(`$recursiveJoinPath$`)` method of the database wrapper. The argument to this method is essentially a path consisting of joins between two kinds, where parts of the path may be repeated. For each join point, we must define the property paths for properties on both sides of the join point, just like when we were defining single joins. Additionally, recall that in the case of any repetitions, the source and destination kind must be the same to satisfy MMQL type constraints. This means that the concept of repeated paths works well with the other concepts like projection or selection, since adding projection for a given kind will naturally extend this projection to all instances of this kind, regardless of the number of repetitions used to arrive to them.

Now that we know how to process projection and join triples, we can look at Figure 5.8, where we can see the native queries generated for the query parts shown in Figure 5.7. Notice that the native queries are identical for both MongoDB query parts in this case, which means that the data will be retrieved twice, but inserted into different instance objects and morphism each time. Naturally, retrieving the same data twice is redundant (although in the general case, the data for multiple variables corresponding to the same schema object can be different), therefore the MongoDB query could be executed only once as an optimization, and its results could be used for both MongoDB query parts.

Translating Filtering Conditions

In MMQL, we support three kinds of filtering operations: `WHERE` clauses containing logic expressions, `VALUES` clauses containing a set of possible values for a given variable, and triples with a constant on one end, forcing the value of a particular property to be equal to that constant. We already addressed the triples with constants in Algorithm 5.7, which means that we still have to deal with `VALUES` and `WHERE` clauses.

Firstly, let us deal with the simpler case of `VALUES` clauses, which accept a single variable and a set of allowed values for this variable. The translation of this construct is quite simple, given that non-identifier filtering is supported by the underlying database. If this kind of filtering is not supported, naturally the processing of this clause will need to be deferred and emulated at the instance category level. The function `processValuesStatements` is shown in Algorithm 5.9.

With the `VALUES` statement out of the way, the last filtering condition we need to handle is `FILTER` statements, which are the most complicated. Not only do

```
// PostgreSQL query for Query Part 1
SELECT customer1.id AS customerId1,
    customer2.id AS customerId2,
    customer1.name AS sharedName
FROM customer AS customer1
    JOIN customer AS customer2
    ON customer1.name = customer2.name;

// MongoDB query for Query Parts 2 and 3
db.orders.aggregate([
    {
        "$project": {
            "id": 0,
            "price": 1,
            "customer_id": 1
        }
    }
])
```

Figure 5.8: Native queries generated for the query parts shown in Figure 5.7.

Algorithm 5.9: Function `processValuesStatements` from Algorithm 5.6.

```
1 function processValuesStatements(queryPart, W):
      Input: queryPart – the query part being translated
             W – database wrapper for this query part
2     foreach VALUES statement values in queryPart do
3         variable, constantList := values
4         W.addValueSet(getVarId(variable, constantList))
```

we have a set of possible logical operators to consider, but operands of a `FILTER` statement can be either *variables*, *constants*, or *aggregations* which we didn't have to consider earlier. In the cases of aggregations, we must first check if the required aggregation is supported by the underlying database, and defer this filter in the case that it is not. If the aggregation is supported, an aggregation root is selected as the lowest possible aggregation root in the access path, meaning if there are multiple nested levels of arrays, the most nested one is always selected (as per the semantics of MMQL). It is worth pointing out that in cases where the aggregation root is outside the query part containing the aggregated variable, this aggregation must necessarily be deferred. Each aggregation call on the database wrapper creates a new variable with the results of the aggregation, and this variable is then used in the filter call on the database wrapper.

The function `processFilterStatements` can be seen in Algorithm 5.10. Note that symmetric cases where the order of operands is swapped is not covered by the function, as these cases are solved by simply swapping the order of operands (for example filters with a variable and a constant are symmetric to filters with a constant and a variable). In addition the shown function omits the checks

which verify that the appropriate aggregation type is available via this database wrapper. In the case that the aggregation type is unavailable, this filter is deferred. This function also uses the helper function `createAggregation`, whose purpose is to create the appropriate aggregation and return the variable identifier for its result. Its implementation is shown in Algorithm 5.11. We will point out that a function `findAggregationRoot` is called in this function, which given the assumption that the aggregation root for this aggregation is located within the same query part, returns the kind containing the aggregation root, as well as the path to the root within the kind. The last function worth mentioning is `getPathsForProps`, which simply returns a property path for each of the provided morphism signatures, effectively just locating them within their source kind.

Algorithm 5.10: Function `processFilterStatements` from Algorithm 5.6.

```
1  function processFilterStatements(queryPart, W):
       Input: queryPart – the query part being translated
              W – database wrapper for this query part
2      foreach FILTER statement filter in queryPart do
3          lhs, op, rhs := filter
4          if lhs is variable and rhs is constant then
5              W.addFilter(getVarId(lhs), op, rhs)
6          else if lhs is variable and rhs is variable then
7              W.addFilter(getVarId(lhs), op, getVarId(rhs))
8          else if lhs is aggregation aggregationType over lhsVar then
9              lhsAggregationVarId := createAggregation(queryPart,
                 aggregationType, lhsVar)
10             if rhs is constant then
11                 W.addFilter(lhsAggregationVarId, op, rhs)
12             else if rhs is variable then
13                 W.addFilter(lhsAggregationVarId, op, getVarId(rhs)
14             else if rhs is aggregation aggregationTypeRhs over rhsVar then
15                 rhsAggregationVarId := createAggregation(queryPart,
                     aggregationTypeRhs, rhsVar)
16                 W.addFilter(lhsAggregationVarId, op,
                     rhsAggregationVarId)
```

Translating Set Operations

The reader has surely noticed that yet again, we omitted the handling of the `OPTIONAL`, `UNION` and `MINUS` clauses for simplicity, which is why we will mention how they would tie into the translation algorithm. In general, we can distinguish two main scenarios – either the entirety of a given clause is within a single query part, or there is only a part of it. In the case that any of these clauses cross query part boundaries, we duplicate the clause to each query part, preserving only the statements which correspond to the respective query part.

Algorithm 5.11: Function `createAggregation` from Algorithm 5.10.

```
1  function createAggregation(queryPart, aggregationType, aggregationVar,
     W):
       Input: queryPart – the query part being translated
              aggregationType – the type of aggregation to perform
              aggregationVar – the variable being aggregated
              W – database wrapper for this query part
2      aggregationRootPath, aggregationRootKind :=
         findAggregationRoot(queryPart, aggregationVar)
3      kind := getKind(aggregationVar)
4      propertyPath := getPropertyPath(aggregationVar, kind.mapping)
5      aggregationRootObj := getSchemaObject(aggregationRootPath)
6      foreach id in aggregationRootObj.ids do
7          if id can be selected from aggregationRootKind then
8              rootIdPaths := getPathsForProps(id,
                 aggregationRootKind.mapping)
9              variableId := generateNewVariableId()
10             W.addAggregation(aggregationType, aggregationRootKind,
                 rootIdPaths, getId(kind), propertyPath, variableId)
11             return variableId
```

In Section 5.1.1, we mentioned the existence of optional overloads of various database wrapper methods. The purpose of these optional overloads is the handling of the `OPTIONAL` clause, as we cannot use an inner join to create optional relationships between kinds, we must use an outer join variant instead. For this reason, all statements within an `OPTIONAL` clause are handled in the same fashion as their non-optional variants, but we call the optional overloads of wrapper methods instead of the non-optional ones.

In the case of `UNION`, we can simply treat both of its operands as optional, retrieving the relevant data if it is present. However, there is a need to filter out solutions for which neither side of the union expression contains a match. If the entire `UNION` clause is located within a single query part, this constraint may be expressed at the native query level in a relatively straightforward way, enforcing the presence of the relevant optional properties in the result. If this is not the case, then this constraint needs to be deferred to the categorical level.

As for `MINUS`, we need to handle the situation differently based on whether we can evaluate the entirety of the clause within a single query part. If so, then we join the second operand of the `MINUS` clause to its first using an optional join, subsequently adding a filter removing all solutions whose optional joins yielded a non-empty value. Otherwise, we again need to defer the elimination of invalid solutions to the categorical level.

Mapping Builder

Lastly, we will briefly mention the functionality of the mapping builder we are using in the translation algorithm shown in Algorithm 5.6. The job of the mapping builder is to construct a mapping (recall Section 2.5) which maps the result of

each query part to the categorical representation. We can imagine this as being equivalent to defining a database view returning exactly the results of our query, and creating a mapping treating this database view as a single kind.

The interface of the mapping builder is quite simple, providing two methods:

- `defineVariable(`*variableId*`,` *propertyPath*`,` *kind*`)` which associates the variable identified by *variableId* with the property represented by the path *propertyPath* in *kind*. The mapping builder simply stores this information until `buildMapping` is called.

- `buildMapping(`*variableMap*`)` which accepts a *variableMap*, mapping variable identifiers to property paths within the native query result. Using this mapping, the builder now has the original property path (including categorical information like the morphism signatures along the path) and the property path within the native query result. Given both of these, the mapping builder can construct a mapping which simply assigns the appropriate categorical identifiers to paths in the native result.

The purpose of the design of the mapping builder is the simple fact that we cannot make general assumption about the allowed variable names or naming conventions for any given database, meaning we cannot simply ask a database to name a given query output a certain way. For this reason, we leave the ultimate naming of constituent parts of the query part result up to the database wrapper, using variable identifiers instead. When defining a projection, we assign a variable identifier to this projection in the database wrapper. Upon building the native query, the database wrapper returns the final variable map, and this map is used to construct the categorical mapping for the native query result. Note that the path returned by the builder has the same structure (meaning length and types of constituent properties) as the property path used to define the variable, which is why the internals of the mapping builder are actually relatively straightforward.

5.2.6 Joining Data

With a query plan formed and selected, in the previous subsection we compiled each query part into a native database query intended to be executed with a specific database. We also mentioned the need to build a corresponding mapping for each query part, but we did not elaborate further, instead referring to this subsection, where we will describe in detail how data retrieved as the result of a native database query is transformed into the categorical model, and how their results are joined together. For reference, this part of the algorithm corresponds to phases IV and V from Figure 5.1.

The first part of this problem is the transformation of data in the context of a particular data model to an instance category conforming to a specific schema category. An algorithm for this model-to-category transformation was proposed by Pavel Koupil and Irena Holubová [6] (presented as Algorithm 1 in their paper). This algorithm, given a schema category and a mapping describing how a particular kind maps to the schema category (recall Chapter 2), transforms data in the shape of this mapping's access path to an instance category. We will not describe this algorithm in more detail, as only its inputs and outputs are relevant

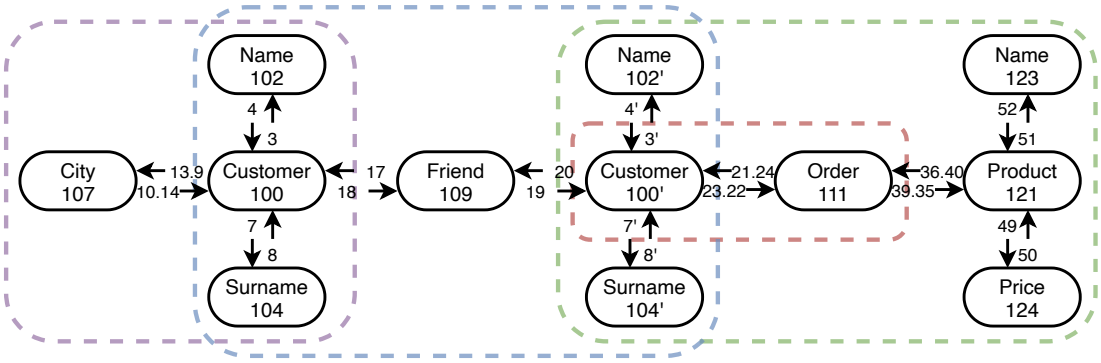
for our purposes, and we instead refer the reader to the original source for more details. The input mapping for this algorithm needs to be constructed in such a way that its structure describes exactly the structure of the data returned by the native database query, specifying how each property maps to a schema object. The most obvious idea would be to perhaps include this logic in the database wrappers themselves, as the wrappers are the ones building the native queries, and therefore will have the information necessary to also build this mapping. However, this would prove impractical, as the main purpose of database wrappers as described in Section 5.1.1 is to isolate the logic specific to each database to a minimal unit, making their implementation as straightforward as possible to allow easy extension for new database systems. Adding the mapping building logic to the database wrappers would therefore directly contradict this idea, as it would now be necessary for each implementer of a database wrapper to work with categorical concepts, which is not necessary as proposed in Section 5.1.1.

Since adding the mapping building logic to database wrappers themselves does not seem like a good idea, we instead introduced the concept of a *mapping builder*, which is shown in the algorithms presented in Section 5.2.8. This mapping builder constructs the mapping by being invoked together with the abstract database wrapper by the algorithms. When this mapping is constructed together with the native database query for a given query part, the aforementioned model-to-category algorithm is executed, resulting in an instance category containing the results of the native query.

As described in the original source of the model-to-category transformation algorithm, instances of each kind are inserted to the instance category one-by-one. This begs the question of what happens when we have multiple kinds (note that a kind in this context refers to the mapping created for each query part, not the original kinds in the queried schema category) which need to be joined together in the instance category. Luckily for us, this is achievable for instance categories using *pullbacks*, which are a generalization of the Cartesian square and intersection [37][38]. Using the example presented by Pavel Koupil and Irena Holubová [6], let us consider the following query: "For each customer who lives in Prague, find a friend who ordered the most expensive product among all customer's friends." We can see the decomposition of this query into query parts, as well as the generated native database query for each query part, in Figure 5.9.

In Figure 5.10, we can then see how the categorical results for each query part will be joined together with pullbacks. The first pullback joins the relational data with graph data ($P_1 = result_{REL} \bowtie_{100} result_{GRAPH}$), while the second one joins the document data with the result of the first pullback ($P_2 = P_1 \bowtie_{100'} result_{DOC}$). As described in Section 5.2.3, the join ordering problem is known to be hard, doubly so in multi-model scenarios, therefore it falls outside the scope of this thesis, and we leave it as part of future work on this topic.

For another example, we will remind the reader of the query in Figure 5.4. Recall that in Figure 5.8, we showed the generated native queries for this MMQL query. In Figure 5.11, we can see the results of these native queries as returned by the respective databases. With these results, after their transformation to an instance category, we can expect to see the instance morphism 4_1 to contain the following relations, showing relations between active domain rows of instance objects (each object active domain row is a set of tuples (signature, value)):




```
SELECT customerId
FROM Customer
WHERE city = "Prague";
```

```
MATCH
  (c:CUSTOMER)
    -[:KNOWS]->
      (f:CUSTOMER)
RETURN
  c.id, c.name, c.surname,
  f.name, f.surname, f.id;
```

```
db.orders.aggregate( [
  { $unwind : "$items" },
  { $sort : { "items.price" : -1 } },
  { $group : {
      _id : "$_id.customerId",
      items : { $push : { name : "$items.name", price : "$items.price" }}
      } },
  { $project: {
      _id : 1,
      name : { $arrayElemAt : [ "$items.name", 0 ] },
      price : { $arrayElemAt : [ "$items.price", 0 ] } }
  } ] );
```

Figure 5.9: Query decomposition into relational (purple), graph (blue) and document (green) query parts, showing the corresponding generated native database queries for each query part [6].

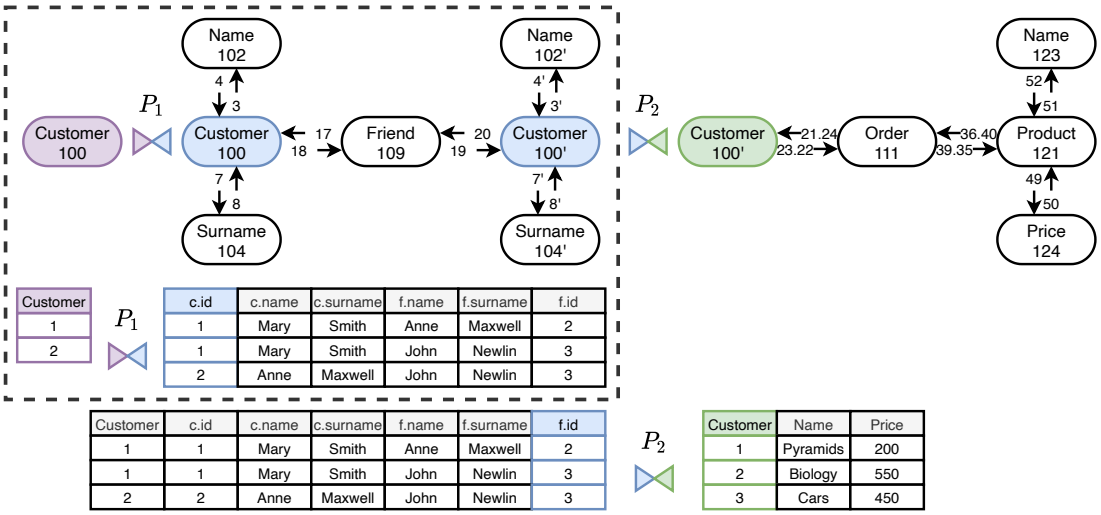


Figure 5.10: Joining of query parts from Figure 5.9 using pullbacks [6].

```
// PostgreSQL query
1,4,Alice
2,3,Bob
5,6,Charlie

// MongoDB query
{ customer_id: 1, price: 30 },
{ customer_id: 2, price: 25 },
{ customer_id: 3, price: 25 },
{ customer_id: 4, price: 30 },
{ customer_id: 5, price: 50 },
{ customer_id: 6, price: 10 }
```

Figure 5.11: Native query results for the queries shown in Figure 5.8.

- `({(6,30)}, {(1,1),(2,Alice)})`
- `({(6,25)}, {(1,2),(2,Bob)})`

As we can see in the first row, there is an order with a total price of 30 which was placed by a customer with an ID of 1 named Alice. Let us also look at the instance morphism 4_2:

- `({(6,30)}, {(1,4),(2,Alice)})`
- `({(6,25)}, {(1,3),(2,Bob)})`

We can see that a different customer named Alice with an ID of 4 has also placed an order with a total price of 30, and similarly, two different customers named Bob with IDs 2 and 3 have both placed an order with a total price of 25. There were also two customers named Charlie, but their orders do not have the same price, which is why they are excluded.

This approach for merging instance categories, coupled with the model-to-category transformation, leaves us with a single instance category representing the result of the MMQL `WHERE` clause.

5.2.7 Projection

With the instance category representing the result of the `WHERE` clause in hand, meaning we have already processed the selection part of the query, we also need to process the projection part, meaning the `SELECT` clause. However, before we do that, there is one more thing which needs to happen – *execution of deferred statements*. To get an overview of how this fits into the overall approach, we refer the reader to Figure 5.1, whose Phase VI corresponds to the contents of this subsection.

Deferred Statements

As mentioned in Section 5.2.8, there are certain MMQL statements which cross database boundaries, and therefore cannot be executed at the database level. For example, we could have a `FILTER` statement demanding the equality of two variables whose values come from different databases. Such statements must necessarily be processed by the MMQL query engine, and in the constraints of our approach where each query part is an executable unit independent of other query parts, this is unavoidable. There is a possibility that for approaches which introduce dependencies between queries, a certain part of this may be optimized, for instance possibly adding a constraint to a query based on the results of another query. However, as this is purely an optimization, we opted not to include this in our approach.

When it comes to the set of features which may possibly be a deferred statement, we have a few options, except for which all statements are *not* deferred:

- *Triples* with repeated compound morphisms may be deferred statements in cases where the compound morphism being repeated crosses query part boundaries;
- `FILTER` statements may be deferred in cases where both operands of the included logic expression are variables or aggregations of variables, each of which is queried from a different database, or in cases where a required aggregation is not supported by the corresponding database; and
- `FILTER` and `VALUES` statements in query parts whose database wrappers define the `isNonIdFilterSupported` property to be false.

The processing of deferred statements essentially means their emulation by the query engine using the instance category corresponding to the query's `WHERE` clause. We will not provide an explicit implementation of this algorithm, as its implementation would be lengthy and its inclusion in this thesis redundant, and all of the necessary operations are straightforward emulation of the statements. Lastly, we will point out that the set of deferred (or deferrable) statements will vary depending on the design of the supporting algorithms for MMQL, as MMQL itself does not necessitate the deferral of any statements, opening the door to future optimizations.

Projection Algorithm

After the processing of deferred statements, we are ready to approach the main topic of this subsection, which is projection to the desired representation. In this step, we take the instance category corresponding to the `WHERE` clause of the query, and transform it into an instance category corresponding to the `SELECT` clause, which induces its own schema category. It is worth mentioning the construction of this schema category, as we will need the morphisms contained within to have correct cardinalities. This can be achieved by iterating over the morphisms defined in the `SELECT` clause, and for each of them, finding all paths in the schema category induced by the `WHERE` clause between the subject and object of the triple where the morphism is defined, disregarding paths containing cycles. We do not

consider paths with cycles, as a cycle starting and ending with the same variable necessarily cannot have any effect on the cardinality of the final morphism, as it forms an identity function. Note that there can indeed be multiple paths, for example if we consider two customers who each have a home address listed, we may have a query expressing the fact that both customers who ordered a specific item live at the same address, giving us two paths between the item ordered and the customer address. For each morphism, we must examine all paths, selecting the minimal and maximal cardinality of the entire path, and then select the minimal value of the sets of minimal and maximal cardinalities to get the final minimal and maximal cardinality of the morphism. This is because if multiple paths exist between the same two variables, a particular result will only be included in the instance category only if all of the paths match (unless some of those paths are optional and generated by the `OPTIONAL` clause, however we still need to consider them for the minimal and maximal calculation). The algorithm generating the schema category induced by the `SELECT` clause can be seen in Algorithm 5.12. We will point out the function `getPathsInSchema` used in line 10 of the aforementioned algorithm, which finds all paths in the schema category between two schema objects, provided to the function using the `src` and `dst` arguments for source and destination respectively.

Algorithm 5.12: Projection Schema Algorithm.

Input: $\mathbf{S}_{WHERE}$ – schema category induced by the WHERE clause
$SELECT$ – SELECT clause of the MMQL query

```
1  S_SELECT := ∅
2  foreach triple subject morphism object in SELECT do
3      foreach var in [subject, object] do
4          oldObj := S_WHERE.getObjectFromVar(var)
5          newObj := S_SELECT.getObjectByKey(oldObj.key)
6          if newObj is NULL then
7              S_SELECT.objects.add(oldObj)
8      newMorphism := S_SELECT.getMorphism(morphism)
9      if newMorphism is NULL then
10         paths := getPathsInSchema(S_WHERE, src=subject, dst=object)
11         foreach path in paths do
12             if path contains cycles then
13                 paths.remove(path)
14         minCards := [ ]
15         maxCards := [ ]
16         foreach path in paths do
17             minCards.add(getMinCardinality(path))
18             maxCards.add(getMaxCardinality(path))
19         minCard := min(minCards)
20         maxCard := min(maxCards)
21         newMorphism := Morphism(morphism, subject, object, minCard,
             maxCard)
22         S_SELECT.morphisms.add(newMorphism)
```

With the schema category for the `SELECT` clause in hand, we can proceed to projecting the actual data, using the algorithm shown in Algorithm 5.13. This algorithm receives two inputs – the instance category for the `WHERE` clause constructed in previous steps, and the schema category induced by the `SELECT` clause. Its output is then data in the form of an instance category corresponding to the `SELECT` clause. This algorithm copies over data from instance objects for the `WHERE` clause to instance objects for the `SELECT` clause (since variable projection does not modify the set of values). It is worth explaining the notation `map(`*path*`,` x `=> ...)` used on line 14 of Algorithm 5.13, which means the application of the lambda function in the second argument to `map` on each element of *path*.

Algorithm 5.13: Projection Instance Algorithm.

Input: $\mathbf{I}_{WHERE}$ – instance category containing the results of the WHERE clause
$\mathbf{S}_{SELECT}$ – schema category induced by the SELECT clause

```
1  I_SELECT := ∅
2  foreach schemaObj in S_SELECT.objects do
3      instanceObj := I_WHERE.objects.getByKey(schemaObj.key)
4      I_SELECT.objects.add(instanceObj)
5  foreach schemaMorphism in S_SELECT.morphisms do
6      subject := schemaMorphism.domain
7      object := schemaMorphism.codomain
8      paths := getPathsInSchema(S_WHERE, src=subject, dst=object)
9      foreach path in paths do
10         if path contains cycles then
11             paths.remove(path)
12     instancePaths := [ ]
13     foreach path in paths do
14         instancePath := map(path, x => getInstanceMorphism(x, I_WHERE))
15         instancePaths.add(instancePath)
16     contractedPaths := map(instancePaths, contractMorphisms)
17     instanceMorphism :=
         baseInstanceMorphismIntersection(contractedPaths)
18     instanceMorphism.schemaMorphism := schemaMorphism
19     I_SELECT.morphisms.add(instanceMorphism)
```

When it comes to creating the set of instance morphisms, it gets more complicated. For morphisms from the `SELECT` clause which correspond to base morphisms from the `WHERE` clause, we can simply copy the instance morphism over to the new instance category, with a new signature corresponding to the signature specified in the `SELECT` clause. However, it is possible for a morphism from the `SELECT` clause to correspond to a path (or set of paths) with length greater than one in the categorical representation of the `WHERE` clause. In this case, we need to introduce the notion of *morphism contractions*. Given a compound morphism mapping a subject to an object, its contraction is a base morphism with the same mapping. In other words, we merge the base morphisms which form the compound morphism into a single base morphism in the new instance category. To do this, we must again get the set of paths (disregarding cycles) in the instance

Algorithm 5.14: Function `contractMorphisms` from Algorithm 5.13.

Input: *instancePath* – list of base instance morphisms forming a path in the instance category induced by the WHERE clause

```
1  contractedPath := copy(instancePath)
2  while len(contractedPath) > 1 do
       // If our path ends with morphisms (X) -a- (Y) -b- (Z),
          contract these morphisms to create a morphism (X) -a+b- (Z)
3      b := contractedPath.popBack()
4      a := contractedPath.popBack()
5      joinedMappings := join(a.mappings, b.mappings, x_a => x_a.codomain, x_b
         => x_b.domain)
6      contractedMappings := [ ]
7      foreach joinedMapping in joinedMappings do
8          contractedMapping := joinedMapping.a.domain,
             joinedMapping.b.codomain
9          contractedMappings.add(jcontractedMapping)
10     contractedMorphism := InstanceMorphism(contractedMappings)
11     contractedPath.add(contractedMorphism)
12 return contractedPath.first()
```

category induced by the `WHERE` clause, and perform morphism contraction on each path. Finally, to produce the new instance morphism for the result instance category, we perform an intersection of all base morphisms created as a result of the path contractions, meaning the final domain-codomain map only contains a mapping of domain to codomain if all paths contain this mapping.

Additional Considerations

The perceptive reader may have noticed that the algorithm which we presented in Algorithm 5.13 does not take into account the `OPTIONAL`, `UNION` and `MINUS` clauses of MMQL. Indeed, the semantics of paths contained within these clauses are different in the context of creating result instance morphisms, however we intentionally omitted them from Algorithm 5.13 to preserve its simplicity and readability.

When it comes to paths within (or partially within) the `OPTIONAL` clause, we can disregard them if there exists at least one non-optional path, as the non-optional paths constrain the instance morphism to a set of mappings, and regardless of the existence of a particular optional mapping, the corresponding non-optional mapping always exists. If no non-optional paths exist, then we define the instance morphism to be the union of all optional paths, after morphism contraction has been applied to them. This is because for any given domain-codomain mapping to exist in the result instance morphism, it is enough for any given optional path to exist in the instance category corresponding to the `WHERE` clause.

Earlier, we specified that both arguments of the `UNION` clause are treated as optional. If the entirety of a given `UNION` clause could be resolved within a single query part, we do not need to perform any extra work, as not only do we have the correct data in the instance morphisms, but the set of instance objects is also

correctly filtered. In the case that the `UNION` clause spans multiple query parts, we need to additionally filter the set of instance objects to eliminate those for whom neither `UNION` argument was matched.

As for the `MINUS` clause, the final instance morphism is the result of taking the instance morphism from the first argument of `MINUS`, while for any given domain-codomain mapping, if this mapping is also contained in the second argument of `MINUS`, it is eliminated. Note that if the `MINUS` clause is contained entirely within a single query part, the required results have already been filtered out, meaning the filtering at this level is trivial, removing no mappings.

The last simplification made in the algorithm is the omission of constants and aggregations present in the `SELECT` clause. While aggregations present here could be computed at the database level to increase overall efficiency, this is an optimization which would complicate the algorithms further, which is why we opted to simply explain it in this fashion. Aggregations and constants in the `SELECT` clause would have schema objects defined just like variables, with corresponding instance objects having a constant value in the case of constants. In the case of aggregations, their values would be computed according to their semantics as described in Section 4.4.8, with the instance morphisms being defined accordingly.

We should also mention that the algorithms presented in this subsection are critical to query performance, as morphism contraction is an extremely expensive operation for a large dataset, entailing an inner join across all data in the instance morphism for each contraction. The true cost of the contraction operation will be further explored in Chapter 8.

The final piece of the projection algorithm is taking care of the `ORDER BY`, `LIMIT` and `OFFSET` clauses of the query. Again, while these could be handled at the database level natively, this is purely an optimization, and it would complicate the structure of our algorithms, which is why we opted to address them here. As far as ordering is concerned, the final instance category would be divided up into maximal connected components, and these connected components ordered by the ordering criteria. A special instance object would then be inserted into the instance category, with a morphism mapping sequence numbers from 1 to N to the connected components. `LIMIT` and `OFFSET` are quite simple in their function, meaning we would simply restrict the result set as required.

To give a concrete example, recall the query from Figure 5.4, and the results of the generated native queries from Figure 5.11. Given these results, the final instance category which represents the result of the MMQL query will have the following active domain rows for the `_:shared` object:

- {(name,Alice),(price,30)}
- {(name,Bob),(price,25)}

Again, note that each active domain row is a set of tuples (signature, value). Normally when working with a schema category, base morphism signatures are integers which are automatically assigned, but technically a base morphism signature can be any string, which is why we allow their naming in the MMQL `SELECT` clause.

5.2.8 Transforming Data

As the result of the algorithm as proposed up to this point, we have an instance category corresponding to the schema category induced by the MMQL query's `SELECT` clause. However, query results in the form of an instance category may not be practical for most use cases. For this reason, we recognize the need for the query results to be transformed into a more suitable format, like JSON or RDF. As MMQL is a categorical query language with categorical inputs and outputs, we believe that the transformation of data is best left out of the language itself, instead relying on supplemental query tooling to perform this job. Despite this, we will describe our proposed approach for such a transformation, as we believe it is necessary to showcase the end-to-end validity and usability of the proposals made in this thesis. This final part of the entire workflow corresponds to Phase VII as shown in Figure 5.1.

In general, we can formalize this problem as the transformation of an instance category to a specific data model. As we mentioned in Section 5.2.6, Pavel Koupil and Irena Holubová proposed an algorithm for model-to-category transformation [6], which we are using to transform data retrieved by native database queries into a categorical representation, and to subsequently join the data from multiple databases into a single instance category representing the `WHERE` clause of the MMQL query. However, in the same paper, they also proposed an algorithm for a transformation in the opposite direction, meaning category-to-model (Algorithms 4 and 5 in their paper). As it turns out, this algorithm solves exactly the problem we need to solve in order to transform the result instance category to a format like JSON or RDF. For this reason, we propose that this algorithm be used for this transformation, and we refer the reader to the original source for more details on this algorithm.

It is worth mentioning that this transformation requires an input in the form of a mapping (recall Section 2.5), which specifies the shape of the resulting data. This mapping can be inferred from the shape of the MMQL `SELECT` clause in a straightforward way for some data models, for example in the graph model, we are simply transforming graph data to graph data in another representation, meaning relatively few modifications are necessary. However, for some data models like the document model, this gets somewhat complicated. The document model operates with tree structures, which in general do not permit lower levels of the tree to refer to upper levels. This can pose a problem in the instance that the schema category induced by the `SELECT` clause contains back edges, forward edges or cross edges (borrowing terminology from the well-known Depth First Search algorithm), which was not an issue with graph data.

There are multiple possible solutions to the problem of transforming general directed graphs to trees [39], but we will mention two main approaches which are best suited to our use case. The first approach is to automatically determine the schema object which will become the root of the tree, and construct a tree rooted in this schema object, replacing back, forward or cross references in the tree with object identifiers where possible, and with inlined data where not possible (for example with objects which have no identifier, such as JSON arrays). The automatic determination of the tree root is simple in the case where the graph already forms a tree, but in the more general case, we would need to pick such a tree root based on an algorithm, for example one which would pick a tree root such

that the number of edges in the tree preserved from the original graph is maximal. The second approach would be to prompt the user to manually select the tree root when requesting a transformation which requires a root to be selected and the root cannot be determined automatically, for example when transforming to JSON when the schema category induced by the `SELECT` clause is not already a tree. However, neither of the proposals is perfect as a transformation from a general graph to a tree necessarily carries some tradeoffs, which is why we believe that users of MMQL should try to formulate their queries to output categorical data in the shape of trees where possible.

6. MM-quecat

In the previous two chapters, we introduced MMQL, as well as the supporting algorithms enabling its implementation. However, we presented both on a theoretical level only. To properly evaluate the validity and correctness of our proposal, it must be verified in practice by way of implementation. Therefore this chapter presents *MM-quecat*[1], a proof-of-concept implementation of the core concepts of MMQL. The purpose of this implementation is the verification of our proposed approach. Full implementation of all MMQL concepts for all scenarios would be beyond the scope of this thesis, which is why the implementation focuses only on the key parts of MMQL. This chapter not only describes the implementation of MM-quecat itself, but also the technical challenges faced along the way.

6.1 Solution Architecture

MM-quecat is a Python 3[2] library which provides the functionality of executing MMQL queries. Its interface is rather simple on the surface - it provides a function `execute_query`, which takes the input query as a string, and returns an instance category (see Section 2.4) representing the result of the query.

To do this, it needs to communicate with MM-evocat [22], which is a multi-model data modeling and evolution framework based on category theory, written as a Java server application. MM-evocat has the functionality of creating a schema category (see Section 2.3). For that reason, MM-quecat's main interface function `execute_query` also contains a parameter containing the ID of the schema category, which refers to a schema category within MM-evocat. In MM-evocat, one can not only model the schema category, but also create the mappings necessary to transform data from its native representation into the categorical representation, using an algorithm for model-to-category transformation [6]. While including MM-evocat in the solution does introduce overhead in the form of network communication, the benefits it brings outweigh the negatives for our purposes, since it includes the categorical modeling tools which are necessary for MM-quecat to do its job. Without MM-evocat, MM-quecat would need to implement all of the necessary functionality itself, effectively duplicating the functionality of MM-evocat to avoid network overhead, which is not desirable at this point of the tool's lifecycle as proof-of-concept software. MM-evocat is also still in active development by its author and is not yet fully finished, which further complicates its usage, as we will mention again going forward in this chapter. Perhaps in the future, if the a full and optimized implementation of MM-quecat is desired and when MM-evocat's development is finished, both tools could be merged into a single one.

The architecture of the whole solution can be seen in Figure 6.1, which shows MM-quecat communicating with MM-evocat via HTTP. We can see that MM-quecat sends requests to an instance of MM-evocat which contains the necessary schema category and mappings representing the data being queried, as well as

[1] `https://github.com/yawnston/querycat`

[2] `https://www.python.org/`

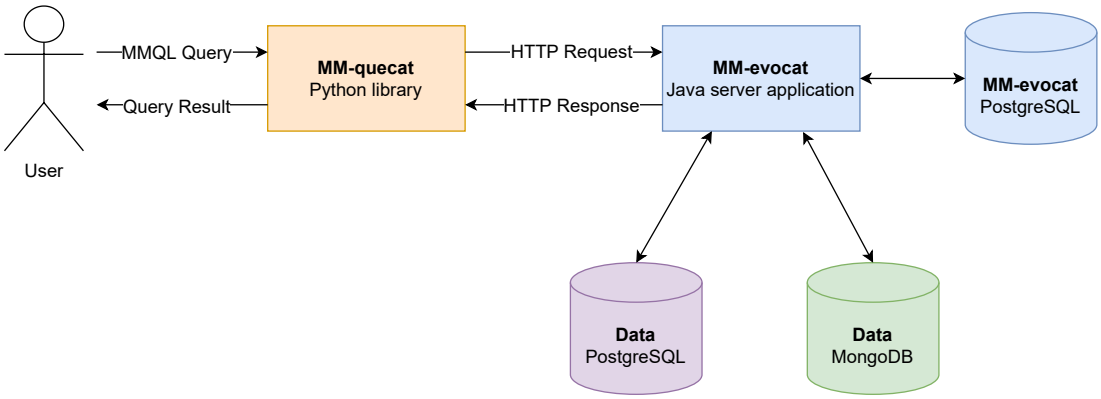


Figure 6.1: Architecture of the MM-quecat solution.

maintaining the connections to the relevant databases. Note that at no point does MM-quecat actually need to communicate with the queried databases directly, as the queries generated by MM-quecat are passed to MM-evocat, which executes the queries and transforms their results to the categorical representation using a mapping provided by MM-quecat[3]. This increases the modularity of the solution, as MM-quecat only needs to rely on the categorical interface of MM-evocat, and the database wrappers (recall Section 5.1.1 where we introduced the notion of database wrappers) have their implementation greatly simplified, as they only need to generate the native queries without having to worry about executing them, or transforming their results to a categorical representation[4]. Lastly, Figure 6.1 shows that MM-evocat also has its own PostgreSQL database which it uses to store schema and mapping data, as well as instance categories. This database is separate from the databases containing the actual data being queried. Going forward, MM-quecat will also be wrapped in a simple HTTP API as required for the purposes of the querying tool which we will introduce in Chapter 7, as this querying tool will continue being worked on as part of a conference demo paper which the author of this thesis is co-authoring [32].

For completeness, we should also mention the choice of Python as the language for the implementation of MM-quecat. The ecosystem of tools developed for multi-model data representation at the Faculty of Mathematics and Physics of Charles University, which MM-quecat and MM-evocat are a part of, already contains projects in both Java and Python, making both languages natural candidates for the cohesion of the tool ecosystem. From these two languages, Python was chosen due to the personal preferences of the author of this thesis, as they have extensive experience with Python, but little to no Java experience. To add to this, it is the author's personal belief that Python, with its simplicity, ease of use and universal usefulness, is the language of the future, and will continue to

[3]While direct database communication is not necessary in MM-quecat's current form, the implementation of a fully-featured multi-model query planner would necessitate this direct connection, as the query planner would need access to the databases' own query planners, usually in the form of a database explain command, to assess the cost of MMQL multi-model query plans.

[4]A potential problem may arise if the database wrappers need to access the database interface to query it for the supported query language version in cases where the database wrapper is deciding whether it may use a newer language feature. However, this would be solved by simply adding the database connection to MM-quecat just like in the case of the query planner.

gain market share while Java slowly fades away to make room for more modern languages.

Since MMQL is a multi-model query language and MM-quecat is a library implementing MMQL, we also need to discuss MM-quecat's multi-model capabilities. As implemented, MM-quecat supports two models - relational in the form of PostgreSQL, and document in the form of MongoDB. The reason for this is relatively straightforward, as both databases are among the most popular in their class, and together they represent a significant market share of the database space. In addition, MM-evocat currently only supports these two databases, which means that MM-quecat cannot support additional databases without their corresponding implementation in MM-evocat. Among other limitations, MM-evocat only supports string values, meaning the data type of everything in MM-evocat is a string. For this reason, MM-quecat is also constrained to only supporting strings, even though MMQL also supports other data types.

With the solution architecture out of the way, we will describe specific parts of MM-quecat and their implementation, loosely following the solution structure proposed in Section 5.1.

6.2 Parsing

The first step in implementing any language, be it a query or programming language, is to implement a parser for this language. This is because it is not feasible to work with a language based on its raw text alone, but rather it is more practical to operate on language-level constructs which are parsed from the raw text by a parser. While the parser for MMQL could be manually implemented, a smarter solution exists in the form of parser generators. These tools generally accept some sort of formal grammar as input, and in return generate a parser for the language represented by the grammar. As it happens, we had been planning to formally specify the grammar of MMQL anyway, which made using the grammar to generate a parser a natural step. The grammar used to generate the parser for MM-quecat is included in Attachment A.1.

This was accomplished using the ANTLR[5] library, which is a parser generator for reading, processing, executing, or translating structured text or binary files. ANTLR was used to generate the parser classes in Python, at which point we implemented an abstract syntax tree (AST) visitor, which walks the AST created by the generated parser, and converts it into a suitable form for further processing in the query execution process. This form consists of translating the input graph patterns into sets of triples, coupled with additional constructs such as `FILTER` expressions or other query features.

6.3 Creating Query Plans

As described in Chapter 5, we need to divide the query up into query parts to form query plans. First, queries are preprocessed to transform triples containing compound morphisms into multiple triples, each containing a base morphism. For each point where a compound morphism was split, a temporary internal variable

[5] `https://www.antlr.org/`

is inserted into the query. As discussed in Section 5.2.1, this greatly simplifies query processing as a whole, but introduces issues when it comes to performance and recursive paths. The performance consideration is not an issue for MM-quecat, as achieving high performance is not within the scope of its proof-of-concept nature (and within the scope of this thesis in general). The consideration of recursive morphisms, while unpleasant, is also not key to demonstrating the viability of MMQL and MM-quecat as a whole. The implementation of recursive morphisms would also be rather complex, which is why they are not supported in the implementation, giving us space to focus on the main focus points of this thesis.

Aside from the preprocessing, the implementation of query plan creation largely copies the algorithm outlined in Section 5.2.2, which is why we will not discuss it here further.

6.4 Query Translation

The query translation algorithm is a simplified, limited version of the algorithm presented in Section 5.2.5, supporting simple queries with both base and compound morphisms. However, the scope of the implementation of this algorithm is not limited only by the implementation effort required. As we mentioned earlier in this chapter, MM-evocat is still undergoing active development by its author, and as a consequence, some of its features are not quite finished yet. For example, there is an issue where if MM-quecat generates queries requiring database joins on multiple kinds (like multiple tables in the relational model or multiple collections in the document model), MM-evocat will not correctly process these queries when translating their results into the categorical representation, yielding an incorrect instance category. For this reason, MM-quecat cannot support queries with multiple kinds in a single query part at this time. While this is bothersome in terms of the completeness of the implementation, it is not an issue in the scope of a proof-of-concept implementation, as we can demonstrate the multi-model capabilities of MM-quecat on multi-model queries which do not require the use of multiple kinds from the same database. One such query is later shown in Figure 8.10 while evaluating the weaknesses of MM-quecat.

6.5 Selecting the Best Query Plan

In Section 5.2.4, we discussed the difficulties which come with multi-model query planning, and the lack of related work on the subject, with most sources only considering single-model query planning within the context of a single database. For that reason, in Section 5.2.4 we left the selection of the best query plan as an open point which is beyond the scope of this thesis, which already breaches the vast and complex area of multi-model querying with little related work to use as a base. Therefore in MM-quecat, the implementation of best plan selection is purposefully very basic - an arbitrary query plan is selected if multiple plans are present due to data redundancy. The implementation of a more sophisticated selection process first necessitates additional research on possible approaches to this problem.

6.6 Query Execution

As discussed earlier in this chapter, MM-evocat has the ability to execute native database queries in PostgreSQL and MongoDB, and then transform their results into the categorical representation. It also has the ability to merge the results of multiple such queries over the same schema category into a single instance category, using the objects' identifiers to determine the proper way to join the data. However, MM-evocat does not have the ability to execute arbitrary queries as required by MM-quecat, and its capability is limited to trivial projection-only queries that exactly copy the structure of the defined mappings. This is an issue, because MM-quecat generates various queries, which then need to be executed against the databases.

Because MM-evocat is in active development, waiting for MM-evocat to support this arbitrary query execution was not realistic within the context of this thesis. This is why MM-quecat is actually using a modified instance of MM-evocat[6], which was forked from the main implementation, in which the author of this thesis has made the modifications necessary for MM-quecat to fully function. The modifications include, but are not limited to:

- Modifying the API to allow execution of arbitrary database queries;
- Modifying the database wrappers for PostgreSQL and MongoDB to execute the provided queries, including the ability for the MongoDB wrapper to execute aggregation pipelines instead of simple find queries;
- Modifying the API to allow the retrieval of instance morphisms, in addition to the retrieval of instance objects, which was already implemented; and
- Addition of Docker[7] configuration files to easily run MM-evocat, since its setup is not very straightforward and includes multiple components.

For this reason, MM-quecat only functions with the modified version of MM-evocat. However, in the future, when MM-evocat is modified by its author to fully support the features required by MM-quecat, these temporary modifications will become obsolete, and MM-quecat will use the main, up-to-date MM-evocat.

Aside from the necessity for these modifications, the query execution algorithm operates mostly as described in Chapter 5. MM-quecat uses the API provided by MM-evocat to create a copy of the queried schema category, and to define the mappings which are needed by the query parts in the query. These mappings define the shape of the results of each individual native database query, effectively telling MM-evocat how to transform the result of this query into the categorical representation. The queries generated by MM-quecat are executed within MM-evocat, including joining of their results into a single instance category. This instance category is then retrieved from MM-evocat using the modified API, and the remaining steps are executed as described in Chapter 5, projecting the instance category to the final instance category, along with creating the schema category defining this result instance category. The final instance category is then returned as the result of the query.

[6] `https://github.com/yawnston/evolution-management`
[7] `https://www.docker.com/`

Note that in our description of the proposed approach in Section 5.1, we also mentioned the need to transform the query result into a more usable representation, such as JSON. We discussed this in greater detail in Section 5.2.8, where we mentioned the fact that we can use an existing algorithm for category-to-model transformation [6].

This is another limitation where MM-evocat is not quite ready, as this functionality is not yet fully implemented as required by MM-quecat[8]. However, when the required functionality is complete on the side of MM-evocat, MM-quecat will take the schema category corresponding to the query result, define the required mappings to transform the result into a JSON representation, and send both over to MM-evocat to perform the actual transformation.

As a whole, the implementation of MM-quecat in this form is sufficient to verify the validity of our proposed approach, as we later demonstrate in Chapter 8 with concrete queries processed by MM-quecat. However, as can be seen from the solution architecture, there is certainly room for performance optimization, since there is quite a bit of network communication overhead between MM-quecat and MM-evocat, along with the proposed algorithms being optimized for readability and simplicity, not performance. These considerations will also be discussed in Chapter 8.

[8]In particular, the algorithm for category-to-model transformation itself is implemented, but not the relevant functionality for the transformation to a format like JSON or RDF.

7. Querying Tools

In the previous chapter, we discussed the proof-of-concept implementation of MM-quecat, which exists in the form of a Python library. However, despite its interface being easy to use, it does not fully leverage the power of MMQL with respect to the end user experience. Recall that in Section 2.6, we articulated the requirements that MMQL *be expressive and readable*, and that it *be intuitive and familiar to users of existing query languages*. While we believe that MMQL meets these design goals as proposed, and that one of its greatest strengths lies in being graphically expressive, meaning users can define graph patterns which visually resemble the actual structures being defined. Therefore, in this chapter, we present the prototype of a user interface (UI) application for MM-quecat[1], which allows the user to visually construct queries.

The presented prototype is not one of the main products of this thesis, but we present it regardless, as it should give the reader an idea of how the language may be used from an end-user perspective. The prototype of the UI for MM-quecat was created by the author of this thesis while working on a not-yet-published demo paper [32], co-authored with Pavel Koupil and Irena Holubová. Therefore its implementation will be completed as the paper nears its publishing date, and in this chapter, we present it as it exists in its current form.

The querying tool is a web application built with React[2] and Next.js[3], utilizing the Cytoscape.js[4] library for graph visualization and MUI[5] as a component library.

7.1 Requirements

Before we introduce our design, we first need to formalize the requirements for such an application. The application should primarily serve as a querying tool, meaning it should be possible for the user to construct a query using the schema category as a visual aid. The user should have the ability to directly execute the query, and retrieve the results in the chosen representation (like JSON for example).

Aside from this primary use case, the user should also have the ability to view all of the possible query plans created by MM-quecat for the execution of the query. Each query plan should be able to be examined further, displaying the native database queries generated by MM-quecat for this specific query plan, as well as visualizing the query plan in a subset of the full schema category.

As a whole, the query tool should heavily leverage the graphical nature of MMQL, nicely visualizing the query in the schema category and possibly even providing semantic syntax highlighting in the query itself, with colors corresponding to the concepts visualized in the schema category.

[1] `https://github.com/yawnston/quecat-frontend`
[2] `https://reactjs.org/`
[3] `https://nextjs.org/`
[4] `https://js.cytoscape.org/`
[5] `https://mui.com/`

7.2 User Interface

Now that we have formalized the requirements for a MMQL query tool, we will showcase our prototype. The main query screen can be seen in Figure 7.1, where on the right side, the user may construct an MMQL query, and on the right side, a visual aid in the form of the schema category is present. To help the user understand the query that they are writing, the schema category display also contains a visualization of various query features.

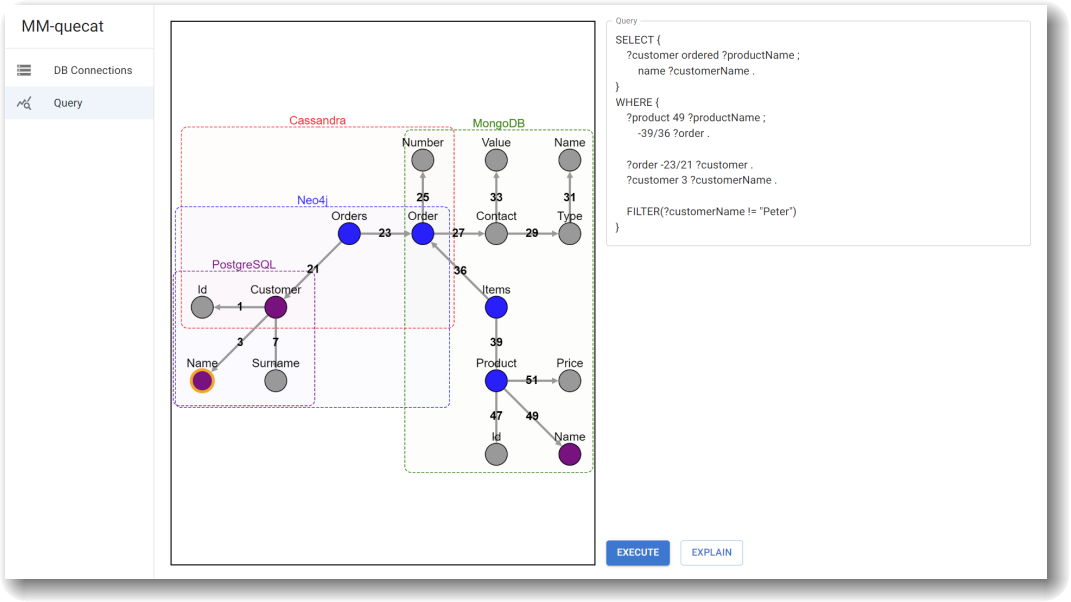


Figure 7.1: MM-quecat querying UI, showing the query on the right, and the schema category with the visualized query on the left.

For example, the `Order` schema object is colored blue, as it lies on the query path, but is not a part of the projection. The `Contact` object is colored gray since it is not part of the query at all, and the `Name` object corresponding to a `Product` is colored purple, since it is part of the query projection. Lastly, we can see that the `Name` object corresponding to a `Customer` is colored purple with an orange outline, meaning that it is part of the projection, and there exists a filter on this schema object.

If the application user clicks on the "EXPLAIN" button in the bottom part of the screen, they will be taken to the screen shown in Figure 7.2. There they can see a list of all possible query plans, coupled with their plan cost and the databases used in the plan.

The user may further examine individual query plans, leading to the screen shown in Figure 7.3. This screen, similarly to the screen shown in Figure 7.1, shows the schema category on the left side, however in this instance, the schema category is restricted to the databases and schema objects which are part of this specific query plan. On the right side, the user can see the native database queries which were generated for this plan by MM-quecat. The user can use this information to verify that the queries generated by MM-quecat are, for instance, using the correct indexes in the corresponding databases.

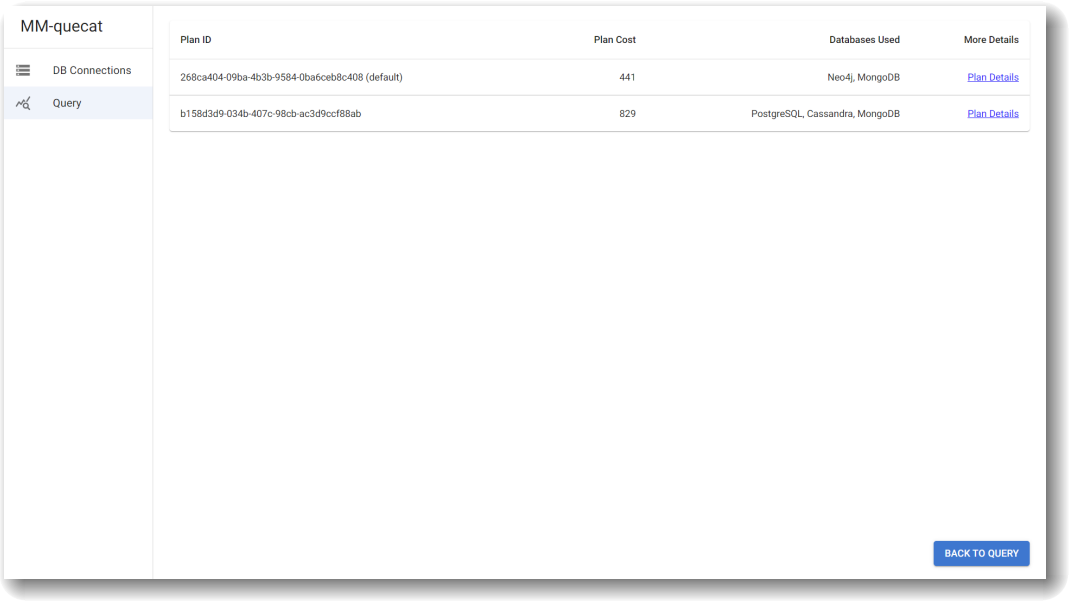


Figure 7.2: A table displaying all query plans generated by MM-quecat and their details.

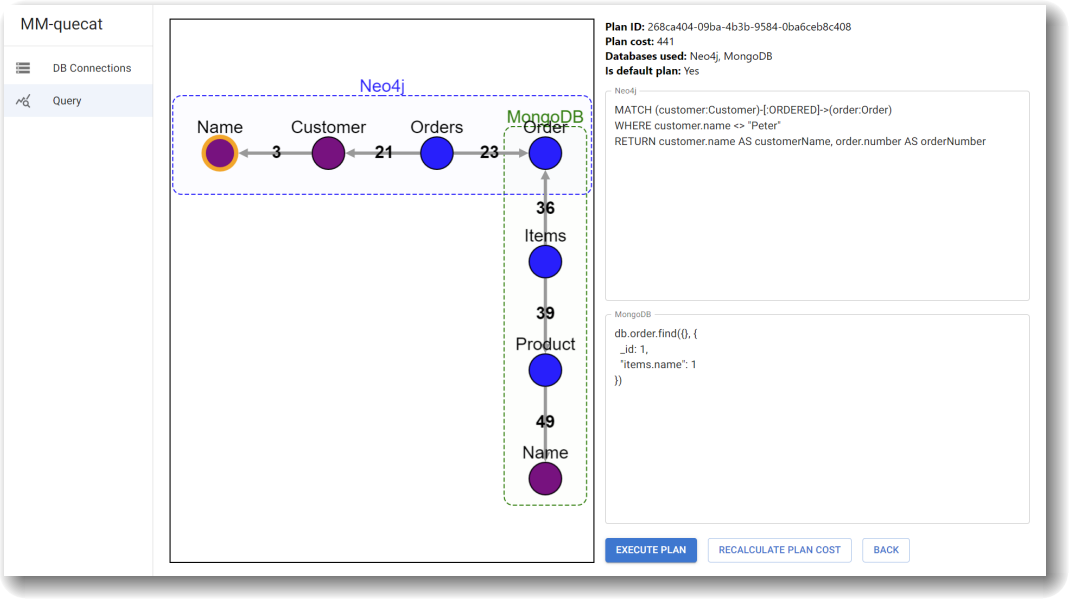


Figure 7.3: Detailed view of a specific query plan, showing the native database queries generated by MM-quecat.

8. Evaluation

Now that we have discussed the goals and scope of the proof-of-concept implementation of MM-quecat, recall that in Chapter 6, we mentioned query performance as one of the main open points of our approach. In Section 2.6, we mentioned that a suitable multi-model query language should *"have the capability of being nearly as performant as native queries where possible"*. To that end, we discussed optimization options in Chapter 5 while discussing the proposed algorithms. However, in order to properly understand the performance of the proposed approach, and to be able to constructively suggest effective optimizations for the future, it would be useful to experimentally evaluate the performance of MM-quecat. This chapter is therefore dedicated to such an evaluation.

Specifically, we have four main goals in mind for the experiments in this chapter, which is to:

1. Understand the overhead introduced by our approach compared to native database queries in a single-model scenario;
2. Understand the overhead in the context of merging data in a multi-model, multi-database scenario;
3. Locate the main performance bottlenecks in MM-quecat; and
4. Establish a performance baseline for future optimizations.

The motivations for the first goal should be quite clear, as our approach introduces overhead in the form of converting data into a categorical representation and network communication overhead. Therefore to properly fulfill the first goal of this chapter, we will need to prepare simple, single-model scenarios which we can use to isolate the amount of overhead introduced in this fashion.

As for the second goal, recall that in Chapter 5, we mentioned the need to merge together data retrieved from different query parts. While the merging itself is not performed directly by MM-quecat, but rather by MM-evocat [22], it is still crucial to understand the impact of this step on the whole approach. To this end, we will naturally need to include a multi-model scenario in our evaluation.

Lastly, a simple observation motivates the third goal - in practice, it is best to avoid premature optimization, and rather use benchmarking to identify specific performance bottlenecks. Because the problem domain of multi-model querying is very complex, naturally the approaches we designed are as well. This means that avoiding premature optimization is even more important, as we want to preserve the simplicity and comprehensibility of our approach where possible. To this end, during the writing of this chapter, two key tools were utilized to aid us in locating performance bottlenecks: Python's cProfile[1] module which allows us to collect performance statistics about Python programs, and SnakeViz[2], which is a tool for the visualization of cProfile's output.

Before we begin with the evaluation itself, it is also worth mentioning that the use cases designed for evaluation purposes also serve as use cases for the

[1] `https://docs.python.org/3/library/profile.html`
[2] `https://jiffyclub.github.io/snakeviz/`

verification of the implementation of MM-quecat. Specifically, the multi-model use case presented in Section 8.3 also presents the main use case of MM-quecat (and this thesis in general), which is the unified querying of multi-model data. By including this use case in the evaluation, we also verify that MM-quecat fulfills its intended primary purpose.

8.1 Evaluation Framework

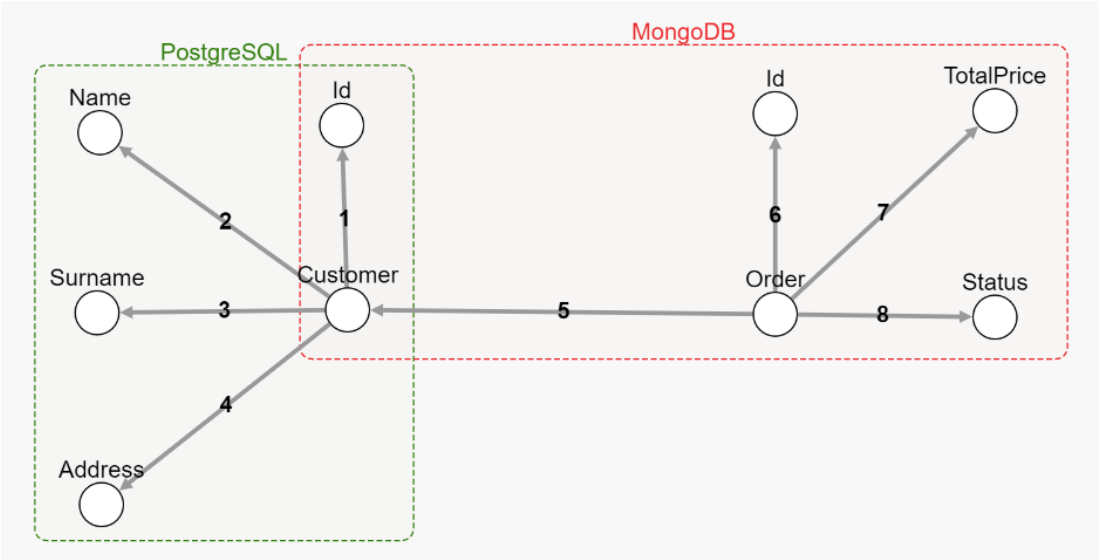


Figure 8.1: Schema category used for all evaluations.

For the purposes of the aforementioned evaluations, we propose a model scenario used for both single-model and multi-model evaluations. The schema category for this scenario is shown in Figure 8.1. The corresponding mappings are then shown in Figure 8.2. The customers along with their properties are stored in a PostgreSQL table, and the orders are stored in a MongoDB collection. Using this schema, we are able to easily test the multi-model functionality of MM-quecat while keeping the example simple and understandable. Note that the `customer_id` stored for each order in MongoDB refers to a customer stored in PostgreSQL, allowing the joining of the data.

For each evaluation scenario, the given MMQL query is executed, timed, and its execution time is compared to the execution time of a native database query (or two native queries in the case of the multi-model scenario). Each scenario was repeated 100 times, and the presented results are the average of all repetitions.

The choice of databases for this example scenario is also not random - MM-evocat [22] supports only PostgreSQL and MongoDB, and therefore MM-quecat does as well. Therefore the only reasonable option is to use both the supported databases if we want to examine the multi-model behavior.

Every proper test needs a well-defined set of input data, and it is no different for our case. The data was generated using the Faker[3] library, using a random

[3] `https://faker.readthedocs.io/en/master/`

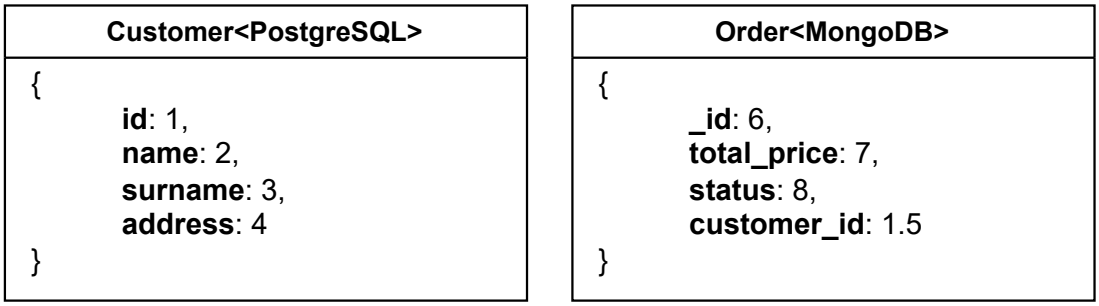


Figure 8.2: Mappings corresponding to the schema category in Figure 8.1.

seed for reproducibility. As for the size of the data, a set of 2000 customers and 6000 orders was used, with each customer being assigned 3 orders.

The evaluations were performed on a laptop with an 11th generation Intel i7 CPU, 32 gigabytes of operating memory and an SSD. Both databases as well as the MM-evocat instance were running locally on the same machine, using Docker[4] for virtualization.

It is also worth reminding that even though there is a single multi-model schema for all evaluations, we are able to use specific parts of it for the single-model evaluations.

All evaluations described in this chapter are part of the MM-quecat source code[5] as executable Python scripts in the `src/experiments` folder.

8.2 Single-Model Evaluation

As mentioned in the introduction of this chapter, the purpose of single-model evaluation is to isolate the overhead introduced by our approach compared to native database queries. In a single-model scenario, we do not need to consider the overhead of merging together data from multiple different databases, and we can focus on only the base overhead. A separate evaluation was carried out for both currently supported databases - PostgreSQL and MongoDB.

8.2.1 PostgreSQL

In the PostgreSQL evaluation, let us consider a query which simply selects all customers along with their properties, using the schema shown in Figure 8.1. Such a MMQL query is shown in Figure 8.3.

We also consider an equivalent PostgreSQL query shown in Figure 8.4, which was used as the benchmarking query. When compared to the query shown in Figure 8.5, which was internally generated by MM-quecat for the given MMQL query, we can see that they are functionally identical.

The results of this evaluation scenario can be seen in Table 8.1, with the first row showing the raw elapsed query time in milliseconds, and the second row showing slowdown relative to the native query. We can see that on average, the

[4] `https://www.docker.com/`

[5] `https://github.com/yawnston/querycat`

```
SELECT {
    ?customer id ?id ;
        name ?name ;
        surname ?surname ;
        address ?address .
}
WHERE {
    ?customer 1 ?id ;
        2 ?name ;
        3 ?surname ;
        4 ?address .
}
```

Figure 8.3: MMQL query used for PostgreSQL single-model evaluation.

```
SELECT id, name, surname, address
FROM experiments_customers
```

Figure 8.4: PostgreSQL query used for benchmarking performance.

MMQL query took a little over 1 second to execute, compared to just 4ms for the native query, giving us a slowdown of roughly 288 times.

	Native Query	**MM-evocat**
Elapsed Time	4ms	1153ms
Slowdown	1x	288x

Table 8.1: Average query time measurements for the single-model PostgreSQL scenario.

This degree of slowdown was within the realm of expectation, because we need to consider the overhead of communication with the MM-evocat instance, as well as the overhead of transforming the data into a categorical representation. However, what is more interesting is to take a look at where the majority of the slowdown is coming from. Looking at Figure 8.6, we can see performance measurement data collected by cProfile and visualized by SnakeViz[6]. This data shows a visualization of the total execution time spent in given function calls, starting at the top and decomposing the function calls further as we go down in the image.

Some function names are omitted from the image due to the available space, but examining the data contained within reveals the main takeaway of this evaluation scenario: the majority of the time was spent on **network communication** with MM-evocat. This network communication necessarily includes the transfer of all retrieved data from MM-evocat to MM-quecat. In other words, the amount of time spent otherwise manipulating the data was trivial. This gives us a good

[6]The profiler run was done separately from the performance measurement runs shown in Table 8.1 in order to eliminate any possible performance effect of the profiler itself.

```
SELECT
  experiments_customers.id AS experiments_customers_id,
  experiments_customers.name AS experiments_customers_name,
  experiments_customers.surname AS experiments_customers_surname,
  experiments_customers.address AS experiments_customers_address
FROM
  experiments_customers
```

Figure 8.5: PostgreSQL query generated by MM-quecat from the query shown in Figure 8.3.

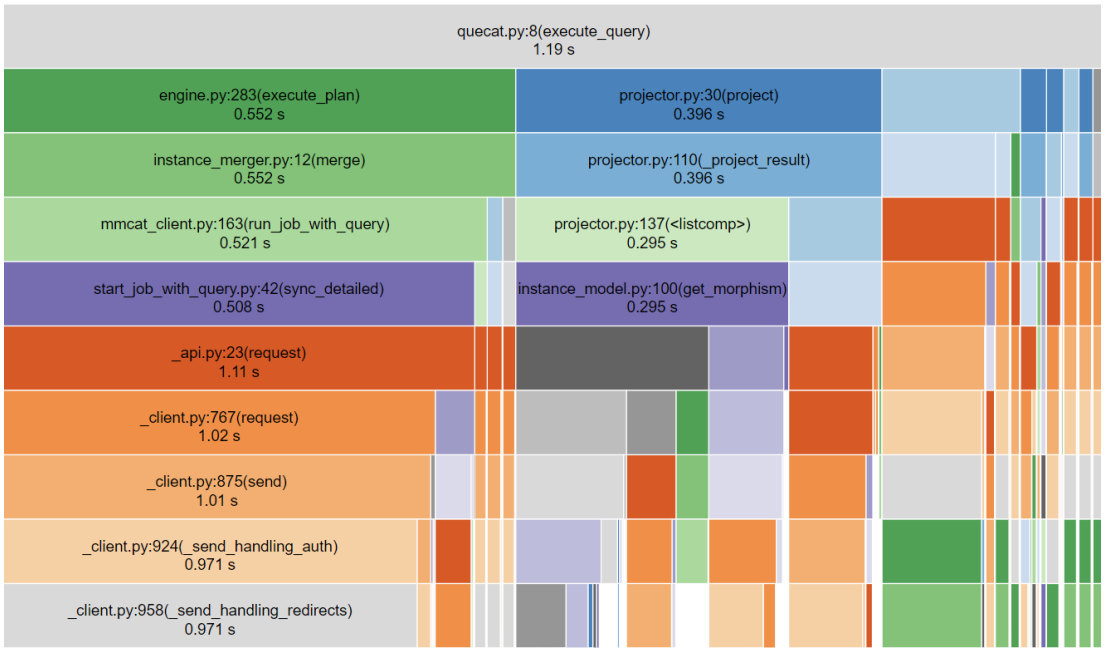


Figure 8.6: Profiler data for the single-model PostgreSQL scenario.

baseline idea of the performance of MM-quecat, as we can expect that execution times for arbitrary queries will likely not go too far below 1 second.

The only feasible way of reducing this overhead further would be to revise the architecture of the ecosystem including MM-evocat and MM-quecat, and to merge them together into a single tool. However, that would be far from trivial, considering the relevant tools were developed independently as part of different theses using different languages. Such an endeavor may not be worth the performance benefits at this point of the tools' lifecycle. Perhaps if in the future a highly performant variant of both MM-evocat and MM-quecat is desired, a new, unified tool can be built from scratch using the knowledge gained from the design and implementation of both tools.

8.2.2 MongoDB

Similarly to Section 8.2.1, this evaluation scenario evaluates MM-quecat in the context of a single data model within a single database. The evaluated MMQL query for this scenario is shown in Figure 8.7, the benchmark MongoDB query

is shown in Figure 8.8. The query generated by MM-quecat is identical to the query shown in Figure 8.8, therefore it is not shown in a separate figure.

```
SELECT {
    ?order id ?orderId ;
        totalPrice ?totalPrice ;
        status ?status ;
        customerId ?customerId .
}
WHERE {
    ?order 6 ?orderId ;
        7 ?totalPrice ;
        8 ?status ;
        5/1 ?customerId .
}
```

Figure 8.7: MMQL query used for MongoDB single-model evaluation.

```
db.experiments_orders.aggregate([
    {
        "$project": {
            "_id": 1,
            "total_price": 1,
            "status": 1,
            "customer_id": 1
        }
    }
])
```

Figure 8.8: MongoDB query used for benchmarking performance, identical to the query generated by MM-quecat.

The results of this evaluation scenario can be seen in Table 8.2. We can see that the average execution time for the native MongoDB query was 16ms, while the average for MM-evocat was 5458ms. This gives us an approximate slowdown of 341x.

	Native Query	**MM-evocat**
Elapsed Time	16ms	5458ms
Slowdown	1x	341x

Table 8.2: Average query time measurements for the single-model MongoDB scenario.

The measurements show an increase in the overhead incurred by MM-evocat, and the reason for this becomes apparent when examining Figure 8.9. As we can see, network communication between MM-evocat and MM-quecat now only takes

up approximately 30% of the total query execution time. The remaining 70% is taken up by work being done in the `QueryProjector` class, specifically by the method `_contract_morphisms`.

Recall that in Section 5.2.7, we discussed the projection algorithm, including the need for so-called morphism *contractions*. A morphism contraction is an operation on the instance category, where we may need to contract multiple instance morphisms into a single one because of the required projection in the query. This is an operation which for m morphisms with data size n effectively requires $m-1$ joins between instance morphism domain rows, where each instance morphism may contain up to n domain rows. These joins are implemented using a linear number of operations and a hash table in MM-quecat, however there may be room for optimization here. Perhaps a more efficient contraction algorithm could be designed, eliminating the need for hashing by clever usage of memory layout for the data being joined. Similarly, the representation of the instance category is not optimized for performance, therefore there may be some room for improvement here as well. In any case, the instance morphism contractions form a major performance bottleneck in MM-quecat, along with the serialization and deserialization of the instance category also having room for optimization.

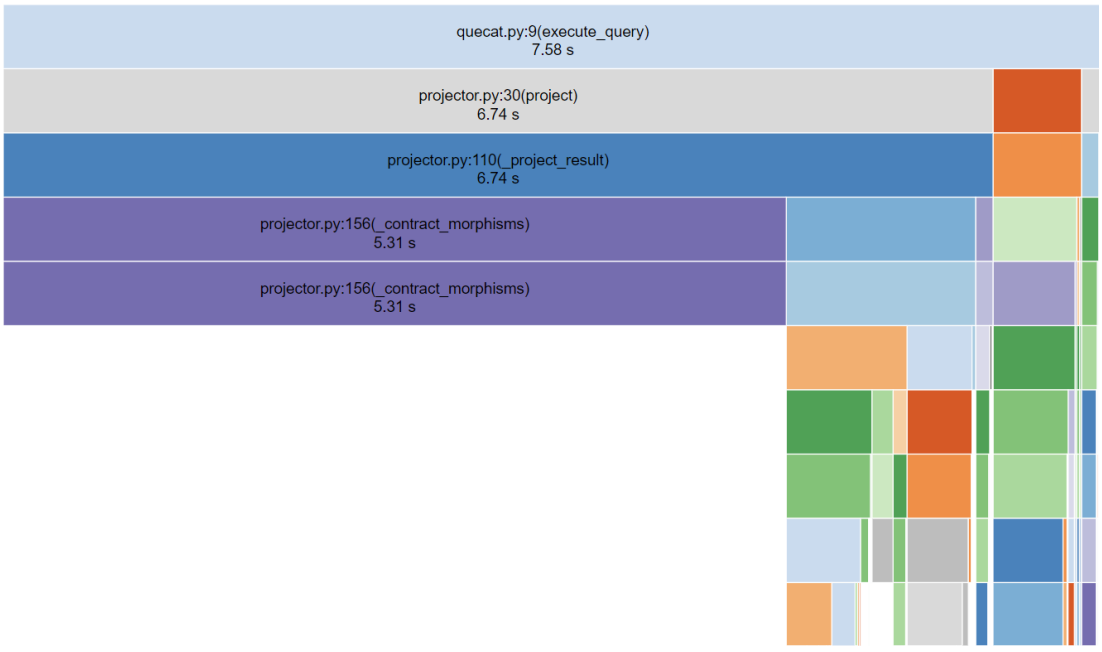


Figure 8.9: Profiler data for the single-model MongoDB scenario.

The reader may be wondering why we did not see this effect of the contractions in Section 8.2.1. The reason for this can be seen in the structure of the MMQL query itself - the query in this scenario contains the triple `?order 5/1 ?customerId`, but this compound morphism is contracted into a single base morphism in the `SELECT` clause. For that reason, the contractions are needed in this instance, whereas in Section 8.2.1, no such operation was needed due to the structure of the query.

8.3 Multi-Model Evaluation

In the previous section, we examined the characteristics of MM-quecat in the context of querying a single database at a time. However, one of the main benefits of MM-quecat and MMQL is the ability to uniformly query data from multiple databases at a time. Also, recall that one of the goals for this chapter was to understand the overhead in the context of merging data in a multi-model, which we have not yet done. This is why this last evaluation scenario is focused on MM-quecat in a multi-model context.

```
SELECT {
    ?order id ?orderId ;
        totalPrice ?totalPrice ;
        status ?status ;
        customerName ?customerName ;
        customerSurname ?customerSurname ;
        address ?customerAddress .
}
WHERE {
    ?order 6 ?orderId ;
        7 ?totalPrice ;
        8 ?status ;
        5 ?customer .

    ?customer 2 ?customerName ;
        3 ?customerSurname ;
        4 ?customerAddress .
}
```

Figure 8.10: MMQL query used for multi-model evaluation.

Recall Figure 8.1, which shows the schema category used in these evaluations. In this scenario, we will finally be using the entire schema category. The MMQL query in Figure 8.10 selects data for each order from MongoDB, while also selecting customer data from PostgreSQL and joining the data based on the customer ID stored with the orders.

As this is a multi-model scenario, we will be running queries in both PostgreSQL and MongoDB, and their combined elapsed time form the performance baseline. These baseline queries are the same as in the previous section (i.e. Figure 8.4 and Figure 8.8), we will just be using both of them at the same time. We will also not be showing the queries generated by MMQL in this instance, as they are virtually identical to the generated queries in the previous scenarios.

	Native Queries	**MM-evocat**
Elapsed Time	46ms	19209ms
Slowdown	1x	417x

Table 8.3: Average query time measurements for the multi-model scenario.

The results for this final scenario are shown in Table 8.3. We can see that the native queries combined took approximately 46 milliseconds on average, whereas MM-evocat finished its work in around 19 seconds.

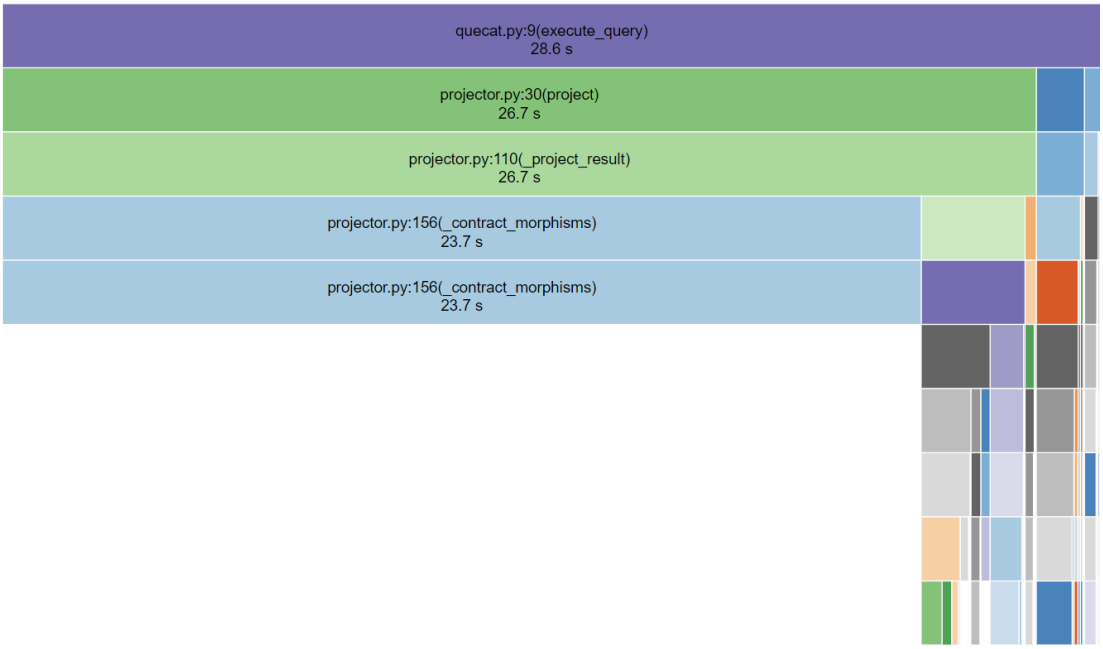


Figure 8.11: Profiler data for the multi-model scenario.

Examining the profiler data shown in Figure 8.11, we can see a similar image as in Section 8.2.2. Again, the vast majority of the query execution time is spent on instance morphism contraction, this time over 80%. In this instance, the fraction of time spent on contractions is higher, because the query itself also has more contractions to perform. While the customer ID is not being retrieved this time, all other customer properties are, leading to additional contractions needing to be done. This, along with the combined volume of data from both databases, explains the increase in query execution time. Again, as discussed in Section 8.2.2, perhaps there exist more optimized ways of doing the instance morphism contractions, which would improve the overall query performance dramatically.

In addition, we can see that the merging process between the two models does not take a significant amount of time, at least compared to the effort required to perform morphism contractions. This makes sense, as the merging process does not need to process that much data if the joining object has a simple enough identifier, which is used for the join. Therefore future efforts in improving the querying performance should not focus on the merging algorithm, but rather on the contraction process (and the projection in general). In other words, for queries whose projection is trivial, performance improvements are likely to be most effective in the network communication between MM-evocat and MM-quecat, whereas for queries with non-trivial projection, improvements to the morphism contraction will yield the most benefit.

Conclusion

Multi-model data is a very complex domain with many unsolved problems and much additional research needed, as the reader is surely aware after reading this thesis. This is especially true for the problem of unified querying of multi-model data, for which no widely usable proposals exist at the time of writing of this thesis, with the vast majority of existing multi-model querying solutions necessitating the knowledge of the specifics of each model involved. Despite these challenges, in this thesis we worked towards creating one of the first proposals for a unified multi-model querying approach, as we believe that enabling the formation of multi-model queries in a model-agnostic way has massive potential upsides.

To this end, we first introduced the unified multi-model data representation [5][6] which we based our efforts on. Following this, we examined the existing field of graph query languages, as a category may be thought of as a directed multigraph, and selected SPARQL as the prime candidate for adaption to a categorical domain. We then proposed MMQL, a categorical multi-model query language which given the aforementioned unified categorical data representation, allows users to query across multiple models and databases in a unified, model- and database-agnostic fashion. In addition to introducing all MMQL concepts together with concrete query examples, we also provided a full formal grammar for this proposed query language. The main characteristics of MMQL include strong expressiveness in terms of matching graph patterns, familiarity thanks to structural similarities to SPARQL, and leveraging the power of the categorical representation to form elegant graph traversals.

A query language is nothing without the supporting algorithms necessary for its implementation. For this reason, following the design of MMQL, we also proposed an approach for the implementation of MMQL for multi-model, multi-database scenarios potentially involving data redundancy. The main accomplishment of our proposed approach is the fact that it can be decomposed into clear, well-defined steps, whose composition forms the whole implementation algorithm. Since this is the first such approach designed, we focused our design on comprehensibility and simplicity, discussing additional complexities along the way. During our presentation of our proposed approach for implementing MMQL, we made a special effort to point out any flaws or limitations of our approach, as this will allow further work on this subject to iterate upon our solution.

Following the lengthy design chapters about MMQL and its supporting algorithms, we put our designs to the test by creating a proof-of-concept implementation of MMQL called MM-quecat. This implementation is limited in scope by the amount of time and work required to propose the entire approach, as well as by constraints placed upon it due to the dependence on another piece of software called MM-evocat, which is in active development, and not all of its required features are finished. Despite this, MM-quecat serves its purpose as a verification of the validity of our process, functioning as a unified query solution for a subset of MMQL for the PostgreSQL and MongoDB databases. Although we present MM-quecat in this thesis as it exists in its current state, its feature set will continue to evolve in the future, as the author of this thesis is the co-

author of a not-yet-published demo paper showcasing MM-quecat and its unified multi-model querying capabilities, which we are excited to share with the wider multi-model data community. Related to this, we also presented our proposal for how a graphical query tool for MMQL may look like as part of MM-quecat, demonstrating the final product we will be striving for with our future academic endeavors.

In order to properly evaluate the weaknesses and limitations of our approach, we also performed a handful of experimental evaluations of MM-quecat in an experiment involving PostgreSQL and MongoDB. We acknowledge that performance is key in the world of multi-model data, but achieving near-native performance is simply too ambitious for an approach with little to no previous related work to support it. For this reason, we collected query execution time data as well as profiler data during the experiments, and we discussed their implications for our approach.

Overall, we feel that we accomplished all goals which we outlined in the introduction of this thesis, having proposed an innovative approach for unified multi-model querying complete with our own query language called MMQL, laying the groundwork for future research.

Future Work

While presenting MMQL, we provided a full formal grammar, discussed its features, and we showed a comparison of its feature set to existing single-model query languages. However, we believe that a more formal analysis and verification of the language may be desirable, in order to formally express the capabilities and limitations of MMQL. Similarly, MMQL may be further extended with features like more aggregation or filtering options, together with additional data types.

When it comes to our proposed MMQL implementation approach, we mentioned a number of open problems in the world of multi-model data which require further study. More work is needed in the area of multi-model query planning, as there are limited academic resources on this matter, and existing multi-model query planners within polystores do not fit neatly into our problem domain, as they do not take into considerations many variables relevant to our approach. Together with multi-model query planning, we also mentioned the problem of multi-model join ordering, which also lacks robust and general solutions.

When it comes to the MMQL implementation approach proposed in this thesis, we acknowledge that it has many limitations and weaknesses, which we discussed at great length in various chapters. Perhaps the largest one of them all is performance, and with performance and scalability often being the driving force behind using multiple data models to begin with, we recognize the need for more optimal versions of algorithms we proposed, possibly using some or all of the optimizations we discussed along the way. There is room for optimization in almost all areas of the proposed approach, from ensuring that the generated native queries are as optimal as possible, to optimizing the manipulation of categorical data in order to minimize the overhead introduced. In general, the goal for unified multi-model querying in the future should be to achieve near-native performance when compared to individual database systems while preserving the benefits of unified querying.

Lastly, our MMQL implementation called MM-quecat is purely experimental in nature, and it does not contain all MMQL features for various reasons outlined in this thesis. In order to bring MM-quecat closer to real-world applicability, we will continue to improve and enhance its implementation as part of our future research efforts. Coupled with this, we believe that a graphical query tool would be highly beneficial for the end user experience, which is why we are also working on the user interface part of MM-quecat, which we hope to demonstrate to the academic community in the coming months.

Bibliography

[1] Valter Uotila, Jiaheng Lu, Dieter Gawlick, Zhen Hua Liu, Souripriya Das, and Gregory Pogossiants. MultiCategory. *Proceedings of the VLDB Endowment*, 14(12):2663–2666, jul 2021.

[2] Valter Uotila and Jiaheng Lu. A formal category theoretical framework for multi-model data transformations. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, pages 14–28. Springer International Publishing, 2021.

[3] Patrick Schultz, David I. Spivak, Christina Vasilakopoulou, and Ryan Wisnesky. Algebraic databases, 2016.

[4] Chris Tuijn and Marc Gyssens. Cgood, a categorical graph-oriented object data model. *Theor. Comput. Sci.*, 160:217–239, 1996.

[5] Martin Svoboda, Pavel Čontoš, and Irena Holubová. *Categorical Modeling of Multi-model Data: One Model to Rule Them All*, pages 190–198. 06 2021.

[6] Pavel Koupil and Irena Holubová. A unified representation and transformation of multi-model data using category theory. *Journal of Big Data*, 9, 05 2022.

[7] Michael Stonebraker. The case for polystores [online]. `http://wp.sigmod.org/?p=1629`. Accessed: 2022-12-19.

[8] Boyan Kolev, Raquel Pau, Oleksandra Levchenko, Patrick Valduriez, Ricardo Jiménez-Peris, and José Pereira. Benchmarking polystores: The cloudmdsql experience. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2574–2579, 2016.

[9] Gianina Mihai (Rizescu). Multi-model database systems: The state of affairs. *Annals of Dunarea de Jos University of Galati Fascicle I Economics and Applied Informatics*, XXVI:211–215, 01 2020.

[10] Jiaheng Lu and Irena Holubová. Multi-model databases: A new journey to handle the variety of data. *ACM Comput. Surv.*, 52(3), jun 2019.

[11] Michael Stonebraker. Sql databases v. nosql databases. *Commun. ACM*, 53(4):10–11, apr 2010.

[12] Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. Unibench: A benchmark for multi-model database management systems: Recognizing outstanding ph.d. research. pages 7–23, 01 2019.

[13] Pavel Koupil. *Modelling and Management of Multi-Model Data*. PhD thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, 2022.

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

[15] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, mar 1976.

[16] Pavel Koupil, Sebastián Hricko, and Irena Holubová. A universal approach for multi-model schema inference. *Journal of Big Data*, 9, 08 2022.

[17] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 1150–1160. VLDB Endowment, 2007.

[18] Andrew Pavlo and Matthew Aslett. What's really new with newsql? *SIGMOD Rec.*, 45(2):45–55, sep 2016.

[19] Omji Mishra, Pooja Lodhi, and Shikha Mehta. *Document Oriented NoSQL Databases: An Empirical Study*, pages 126–136. 10 2018.

[20] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. The case against specialized graph analytics engines. In *Conference on Innovative Data Systems Research*, 2015.

[21] Tom Leinster. *Basic category theory*, volume 143. Cambridge University Press, 2014. pages 1-26.

[22] Pavel Koupil, Jáchym Bártík, and Irena Holubová. Mm-evocat: A tool for modelling and evolution management of multi-model data. In *Proceedings of the 31st ACM International Conference on Information &; Knowledge Management*, CIKM '22, page 4892–4896, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Pavel Koupil, Sebastián Hricko, and Irena Holubová. Schema inference for multi-model data. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, page 13–23, New York, NY, USA, 2022. Association for Computing Machinery.

[24] SPARQL Query Language for RDF [online]. `https://www.w3.org/TR/rdf-sparql-query/`. Accessed: 2022-07-12.

[25] openCypher [online]. `https://opencypher.org/`. Accessed: 2022-07-12.

[26] Gremlin [online]. `https://tinkerpop.apache.org/gremlin.html`. Accessed: 2022-07-15.

[27] Property Graph Query Language [online]. `https://pgql-lang.org/`. Accessed: 2022-12-17.

[28] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. G-core: A core for future graph query languages, 2017.

[29] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of modern graph query languages. *CoRR*, abs/1610.06264, 2016.

[30] Resource Description Framework [online]. `https://www.w3.org/TR/rdf11-concepts/`. Accessed: 2022-07-12.

[31] SPARQL 1.1 Update [online]. `https://www.w3.org/TR/sparql11-update/`. Accessed: 2022-07-14.

[32] Pavel Koupil, Daniel Crha, and Irena Holubová. Mm-quecat: A tool for unified querying of multi-model data [not yet published].

[33] SPARQL 1.1 Property Paths [online]. `https://www.w3.org/TR/sparql11-property-paths/`. Accessed: 2022-12-18.

[34] Erich Gamma, Richard Helm, and Ralph E. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, 1st ed., reprint. edition, 1994.

[35] Jin Chen, Guanyu Ye, Yan Zhao, Shuncheng Liu, Liwei Deng, Xu Chen, Rui Zhou, and Kai Zheng. Efficient join order selection learning with graph-based representation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '22, page 97–107, New York, NY, USA, 2022. Association for Computing Machinery.

[36] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, jun 1984.

[37] David I. Spivak and Ryan Wisnesky. Relational foundations for functorial data migration. In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, page 21–28, New York, NY, USA, 2015. Association for Computing Machinery.

[38] Jiaheng Lu and Irena Holubová. Multi-model data management: What's new and what's next? In *International Conference on Extending Database Technology*, 2017.

[39] Yingfei Xiong, Zhenjiang Hu, Dongxi Liu, Haiyan Zhao, Hong Mei, and Masato Takeichi. Realizing bidirectional graph transformations from bidirectional tree transformations. 01 2008.

List of Figures

List of Tables

List of Abbreviations

ACID Atomicity, Consistency, Isolation, Durability
API Application Programming Interface
AST Abstract Syntax Tree
CPU Central Processing Unit
CQL Categorical Query Language
ER Entity Relationship diagram
HTTP Hypertext Transfer Protocol
IRI Internationalized Resource Identifier
JSON JavaScript Object Notation
MMQL Multi-Model Query Language
OLAP Online Analytical Processing
OLTP Online Transaction Processing
ORM Object-Relational Mapping
RDF Resource Description Framework
REST Representational State Transfer
SPARQL SPARQL Protocol and RDF Query Language
SSD Solid-State Drive
SQL Structured Query Language
UI User Interface
XML Extensible Markup Language

A. Attachments

A.1 MMQL Grammar

This attachment contains a full formal grammar of MMQL, which was used to generate a parser for the query language. The grammar is in the G4 format.

```
grammar MMQL;

query: selectQuery EOF;

selectQuery: selectClause fromClause? whereClause solutionModifier;

subSelect: selectQuery;

selectClause: 'SELECT' selectGraphPattern;

selectGraphPattern: '{' selectTriples? '}';

fromClause: 'FROM' SCHEMA_IDENTIFIER;

whereClause: 'WHERE'? groupGraphPattern;

solutionModifier: orderClause? limitOffsetClauses?;

limitOffsetClauses: (
                limitClause offsetClause?
                | offsetClause limitClause?
        );

orderClause: 'ORDER' 'BY' orderCondition+;

orderCondition: (( 'ASC' | 'DESC') brackettedExpression)
        | ( constraint | var_);

limitClause: 'LIMIT' INTEGER;

offsetClause: 'OFFSET' INTEGER;

groupGraphPattern:
        '{' (subSelect | (triplesBlock? (
                (graphPatternNotTriples | filter_) '.'? triplesBlock?
        )*)) '}';

triplesBlock: triplesSameSubject ( '.' triplesBlock?)?;

graphPatternNotTriples:
        optionalGraphPattern
        | groupOrUnionGraphPattern
        | inlineData;
```

```
optionalGraphPattern: 'OPTIONAL' groupGraphPattern;

groupOrUnionGraphPattern:
        groupGraphPattern (('UNION' | 'MINUS') groupGraphPattern)*;

inlineData: 'VALUES' dataBlock;

dataBlock: var_ '{' dataBlockValue* '}';

dataBlockValue: numericLiteral
        | booleanLiteral
        | string_;

filter_: 'FILTER' constraint;

constraint: brackettedExpression;

selectTriples: triplesSameSubject ( '.' selectTriples?)?;

triplesSameSubject: varOrTerm propertyListNotEmpty;

propertyListNotEmpty: verb objectList ( ';' ( verb objectList)?)*;

propertyList: propertyListNotEmpty?;

objectList: object_ ( ',' object_)*;

object_: graphNode;

verb: schemaMorphismOrPath;

schemaMorphismOrPath: pathAlternative;

pathAlternative: pathSequence ( '|' pathSequence )*;

pathSequence: pathWithMod ( '/' pathWithMod )*;

pathWithMod: pathPrimary pathMod?;

pathMod: '?' | '*' | '+';

pathPrimary: schemaMorphism | ( '(' schemaMorphismOrPath ')' ) ;

schemaMorphism: primaryMorphism | dualMorphism;

primaryMorphism: SCHEMA_MORPHISM;

dualMorphism: '-' primaryMorphism;

graphNode: varOrTerm ( 'AS' var_)?;
```

```
varOrTerm: var_ | constantTerm | aggregationTerm;

var_: VAR1 | VAR2;

constantTerm:
        numericLiteral
        | booleanLiteral
        | string_
        | blankNode
        | NIL;

aggregationTerm:
        aggregationFunc '(' (distinctModifier)? var_ ')';

distinctModifier:
        'DISTINCT';

aggregationFunc:
        'COUNT'
        | 'SUM'
        | 'AVG'
        | 'MIN'
        | 'MAX';

expression: conditionalOrExpression;

conditionalOrExpression:
        conditionalAndExpression ('||' conditionalAndExpression)*;

conditionalAndExpression: valueLogical ( '&&' valueLogical)*;

valueLogical: relationalExpression;

relationalExpression:
        expressionPart (
                '=' expressionPart
                | '!=' expressionPart
                | '<' expressionPart
                | '>' expressionPart
                | '<=' expressionPart
                | '>=' expressionPart
        )?;

expressionPart: primaryExpression;

primaryExpression:
        brackettedExpression
        | numericLiteral
        | booleanLiteral
        | string_
```

```
        | varOrTerm;

brackettedExpression: '(' expression ')';

numericLiteral:
        numericLiteralUnsigned
        | numericLiteralPositive
        | numericLiteralNegative;

numericLiteralUnsigned: INTEGER | DECIMAL | DOUBLE;

numericLiteralPositive:
        INTEGER_POSITIVE
        | DECIMAL_POSITIVE
        | DOUBLE_POSITIVE;

numericLiteralNegative:
        INTEGER_NEGATIVE
        | DECIMAL_NEGATIVE
        | DOUBLE_NEGATIVE;

booleanLiteral: 'true' | 'false';

string_:
        STRING_LITERAL1
        | STRING_LITERAL2;

blankNode: BLANK_NODE_LABEL | ANON;

// LEXER RULES

SCHEMA_MORPHISM: (PN_CHARS)+;

SCHEMA_IDENTIFIER: (PN_CHARS)+;

BLANK_NODE_LABEL: '_:' PN_LOCAL;

VAR1: '?' VARNAME;

VAR2: '$' VARNAME;

INTEGER: DIGIT+;

DECIMAL: DIGIT+ '.' DIGIT* | '.' DIGIT+;

DOUBLE:
        DIGIT+ '.' DIGIT* EXPONENT
        | '.' DIGIT+ EXPONENT
        | DIGIT+ EXPONENT;

INTEGER_POSITIVE: '+' INTEGER;
```

```
DECIMAL_POSITIVE: '+' DECIMAL;

DOUBLE_POSITIVE: '+' DOUBLE;

INTEGER_NEGATIVE: '-' INTEGER;

DECIMAL_NEGATIVE: '-' DECIMAL;

DOUBLE_NEGATIVE: '-' DOUBLE;

EXPONENT: ('e' | 'E') ('+' | '-')? DIGIT+;

STRING_LITERAL1:
        '\'' (
                ~('\u0027' | '\u005C' | '\u000A' | '\u000D') | ECHAR
        )* '\'';

STRING_LITERAL2:
        '"' (
                ~('\u0022' | '\u005C' | '\u000A' | '\u000D') | ECHAR
        )* '"';

STRING_LITERAL_LONG1:
        '\'\'\'' (
                ( '\'' | '\'\'')? (~('\'' | '\\') | ECHAR)
        )* '\'\'\'';

STRING_LITERAL_LONG2:
        '"""' (
                ( '"' | '""')? ( ~('\'' | '\\') | ECHAR)
        )* '"""';

ECHAR: '\\' ('t' | 'b' | 'n' | 'r' | 'f' | '"' | '\'');

NIL: '(' WS* ')';

ANON: '[' WS* ']';

PN_CHARS_U: PN_CHARS_BASE | '_';

VARNAME: (PN_CHARS_U | DIGIT) (
                PN_CHARS_U
                | DIGIT
                | '\u00B7'
                | ('\u0300' ..'\u036F')
                | ('\u203F' ..'\u2040')
        )*;

fragment PN_CHARS:
        PN_CHARS_U
```

```
        | '-'
        | DIGIT;

PN_PREFIX: PN_CHARS_BASE ((PN_CHARS | '.')* PN_CHARS)?;

PN_LOCAL: ( PN_CHARS_U | DIGIT) ((PN_CHARS | '.')* PN_CHARS)?;

fragment PN_CHARS_BASE:
        'A' ..'Z'
        | 'a' ..'z'
        | '\u00C0' ..'\u00D6'
        | '\u00D8' ..'\u00F6'
        | '\u00F8' ..'\u02FF'
        | '\u0370' ..'\u037D'
        | '\u037F' ..'\u1FFF'
        | '\u200C' ..'\u200D'
        | '\u2070' ..'\u218F'
        | '\u2C00' ..'\u2FEF'
        | '\u3001' ..'\uD7FF'
        | '\uF900' ..'\uFDCF'
        | '\uFDF0' ..'\uFFFD';

fragment DIGIT: '0' ..'9';

WS: (' ' | '\t' | '\n' | '\r')+ -> skip;
```

A.2 Query Planner Information in PostgreSQL

Below, we can see the query planner information exposed by the `EXPLAIN` statement in PostgreSQL for a simple query selecting all theaters whose primary identifier is greater than a constant number. We can see that the set of information returned contains a full description of the winning query plan, complete with estimated costs for each step of the plan, and the information about which indexes the query is using.

```
EXPLAIN (FORMAT YAML) SELECT * FROM theaters WHERE id > 1010;

          QUERY PLAN
------------------------------
 - Plan:
    Node Type: "Bitmap Heap Scan"
    Parallel Aware: false
    Async Capable: false
    Relation Name: "theaters"
    Alias: "theaters"
    Startup Cost: 6.34
    Total Cost: 19.88
    Plan Rows: 283
    Plan Width: 68
    Recheck Cond: "(id > 1010)"
    Plans:
      - Node Type: "Bitmap Index Scan"
        Parent Relationship: "Outer"
        Parallel Aware: false
        Async Capable: false
        Index Name: "theaters_pkey"
        Startup Cost: 0.00
        Total Cost: 6.27
        Plan Rows: 283
        Plan Width: 0
        Index Cond: "(id > 1010)"
(1 row)
```

A.3 Query Planner Information in MongoDB

Below, we can see two sets of query planner information returned by MongoDB. The first output shown calls the `explain()` function with the default `queryPlanner` mode, which does not execute the query, but also does not include cost or the approximate result size.

The second output uses the `executionStats` mode, which executes the query, and subsequently returns query plan information including cost information (represented in the form of the `works` property signifying units of work required).

```
db.theaters.explain().find({ theaterId: { $gt: 1020 } });
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "sample_mflix.theaters",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "theaterId" : {
        "$gt" : 1020
      }
    },
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "IXSCAN",
        "keyPattern" : {
          "theaterId" : 1
        },
        "indexName" : "theaterId_1",
        "isMultiKey" : false,
        "multiKeyPaths" : {
          "theaterId" : [ ]
        },
        "isUnique" : true,
        "isSparse" : false,
        "isPartial" : false,
        "indexVersion" : 2,
        "direction" : "forward",
        "indexBounds" : {
          "theaterId" : [
            "(1020.0, inf.0]"
          ]
        }
      }
    },
    "rejectedPlans" : [ ]
  }
}
```

```
db.theaters.explain({verbosity: "executionStats"}).find(
    {theaterId: {$gt: 1020}}
);
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "sample_mflix.theaters",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "theaterId" : {
        "$gt" : 1020
      }
    },
    "winningPlan" : {
      "stage" : "FETCH",
      "inputStage" : {
        "stage" : "IXSCAN",
        "keyPattern" : {
          "theaterId" : 1
        },
        "indexName" : "theaterId_1",
        "isMultiKey" : false,
        "multiKeyPaths" : {
          "theaterId" : [ ]
        },
        "isUnique" : true,
        "isSparse" : false,
        "isPartial" : false,
        "indexVersion" : 2,
        "direction" : "forward",
        "indexBounds" : {
          "theaterId" : [
            "(1020.0, inf.0]"
          ]
        }
      }
    },
    "rejectedPlans" : [ ]
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 861,
    "executionTimeMillis" : 1,
    "totalKeysExamined" : 861,
    "totalDocsExamined" : 861,
    "executionStages" : {
      "stage" : "FETCH",
      "nReturned" : 861,
      "executionTimeMillisEstimate" : 0,
      "works" : 862,
```

```
      "advanced" : 861,
      "needTime" : 0,
      "needYield" : 0,
      "saveState" : 6,
      "restoreState" : 6,
      "isEOF" : 1,
      "invalidates" : 0,
      "docsExamined" : 861,
      "alreadyHasObj" : 0,
      "inputStage" : {
        "stage" : "IXSCAN",
        "nReturned" : 861,
        "executionTimeMillisEstimate" : 0,
        "works" : 862,
        "advanced" : 861,
        "needTime" : 0,
        "needYield" : 0,
        "saveState" : 6,
        "restoreState" : 6,
        "isEOF" : 1,
        "invalidates" : 0,
        "keyPattern" : {
          "theaterId" : 1
        },
        "indexName" : "theaterId_1",
        "isMultiKey" : false,
        "multiKeyPaths" : {
          "theaterId" : [ ]
        },
        "isUnique" : true,
        "isSparse" : false,
        "isPartial" : false,
        "indexVersion" : 2,
        "direction" : "forward",
        "indexBounds" : {
          "theaterId" : [
            "(1020.0, inf.0]"
          ]
        },
        "keysExamined" : 861,
        "seeks" : 1,
        "dupsTested" : 0,
        "dupsDropped" : 0,
        "seenInvalidated" : 0
      }
    },
    "allPlansExecution" : [ ]
  }
}
```

A.4 Query Planner Information in Neo4j

Below, we can see the query planner information exposed by the `EXPLAIN` statement in Neo4j for a simple query matching nodes with the `name` property set to "Tom Hanks". For each stage in the query plan, the estimated number of rows is returned.


```
{
  "query": {
    "text": "EXPLAIN MATCH (tom {name: \"Tom Hanks\"}) RETURN tom",
    "parameters": {}
  },
  "queryType": "r",
  "counters": {
    "_stats": {
      "nodesCreated": 0,
      "nodesDeleted": 0,
      "relationshipsCreated": 0,
      "relationshipsDeleted": 0,
      "propertiesSet": 0,
      "labelsAdded": 0,
      "labelsRemoved": 0,
      "indexesAdded": 0,
      "indexesRemoved": 0,
      "constraintsAdded": 0,
      "constraintsRemoved": 0
    },
    "_systemUpdates": 0
  },
  "updateStatistics": {
    "_stats": {
      "nodesCreated": 0,
      "nodesDeleted": 0,
      "relationshipsCreated": 0,
      "relationshipsDeleted": 0,
      "propertiesSet": 0,
      "labelsAdded": 0,
      "labelsRemoved": 0,
      "indexesAdded": 0,
      "indexesRemoved": 0,
      "constraintsAdded": 0,
      "constraintsRemoved": 0
    },
    "_systemUpdates": 0
  },
  "plan": {
    "operatorType": "ProduceResults@neo4j",
    "identifiers": [
      "tom"
    ],
    "arguments": {
      "planner-impl": "IDP",
```

```
    "Details": "tom",
    "PipelineInfo": "Fused in Pipeline 0",
    "planner-version": "4.4",
    "runtime-version": "4.4",
    "runtime": "PIPELINED",
    "runtime-impl": "PIPELINED",
    "version": "CYPHER 4.4",
    "EstimatedRows": 17,
    "planner": "COST"
  },
  "children": [
    {
      "operatorType": "Filter@neo4j",
      "identifiers": [
        "tom"
      ],
      "arguments": {
        "Details": "tom.name = $autostring_0",
        "EstimatedRows": 17,
        "PipelineInfo": "Fused in Pipeline 0"
      },
      "children": [
        {
          "operatorType": "AllNodesScan@neo4j",
          "identifiers": [
            "tom"
          ],
          "arguments": {
            "Details": "tom",
            "EstimatedRows": 340,
            "PipelineInfo": "Fused in Pipeline 0"
          },
          "children": []
        }
      ]
    }
  ]
},
"profile": false,
"notifications": [],
"server": {
  "address": "localhost:7687",
  "version": "Neo4j/4.4.5",
  "agent": "Neo4j/4.4.5",
  "protocolVersion": 4.4
},
"resultConsumedAfter": {
  "low": 0,
  "high": 0
},
"resultAvailableAfter": {
```

```
    "low": 0,
    "high": 0
  },
  "database": {
    "name": "neo4j"
  }
}
```