Creating SQL Query Input in Atlas: Designing a Virtual Schema for Query Transformation

Maria João Madeira Duarte

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Pedro Manuel Moreira Vaz Antunes de Sousa

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Pedro Manuel Moreira Vaz Antunes de Sousa
Member of the Committee: Prof. Flávio Nuno Fernandes Martins

May 2025

Acknowledgments

Primeiramente, agradeço ao meu orientador, Professor Pedro Manuel Moreira Vaz Antunes de Sousa, que esteve disponível para me guiar e dar conhecimento no que precisava, desde o início ao fim desta tese. Sem ele este trabalho não seria possível.

Estou muito grata a todos os meus colegas da Link que me motivaram nesta jornada. David Moreira, obrigada pela tua disponibilidade ao longo destes meses. Facilitaste a compreensão da complexidade do código Atlas e esclareceste todas as minhas questões.

Gostava de agradecer à minha família. Mãe e pai, o sonho de fazer este curso foi construído ao vosso lado. Partilharam comigo o sentimento de êxtase quando fui colocada no Instituto Superior Técnico e ainda mais o nervosismo quando deixei a nossa cidade rumo à capital. Obrigada por me incentivarem a seguir os meus interesses e a lutar por eles. Foram anos de esforço e longe de casa que me ensinaram a valorizar a sorte que tenho em vos ter.

Casa também é sinónimo da minha irmã Inês, da avó Lina, do tio Licínio, da tia Margarida, e dos patudos Laika e Sky. Foram anos de muita saudade, e a contar os dias até vos voltar a ver. Obrigada pelo quanto fizeram por mim, por o quanto me ajudaram a ser quem sou.

Aos que partiram nos meus anos de estudante, avó Maria, avô Licínio, e avó Ana, nada me saberia melhor do que poder partilhar esta e futuras conquistas convosco. Obrigada por tudo o que fizeram. Por tudo o que foram.

Aos que chegaram, os meus sobrinhos Henrique e Bernardo, que enquanto viam a tia a aprender a programar, desenvolviam capacidades ainda mais fascinantes: saber andar e falar. Obrigada por me darem uma alegria inexplicável, adoro-vos.

À minha grande companheira de curso e melhor amiga, Laura, obrigada por seres a minha casa longe de Coimbra. Vivemos e alcançámos tanto juntas, as nossas conquistas souberam melhor por serem partilhadas tão intensamente.

Um agradecimento a ti Sacra, pelo teu apoio e amizade numa altura difícil. Estendeste a mão e ajudaste-me a recuperar uma visão otimista. Adoro-te muito.

A todos os membros da Tuna Feminina do Instituto Superior Técnico, agradeço por me receberem de braços abertos numa casa que me é tão querida. Foram dois anos de Júlias Floristas, viagens, ensinamentos e, principalmente, união e espírito de equipa. Encontrei pessoas que tanto valorizo e respeito, que continuamente deram parte de si para levar este grupo a bom porto.

Aos meus amigos das 13 Tragédias, nomeadamente à Regina, ao Martim e ao Gui, continuarei a relembrar o quanto nos divertimos naquelas férias em Arganil. Até ao nosso próximo jantar.

Ao grupo de amigos da Margem Sul, Pedreiro, Margarida, Rafa e Vanessa, obrigada pelas inúmeras noites que terminaram no Bardo a falarmos da vida e a jogarmos algo novo. Espero que continuemos assim.

A toda a família Pires Afonso, que passei a considerar minha também, obrigada por me receberem tão bem. Elvira e Afonso, foi debaixo do vosso teto que construí esta tese, e, acima de tudo, laços de muito respeito e carinho.

Por último, gostava de agradecer ao meu namorado pelo apoio incondicional. Micael, acompanhaste toda a evolução deste projeto, e o teu envolvimento e curiosidade geraram inspiração para me dedicar ainda mais. Nos momentos desafiantes fizeste-me perceber que basta dar um passinho de cada vez. Obrigada. Mesmo que nem sempre o escreva como agora, as minhas conquistas serão sempre dedicadas a ti.

A todos vocês que me moldam ou moldaram, que tornaram os maus momentos mais leves e os bons momentos merecedores de muito amor e partilha. Tenho muita sorte em vos levar comigo. Obrigada.

Abstract

This thesis explores the integration of a new Structured Query Language (SQL) query input mechanism for data retrieval within Atlas, an enterprise architecture tool developed by Link Consulting. The existing Graphical Query System in Atlas presents limitations in performance, as it relies on an inefficient execution process which involves interpreting Extensible Markup Language (XML) representations into executable statements. As a solution, this work introduces a Virtual Schema that allows intuitive and efficient SQL query formulation while maintaining compatibility with Atlas' relational model.

The project includes the design of an intuitive query format supported by the new Virtual Model, the development of a query translation mechanism that maps the SQL-based input to Atlas' relational schema, and the implementation of an execution optimization strategy that reduces redundant parsing by storing the translated queries. Additionally, we incorporate security measures to mitigate SQL injection risks, and adapt the user interface to integrate the new feature.

This thesis contributes to the field of enterprise data management by demonstrating how Virtual Schemas can serve as a bridge to create a secure SQL query search mechanism in complex relational database systems, as an efficient querying alternative.

Keywords

Virtual Schema; Relational Model; Schema Mapping; Query Translation; Query Optimization.

Resumo

Esta tese tem como objetivo a integração de um novo mecanismo de input de queries baseadas em Structured Query Language (SQL) para pesquisa de dados no Atlas, uma ferramenta de arquitetura empresarial desenvolvida pela Link Consulting. A plataforma integra um sistema de queries gráficas que revela limitações de desempenho, uma vez que apresenta um processo de execução ineficiente, baseado na interpretação de uma linguagem intermédia em Extensible Markup Language (XML) para queries executáveis. Para contornar esta limitação, este trabalho introduz a criação de um Modelo Virtual que permite a utilização intuitiva de SQL, enquanto garante a compatibilidade com o modelo relacional do Atlas.

A solução desenvolvida consiste na definição de uma estrutura de query intuitiva baseada num Modelo Virtual, na implementação de um mecanismo de tradução de queries capaz de mapear as queries inseridas para o modelo relacional do Atlas, assim como numa estratégia de otimização da execução que evita parsing redundante, através do armazenamento das queries traduzidas na base de dados. Adicionalmente, foram implementadas medidas de segurança para mitigar os riscos de injeção de SQL, e a interface de utilizador existente foi adaptada para integrar a nova funcionalidade.

Esta tese contribui para o campo da gestão de dados empresariais ao demonstrar como os Modelos Virtuais podem servir como uma ponte para implementar mecanismos seguros de pesquisa em SQL em sistemas relacionais complexos.

Palavras Chave

Modelo Virtual; Modelo Relacional; Mapeamento de Modelos de Dados; Tradução de Queries; Otimização de Queries.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

API	Application Programming Interface
CTE	Common Table Expression
ID	Identifier
JPA	Java Persistence API
OID	Object Identifier
OOSQL	Object-Oriented Structured Query Language
ORM	Object-Relational Mapping
OSQL	Object-Structured Query Language
SQL	Structured Query Language
XML	Extensible Markup Language
UI	User Interface

1 Introduction

Contents

Atlas is an enterprise architecture tool developed by **Link Consulting** to support organizational transformation by providing structured visualization models and dashboards of a company's architecture and artifacts [1], offering a structured representation of business processes to assist decision-making.

To facilitate data retrieval for analysis, Atlas includes the **Query Designer**, an integrated query editor that allows users to construct **graphical queries** using a predefined syntax. While effective in retrieving data, this system has limitations in performance and flexibility, as it presents inefficient execution that depends on Extensible Markup Language (XML) interpretation, and requires extensive manual interaction, making complex queries time-consuming to build and modify.

This thesis addresses these limitations by introducing Structured Query Language (SQL)-based queries into Atlas, allowing users to input SQL queries while maintaining the performance expectations of the Atlas platform. The following sections outline the **motivation** behind this work, the **work objectives**, the **main contributions**, and the overall **structure** of this thesis.

1.1 Motivation

Integrating SQL-based query input into Atlas presents a challenge due to the structure of its relational model. Atlas' current relational schema is designed to store enterprise architecture data in generalized tables, which is not suitable nor secure as a direct base for user-written SQL queries.

To bridge this gap, this work introduces a **virtual schema** that allows users to interact with an intuitive structure, without manually accounting for the complexities of the underlying database. Moreover, this approach focuses on translating user-written SQL queries into a format compatible with Atlas' relational model.

There is limited research on integrating virtual schemas into existing relational models for enterprise architecture tools, leaving a significant margin for improvement in terms of efficiency, usability and security. This thesis contributes by providing insights on designing and implementing a virtual schema that supports SQL-based querying in Atlas.

1.2 Work Objectives

The aim of this thesis is to define a virtual schema to integrate intuitive SQL-based query input into Atlas, abstracting the complexities of the underlying relational model. To achieve this, we explore adapting an SQL **parser** to validate and process input queries, ensuring they conform to Atlas-specific constraints, and translating them into executable statements that align with Atlas' relational schema.

Additionally, to eliminate redundant parsing, this work delves into introducing a **new column** in the relational schema to store pre-translated SQL queries, significantly improving query saving times and allowing support for nested query execution.

1.3 Main Contributions

The key contributions of this work include:

– Designing a virtual schema for executing intuitive SQL queries over a complex relational model;

– Defining accurate query mappings to translate various types of user-written SQL queries;

– Impacting query execution efficiency through techniques that eliminate redundant query parsing.

– Implementing support for query execution within Atlas' scenarios.

– Performance evaluation through practical use cases, testing query accuracy and efficiency improvements compared to the existing graphical query system.

1.4 Organization of the Thesis

In addition to this introduction, the document is divided into six sections. Chapter 2 provides background knowledge on the Atlas platform, along with the related work; Chapter 3 presents the problem statement, a brief discussion on the solution, and the adaptations made to the SQL parser; Chapter 4 introduces the virtual schema and explains the structure of the SQL-based queries, showcasing the different types of queries supported; Chapter 5 focuses on the query transformation and execution processes and the user interface integration; In Chapter 6 we evaluate the implemented solution by comparing it with the current system; Finally, Chapter 7 presents a discussion on this thesis, summarizing the main conclusions and outlining possible directions for future work.

2 Concepts and Related Work

Contents

2.1 Background and Key Concepts on the Atlas Platform

In this section, we provide an overview of the **Atlas** platform, focusing on its role as an enterprise architecture tool. Additionally, we discuss its intricate **relational data model**, which manages the complexities of Atlas' data, and the unique **graphic querying system**, a query editor that allows users to design graphic queries to retrieve and manipulate data.

2.1.1 Atlas as an Enterprise Architecture Tool

Atlas is an **enterprise architecture tool** [1] designed by Link Consulting to simplify the management of complex organizational ecosystems, providing an integrated view of the various layers and elements within an organization.

Inspired by the ArchiMate language, Atlas builds upon this foundation while allowing the introduction of new classes and relationships that may not exist in ArchiMate but better suit specific cases, offering a more adaptable approach to capturing the complexity of organizational structures.

At the core of Atlas is a robust **metamodel** that defines how data is structured and interconnected. Atlas allows the definition of different metamodels by specifying **classes** and how they relate to each other through **properties** and types of **relations**.

Each **class** in the metamodel represents essential elements such as Business Processes, Application Components, and Technologies, along with their connections. These classes act as templates, enabling the creation of **objects** with specific attributes. Each object is an instance of its class, inheriting its predefined characteristics.

An **attribute** in Atlas consists of two components:

1. A **property**: Defines the name of the connection, linking an origin object to a destination object. Its name begins with an uppercase letter. For example, *Owner*.
2. A **relation**: Specifies the type of relationship from one object to another, including both the direction from the origin to the destination and its inverse from the destination back to the origin. Its name begins with a lowercase letter. For example, *owns* and *owned by*.

Objects can be associated with one another through properties or relations, enabling users to navigate and analyze complex object dependencies. These associations are particularly useful for exploring organizational structures and displaying data in predefined **blueprints**, views made in the platform that provide clear visualizations of the modeled architecture.

To facilitate data retrieval and exploration, Atlas includes a query designer platform that enables users to design **graphic queries**. These queries allow for specific searches within a repository, filtering and extracting relevant information to display in views.

Each Atlas **repository** contains a distinct set of architectural elements, including classes, blueprints, and queries. These repositories are isolated from one another, ensuring that each repository can reflect the specific characteristics and requirements of its architecture.

Atlas also supports the use of **scenarios**, which provide an isolated, parallel view of the main repository. Scenarios enable designers to modify objects without affecting the main repository, offering a secure environment for testing and experimentation.

2.1.2 Atlas' Relational Data Model

Atlas uses a **relational model** to manage its data, providing a robust framework for handling the complex data structures typical of organizational architectures. This model enables dynamic management of relationships between data entities.

The relational schema is composed of multiple interconnected tables, each serving a distinct role within the system. Below is an overview of the relevant tables and attributes for our work:

- **t_repository**: This table tracks all the repositories in Atlas.
  - Relevant attribute: **id** - Unique identifier for each repository.
- **t_class**: Stores all class definitions in the system.

  - Relevant Attributes: **id** - Unique identifier for each class.
  - **repository_id** - Associates the class with its repository.
- **t_scenario**: Contains all scenario definitions in the system.
  - Relevant Attribute: **id** - Unique identifier for each scenario.
- **t_object**: Stores all objects in Atlas.
  - Relevant Attributes: **id** - Unique identifier for each object.
  - **class_id** - Links each object to its corresponding class.
- **t_object_properties**: Tracks the properties of objects.
  - Relevant Attributes: **object_id** - Links the properties to the corresponding object.
  - **config_container_id** - Associates the object's properties with a repository_id or a scenario_id, depending on whether the properties were edited in a repository or in a scenario.
- **t_property**: Catalogs all property definitions in the system.
  - Relevant Attribute: **id** - Unique identifier for each property.
- **t_base_type**: Represents all relation definitions in Atlas.
  - Relevant Attribute: **id** - Unique identifier for each relation.
- **t_object_relation**: Manages connections between objects.
  - Relevant Attributes: **origin_id** - Identifier of the object initiating the connection.
  - **destination_id** - Identifier of the object being connected to.
  - **property_id** - Links the connection to the corresponding property in t_property.
  - **relationtype_id** - Links the connection to the corresponding relation in t_base_type.
  - **config_container_id** - Associates the relation with a repository_id or a scenario_id, depending on whether the connection was created in a repository or in a scenario.
- **t_query_type**: Stores queries designed within Atlas.
  - Relevant Attributes: **id** - Unique identifier for each query.
  - **repository_id** - Links the query to its repository.
  - **implementation** - Contains the graphic query representation in XML.

The Entity-Relationship Diagram in Figure 2.1 illustrates Atlas' relational schema, highlighting the key tables and relationships.

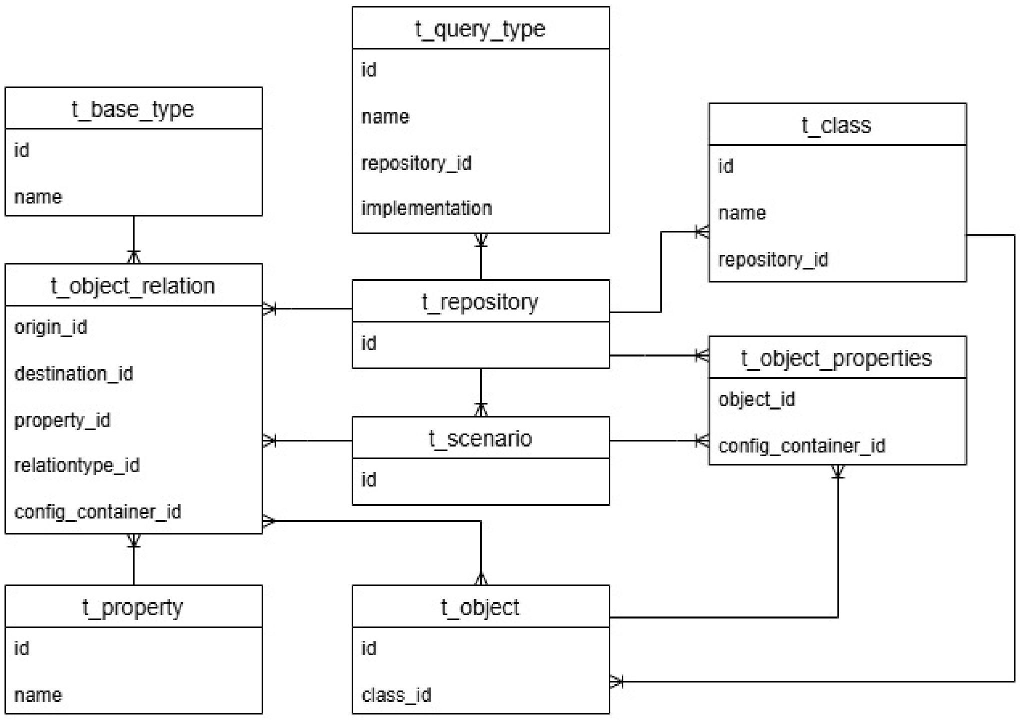


Figure 2.1: Entity-Relationship Diagram of Atlas' Relational Schema.

2.1.3 Atlas' Graphic Querying System

The Atlas platform features a **graphical querying system** named **Query Designer** that allows users to design and execute graphic queries to retrieve and manipulate data. This system provides an intuitive interface for non-technical users, using predefined symbols and a drag-and-drop canvas to simplify the query building process.

2.1.3.A Capabilities of the Graphic Query Language

Atlas' graphic query language supports a range of functionalities that enable users to model and retrieve data intuitively. These capabilities include:

– **Filtering**: Define conditions to select a subset of objects from an initial dataset based on specified criteria.

– **Navigation**: Transition from one set of objects to another. A named navigation selects objects referenced by the specific property or relation, while an unnamed navigation simply passes the current objects to the next step in the query flow.

– **Updating**: Update objects by assigning new values to their properties or relations.

– **Merging**: Combine multiple sets of objects into a unified set through a union operation.

The Query Designer canvas is equipped with a set of intuitive symbols that users can use to build queries. Table 2.1 illustrates these symbols and their corresponding functionalities.

Icon	Name	Description
○	**Start**	Represents the initial set of objects handled by the query.
→	**Flow**	Directs the objects. If named, it selects the objects referred by the attribute.
⊟	**Filter**	Defines a where condition to filter the set of objects.
▭	**Set**	Updates a property or relation for all objects in the set.
⊂⊃	**Merge**	Combines multiple sets of objects into one using a union operation.
◫	**Repeater**	Repeats a query template within the query for iterative operations.
◎	**Finish**	Marks the end of the query.

Table 2.1: Atlas' Query Language Visual Representation.

A graphic query starts with the **Start** symbol, where the initial set of objects is defined by specifying a class. As the workflow progresses, users can apply symbols to perform actions such as filtering, navigation, updates, and merging. The query concludes with the **Finish** symbol, where the final result set is defined.

To demonstrate the syntax and workflow, the Figure 2.2 illustrates a simple query designed in the Graphic Query Language, considering a repository with a **class** named *Business Actor*, that has a **property** named *Location*.

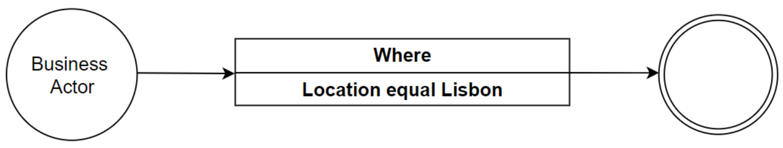


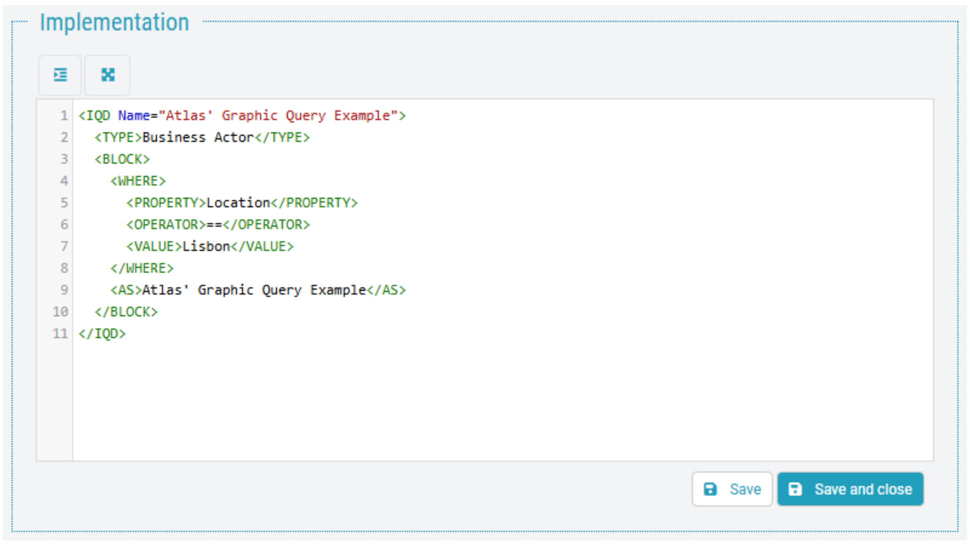
Figure 2.2: Atlas' Graphic Query.

The query retrieves the objects from the class that have the **property** *Location* equal to *Lisbon*.

While this example showcases a basic query, Atlas' graphic queries can be significantly more complex, incorporating multiple functionalities. More advanced examples will be provided in the following chapters.

2.1.3.B Graphic Query Execution Process

When a graphic query is designed and saved in Atlas, the system translates it into an XML representation, which is then stored in the **implementation** column of the **t_query_type** table. This translation step converts visually structured queries into a format interpretable by the backend. In Figure 2.3 is the XML translation corresponding to the graphic query previously presented in Figure 2.2:



```xml
<IQD Name="Atlas' Graphic Query Example">
  <TYPE>Business Actor</TYPE>
  <BLOCK>
    <WHERE>
      <PROPERTY>Location</PROPERTY>
      <OPERATOR>==</OPERATOR>
      <VALUE>Lisbon</VALUE>
    </WHERE>
    <AS>Atlas' Graphic Query Example</AS>
  </BLOCK>
</IQD>
```

Figure 2.3: Graphic Query XML Representation.

When executing a graphic query, the system retrieves the corresponding XML code from the database and processes it into executable statements, through the Worker module. This step involves interpreting the XML representation, adding inefficiency and noticeable waiting times to the query execution process. Thereafter, the Worker executes the derived statements against the relational database, retrieves the requested data and prepares it for presentation on the platform.

2.2 Related Work

This section reviews existing research in two significant areas: schema transformation from object-oriented to relational databases and query translation between object-oriented and relational models. The approaches discussed involve schema mapping techniques, use of reference and inheritance tables, and path expression interpretation.

2.2.1 Schema Transformation from Object-Oriented to Relational Models

Hsieh et al. propose an approach for transforming an object-oriented database schema into a relational schema [2], consisting on the following mapping steps:

Mapping classes to relations:

1. Every class in the object-oriented model is mapped to a corresponding table in the relational model.
2. The Object Identifier (OID) from the object-oriented model serve as primary keys in the relational tables.
3. Class instances map directly to rows in their corresponding tables.

Mapping non-reference attributes:

- Non-reference, single-value attributes become direct columns in the relational tables.
- Non-reference, set attributes are represented through virtual relations that consist of the class OID and the attribute itself, collectively serving as the primary key of the new virtual relation.

Naming relations and attributes:

- Non-reference, single-value attributes have the same name in the relational table as in their original class.
  - For instance, a class *Person* with an attribute *name* translates to a column named *name* in the *Person* relational table.
- Reference, single-value attributes are named by combining the attribute name with the primary key of the referenced class.
  - For example, if *Person* has a reference to the class *Department* which has a primary key *DepartmentOid*, then the relational table *Person* will have a column *departmentDepartmentOid*.
- Virtual relations for non-reference set attributes are named by concatenating the enclosing class name with the attribute name.
  - For example, if *Person* has multiple phone numbers, a virtual table *PersonPhoneNumber* is created with a combined primary key (*PersonOid*, *PhoneNumber*).
- Virtual relations for reference set attributes follow a similar naming pattern, combining the class name with the referenced attribute and using a combined primary key from the related classes.
  - For instance, if *Person* has multiple references to the class *Project*, which has a primary key *ProjectOid*, then a virtual relation *PersonProject* with primary keys (*PersonOid*, *ProjectOid*) is created.

2.2.2 Query Translation Between Object-Oriented and Relational Models

2.2.2.A Using Relationship and Inheritance Tables

Huang et al. build upon the approach presented by Hsieh [2] to translate Object-Oriented Structured Query Language (OOSQL) queries into relational SQL queries [3]. Their solution introduces two reference structures: a Relationship Reference Table and an Inheritance Reference Table. The translation process involves:

1. Analyzing the OOSQL query's reference paths defined by the syntax pattern *C.r1.r2..rn.attr*, where *C* represents the root class, *r1..rn* are reference attributes, and *attr* is the final attribute.
2. Creating the Relationship and Inheritance tables to identify references between entities.

3. Analyzing the reference path specified in the WHERE clause of the OOSQL query.

4. Identifying the root class of the reference path to assign to the FROM clause of the relational SQL query. If an attribute in the reference path belongs to a different class, the translation process involves consulting the respective reference table to form the correct WHERE clause predicate, linking foreign keys and primary keys appropriately.

Below is an example illustrating the process of converting an OOSQL query into relational SQL. This query retrieves the names of all drugs where the order number is *123*. To support this, the Entity-Relationship Diagram of the relational schema is presented in Figure 2.4, along with the Relationship Reference Table (2.2), which serves as the reference table corresponding to the relation in case, as defined in the object-oriented model.

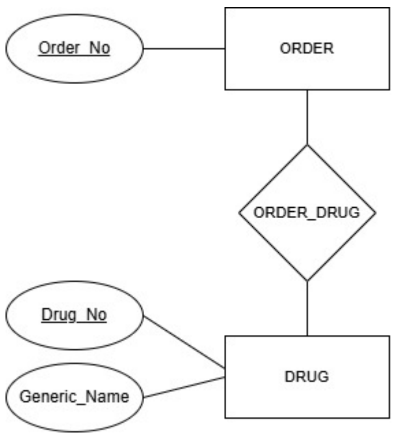


Figure 2.4: Entity-Relationship Diagram Representing the Order-Drug Relation.

Referencing Relation	Relationship Attribute	Referenced Relation	Reference Attributes
DRUG	order_code	ORDER	Drug_No, Order_No

Table 2.2: Relationship Reference Table for the Order-Drug Relation.

Original OOSQL query:

```
SELECT DRUG.Generic_Name
FROM DRUG
WHERE DRUG.order_code.Order_No = 123;
```

Translated relational SQL query:

```
SELECT DRUG.Generic_Name
FROM DRUG, ORDER_DRUG, ORDER
WHERE DRUG.Drug_No = ORDER_DRUG.Drug_No
AND ORDER_DRUG.Order_No = ORDER.Order_No
AND ORDER.Order_No = 123;
```

2.2.2.B Path Expression Interpretation

Fong and Cheung propose an approach centered on interpreting and reconstructing Object-Structured Query Language (OSQL) queries into relational SQL queries [4]. Their method involves two phases:

Interpretation phase:

1. Break the query into SELECT, FROM, and WHERE clauses.
2. Traverse the object-oriented paths to locate attributes for the SELECT clause, potentially creating temporary tables for multi-valued attributes.
3. Determine tables for the FROM clause based on identified attributes.
4. Analyze conditions for the WHERE clause, identifying involved tables and necessary join operations.

SQL query construction phase:

1. Extract and format attributes individually from the SELECT clause.
2. Store involved tables and attribute names in temporary variables.
3. Construct the relational SQL query using these temporary variables for the SELECT and FROM clauses.
4. Decompose the WHERE clause conditions, applying join operations accordingly.

Below is an example demonstrating the conversion of an OSQL query into relational SQL. This query retrieves the names and Identifier (ID) numbers of staff members, as well as the name of the department with number equal to *1234*. To support this, the Entity-Relationship Diagram of the relational schema is provided in Figure 2.5, along with the Attributes Table (2.3), which details the classes involved in the object-oriented model.

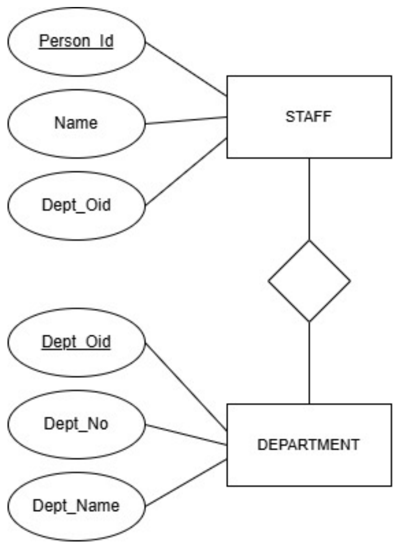


Figure 2.5: Entity-Relationship Diagram Representing the Staff-Department Relation.

STAFF		DEPARTMENT	
Attributes	**Data Type**	**Attributes**	**Data Type**
Person_Id	integer	Dept_No	integer
Name	varchar	Dept_Name	varchar
Dept	DEPARTMENT		

Table 2.3: Attributes Table for the Staff and Department Classes.

Original OSQL query:

```
SELECT STAFF.Name, STAFF.Person_Id, STAFF.Dept.Dept_Name

FROM STAFF

WHERE STAFF.Dept.Dept_No = 1234;
```

Translated relational SQL query:

```
SELECT STAFF.Name, STAFF.Person_Id, DEPARTMENT.Dept_Name

FROM STAFF, DEPARTMENT

WHERE STAFF.Dept_Oid = DEPARTMENT.Dept_Oid AND

DEPARTMENT.Dept_No = 1234;
```

3 SQL-Based Input Queries in Atlas

Contents

This chapter begins by presenting **the Problem**, addressing the challenges that motivated this work. Following this, we provide an overview of **the Solution**, together with a workflow snippet illustrating the designed transformation process of user-written queries. The chapter concludes with a focus on the SQL **parser's** characteristics and the specific modifications implemented to ensure secure and accurate query parsing, validation, translation, and execution.

3.1 Overview of the Problem

The Atlas platform relies on a graphic querying system that, while user-friendly for non-technical users, has limitations on performance. Its execution is inefficient, as it depends on interpreting the XML representation and transforming it into executable statements, at runtime. This process also relies on the Worker module, which introduces additional overhead. Moreover, users familiar with SQL often find it time-consuming to model queries graphically as it requires extensive interaction through a drag-and-drop canvas. To address these limitations, this work introduces a feature that allows users to define complex queries more efficiently with intuitive syntax, while maintaining the performance expectations of the Atlas platform.

However, implementing this feature presents a challenge. The platform's relational model, which has abstract and generalized tables, although serves as the foundation for modeling and executing graphic queries, it does not align with an intuitive or safe query format for user-written SQL queries, creating a barrier to integrating SQL query input into the platform.

The problem, therefore, is twofold:

1. Atlas' current graphic querying system is inefficient for modeling and executing queries.
2. Developing a feature for intuitive SQL query input requires addressing the **relational model**, which is abstract and does not align with an easy query format for users.

3.2 Overview of the Solution

To address these challenges, a solution comprising the following key components was developed:

1. Designing a **virtual schema** that allows users to interact with a structure resembling standard tables and columns, providing an intuitive format for user-written SQL queries.

   With the virtual schema, users can reference Atlas' **classes** as tables and Atlas' **attributes** as columns. For example, a query to retrieve objects connected to the class *Business Actor* through the attribute *Job Title* would be written as:

   ```
   SELECT "Job Title" FROM "Business Actor";
   ```

   In reality, no such table or column exists in the Atlas relational schema. Classes like *Business Actor* are stored as instances in the *t_class* table, while properties like *Job Title* are stored as entries in the *t_property* table.

2. Using an existing **SQL parser** and extending it to handle user-written SQL queries. The parser is the base for performing the following tasks:

   (a) Interpreting and validating the syntax of input queries referencing the virtual schema, ensuring compliance with SQL standards and Atlas-specific requirements.

   (b) Transforming the user-written SQL query into its translated version that conforms to the relational schema, by mapping references to virtual tables and columns into their corresponding entries in the relational schema.

   (c) Incorporating new functionalities into SQL queries to extend the current querying capabilities of graphic queries:

      i. **Nested queries**, allowing users to reference the result set of one query within another.

ii. **Scenario verification**, so query results consider not only objects from the main repository but also those from loaded scenarios.

With this implemented model, queries submitted in Atlas undergo a transformation process to ensure accurate execution, as summarized below:

```
1. User submits query through Atlas UI:
    1.2 System receives query code and analyzes it:
        - if a Graphic Query, invoke traditional execution mechanism.
        - if User-Written SQL, invoke SQL Parser.
    1.3. If SQL Parser is invoked:
        - parse User-Written SQL.
        - perform lexical and syntax validation.
        - if syntax is incorrect, return error message to the user.
    1.4. If syntax is valid, interpret clauses using Java classes:
        - translate clauses into the Translated SQL Query.
    1.5. Save the Translated SQL Query in the database.
2. User requests to execute User-Written SQL through Atlas UI:
    2.1. Retrieve the Translated SQL Query from the database.
    2.2. Check user's active working context:
        - if a Repository, maintain the Translated SQL Query as is.
        - if a Scenario, reformulate the Translated SQL Query to include
            scenario data.
    2.2. Run the Outputed Translated Query against the database.
    2.3. Format results.
    2.4. Display results through Atlas UI.
```

3.3 Adapting JSqlParser for Query Validation

Integrating SQL queries into the Atlas product requires a robust mechanism for validating user-submitted queries, which involves **lexical analysis**, that breaks the query into tokens, and **syntax analysis**, which ensures the query adheres to standard SQL grammar as well as custom rules introduced for specific needs of the Atlas platform.

To achieve this, the **JSqlParser framework** [5] was adapted. JSqlParser, built using JavaCC, parses SQL statements and translates them into a hierarchical structure of Java classes [6], allowing inspection, validation, and manipulation of query components.

To meet Atlas' requirements, the following adjustments were introduced to restrict user input to a specific query format:

- Since queries in Atlas are used exclusively to retrieve data to display, the JSqlParser was adapted to **only accept SELECT statements**, which simplifies validation and disallows data modification through INSERT, UPDATE, or DELETE statements.

- Since the Atlas Platform does not impose restrictions on the names of classes, properties, or queries, which can include spaces, numbers, and special characters, to ensure valid parsing, we introduce the requirement that **all names in SQL queries must be enclosed in quotation marks**. Queries that do not follow this convention are rejected.

A custom Java class **QueryParser** was developed which performs lexical and syntax analysis of queries. This class ensures the correctness of SQL syntax, verifies if queries align with Atlas' constraints, and provides immediate feedback on query validation.

3.3.1 Lexical Analysis

In the QueryParser class, the *CCJSqlParserManager.parse* method provided by JSqlParser is called, which tokenizes the SQL query and parses it into an object implementing the **Statement** interface [7]. During this step, tokens are analyzed for validation, including keywords, like SELECT, FROM and WHERE, identifiers and symbols. Incorrect or unexpected tokens, like invalid characters or missing keywords, result in error messages to the user.

3.3.2 Syntax Analysis

The *CCJSqlParserManager.parse* method also validates the query's structure, checking the ordering of clauses, and correct use of balanced parentheses and operators.

In this stage, the resulting **Statement** object is analyzed. The accepted queries are restricted to those represented by the **PlainSelect** class, which handles solely SELECT statements. The Statement object is cast to the PlainSelect class, and queries that do not conform to this type result in error messages to the user.

The PlainSelect class provides access to the following components of the query:

– **SelectItem** - Contains the **column** specified in the SELECT clause.

– **FromItem** - Contains the **table** specified in the FROM clause.

– **Expression** - Contains the entire **condition** specified in the WHERE clause.

As part of syntax validation, we include the requirement of ensuring that all names are enclosed in quotation marks. The *getSelectItem* and *getFromClause* methods verify that column and table names comply with this rule, and similarly, later in the query translation process, this requirement is enforced for all names within the WHERE clause conditions. Queries missing required quotation marks are rejected with an error message.

The Java code snippet in A.1 demonstrates the use of the JSqlParser for lexical and syntax analysis, performed in our custom **QueryParser** class.

3.3.3 Query Processing

Using the components SelectItem, FromItem, and Expression extracted during validation, the system processes the user-written query and translates it into its translated version to be executed against the database. The Java code snippet in A.2 shows our custom class **QueryHandler**, which collects the tokens and handles the query interpretation and transformation, with the process beginning in the *interpreteQuery* method. Further details about how the class manages the processing are provided in Chapter 5, after a detailed discussion in the next chapter on the structure of user-written queries, to give necessary context to better understand the translations.

4 Expressing SQL Queries Using a Virtual Model

Contents

This chapter introduces a detailed explanation on the implemented **SQL-Based Queries**. It begins by presenting the **virtual schema** that serves as a foundation for the user-written queries. Next, we detail the query **structure**, focusing on key components such as clause restrictions and query patterns. We then introduce two new querying functionalities, **nested queries** and **scenario verification**, which are not available in the graphic querying model. Finally, we present examples that compare queries in the existing graphic query language with their SQL-based equivalents.

Throughout the following chapters, the term **SQL-Based Query** refers to the user-written query that complies with the virtual schema, and **Translated SQL Query** refers to its transformed version that is executed against Atlas' relational schema.

4.1 Designing a Virtual Schema

To implement the new querying functionality in Atlas, a **virtual schema** was designed to serve as the foundation for an intuitive SQL query format and abstract the complexities of the relational model, allowing users to interact with a simplified query structure that resembles standard tables and columns.

Instead of directly referencing the intricate and abstract tables of Atlas' **relational model**, users can rely on a virtual schema that dynamically adapts to the system's state. This schema creates tables and columns based on the classes, queries, properties, and relations currently stored in Atlas, operating under the following assumptions:

1. Each **class** and each **query** saved in Atlas is treated as a **virtual table**. For instance, if a class named *Application Component* or a query named *Employees in Lisbon* exists in Atlas, it is represented as a virtual table in the virtual schema.

2. Each **virtual table** can be preceded by the keyword *"q="* or *"c="*, indicating whether it corresponds to a query or a class, respectively. The final name must be enclosed in quotation marks, for example, *"q=Employees in Lisbon"*.

3. Each **attribute** (**properties** and **relations**) saved in Atlas is treated both as a **virtual column** and as a **virtual table**. For instance, if a property named *Location* or a relation named *owned by* exists in Atlas, it is represented as a virtual column in all tables of the virtual schema and simultaneously functions as an individual virtual table that can be directly queried, with its name enclosed in quotation marks.

Below is Algorithm 4.1, a pseudocode representation of the process of constructing the virtual schema based on the data stored in Atlas, and Figure 4.1, both illustrating how virtual tables are dynamically derived from the stored classes, queries, properties, and relations, while virtual columns are generated from the properties and relations.

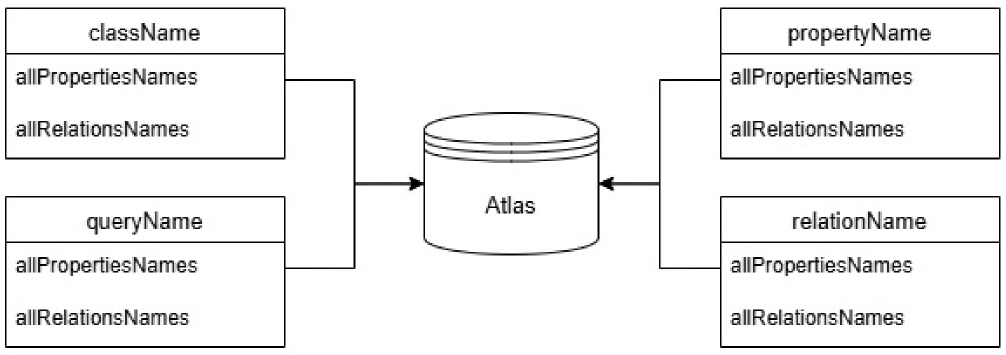


Figure 4.1: Visualization of the Virtual Schema Dynamically Generated from Atlas' Data.

To ensure that the user-written queries are translated into valid queries to interact with Atlas' relational schema, the following mappings are applied during the query translation phase:

- References to **classes** in FROM clauses are translated to queries against the table **t_class**.

Algorithm 4.1: Pseudocode for Constructing the Virtual Schema from Atlas' Data.

```
begin
    // Retrieve object names from Atlas, enclosed in quotation marks, with and
       without the "c=" and "q=" prefixes for classes and queries
    $allClassesNames \longleftarrow$ "'" $+ getClassesFromAtlas() +$ "'"
    $allClassesNames \longleftarrow$ "'c = " $+ getClassesFromAtlas() +$ "'"
    $allQueriesNames \longleftarrow$ "'" $+ getQueriesFromAtlas() +$ "'"
    $allQueriesNames \longleftarrow$ "'q = " $+ getQueriesFromAtlas() +$ "'"
    $allPropertiesNames \longleftarrow$ "'" $+ getPropertiesFromAtlas() +$ "'"
    $allRelationsNames \longleftarrow$ "'" $+ getRelationsFromAtlas() +$ "'"

    // Generate virtual tables for classes
    foreach $className \in allClassesNames$ do
        $tableName \longleftarrow className$
        $columnNames \longleftarrow allPropertiesNames + allRelationsNames$

    // Generate virtual tables for queries
    foreach $queryName \in allQueriesNames$ do
        $tableName \longleftarrow queryName$
        $columnNames \longleftarrow allPropertiesNames + allRelationsNames$

    // Generate virtual tables for properties
    foreach $propertyName \in allPropertiesNames$ do
        $tableName \longleftarrow propertyName$
        $columnNames \longleftarrow allPropertiesNames + allRelationsNames$

    // Generate virtual tables for relations
    foreach $relationName \in allRelationsNames$ do
        $tableName \longleftarrow relationName$
        $columnNames \longleftarrow allPropertiesNames + allRelationsNames$
```

- References to **queries** in FROM clauses are translated to queries against the table **t_query_type**.
- References to **properties** in SELECT and WHERE clauses are translated to queries involving the tables **t_property** and **t_object_relation**.
- References to **relations** in SELECT and WHERE clauses are translated to queries involving the tables **t_base_type** and **t_object_relation**.
- All translated queries can also be adjusted to incorporate checks against the table **t_scenario** to include scenario objects.

4.2 Defining SQL-Based Queries

The implemented SQL-Based Query format follows a well-defined structure, with restrictions in place to ensure that the queries align with Atlas' virtual schema. The components of the query include the SELECT, FROM, and WHERE clauses, each with specific rules and limitations.

4.2.1 SELECT Clause

The implemented system supports only a single SELECT item, which can be used in the following way:

- **An attribute name:** Users can navigate connections within Atlas and retrieve objects linked to the dataset specified in the FROM clause through a particular property or relation. The attribute name must be enclosed in quotation marks. For example:

```
        SELECT "realizes" FROM "System Software";
```

  This query selects the objects connected to the *System Software* dataset through the *realizes* relation. In the virtual schema, *"realizes"* is a column of the *"System Software"* table.

- **Asterisk (*):** The asterisk is used to select the objects defined in the FROM clause themselves. For example:

```
        SELECT * FROM "System Software";
```

  This query selects the objects from the *System Software* dataset.

While the current functionality supports only a single SELECT item, we can implement support for multiple SELECT items in the future, allowing users to retrieve multiple connected attributes.

4.2.2 FROM Clause

The FROM clause accepts either the name of a **class** or the name of another **query**, treating both classes and query result sets as **virtual tables** in the virtual schema. The latter allows the use of nested queries, as the result set of one query can be used as a table in another. We support the use of the prefixes *"c="* for classes and *"q="* for queries, for easy distinction between them. Additionally, the FROM clause supports subqueries as well as UNION statements to combine the results of multiple queries, which must be enclosed in parentheses.

Class and query names must be enclosed in quotation marks. Here are some examples:

```
   SELECT * FROM "c=Business Actor";
```

This query selects the objects from the *Business Actor* class. In the virtual schema, *"c=Business Actor"* is represented as a virtual table.

```
   SELECT * FROM "q=Employees in Lisbon";
```

This query selects the objects from the result set of the *Employees in Lisbon* query. In the virtual schema, *"q=Employees in Lisbon"* is represented as a virtual table.

```
SELECT "Components" FROM (SELECT "realizes" FROM "System Software");
```

This query selects the objects connected to the *System Software* dataset through the *realizes* relation, followed by those connected to the resulting set via the *Components* property. In the virtual schema, *"Components"* is a column of the *"realizes"* table, and *"realizes"* is a column of the *"System Software"* table.

```
SELECT * FROM (SELECT * FROM "Business Actor" UNION SELECT * FROM "System
    Software");
```

This query combines objects from both the *Business Actor* class and the *System Software* class using the UNION operator. In the virtual schema, both classes are represented as virtual tables.

4.2.3 WHERE Clause

The WHERE clause is used to filter results based on attributes, which are treated as **virtual columns**.

Here are some examples of how users can specify conditions:

```
SELECT * FROM "Business Actor" WHERE "Location" = "Lisbon";
```

This query selects the objects from the *Business Actor* class that have the property *Location* equal to *Lisbon*. In the virtual schema, *"Location"* is a column of the *"Business Actor"* table.

```
SELECT * FROM "System Software" WHERE "realizes" = "Integration Framework";
```

This query selects the objects from the *System Software* class that have the relation *realizes* equal to *Integration Framework*. In the virtual schema, *"realizes"* is a column of the *"System Software"* table.

Additionally, the system supports:

- **AND/OR operators** for multiple conditions:

```
SELECT * FROM "Business Actor" WHERE "Location" = "Lisbon" OR "Job
    Title" = "Service Account";
```

- **Null/Not Null filtering** to find objects where a property or relation is either null or not null:

```
SELECT * FROM "Business Actor" WHERE "Location" IS NOT NULL;
```

4.3 Introducing Nested Queries

While the graphic query language provides a visual method for users to define queries, it lacks support for advanced features such as **nested queries**.

Nested queries allow one query to be used as the basis for another query. In our approach, this is achieved by using the result set of a query as a virtual table, which can then be referenced by its name in the FROM clause of the outer query. For instance, a user can define a query that selects a dataset based on a condition and then reference this query to perform further filtering, as shown in the following example:

Inner Query: A user defines a query named *Employees in Lisbon* that retrieves all objects from the class *Business Actor* that have a specific *location*.

```
# Query Name:
 Employees in Lisbon
# Query Implementation:
 SELECT * FROM "c=Business Actor" WHERE "Location" = "Lisbon";
```

Outer Query: The user then uses the result set of *Employees in Lisbon* in another query named *Service Accounts in Lisbon* to further filter the objects by their *job title*.

```
# Query Name:
Service Accounts in Lisbon
# Query Implementation:
SELECT * FROM "q=Employees in Lisbon" WHERE "Job Title" = "Service Account";
```

4.4 Introducing Scenario Verification

Another feature implemented in our system is **scenario verification**, which ensures that queries return data relevant to the user's active working context, by considering currently loaded scenarios.

In Atlas, **scenarios** serve as parallel and isolated views of the main repository. When a scenario is created, it begins as an exact replica of the main repository's data, allowing users to create, modify, or delete objects without affecting the repository itself, providing a safe environment for testing and simulations in a specific context.

Changes made in the repository propagate to its scenarios unless the objects have been edited or deleted within the scenario, in which case the scenario's version of the object takes precedence. On the other hand, changes made in a scenario do not impact the main repository unless explicitly published and merged.

When a user executes an SQL-Based Query, the system first checks whether a scenario is currently active:

- If it is, the query is adapted to include data from that scenario, showing:
  - Objects edited in the scenario that meet the query conditions.
  - Objects from the repository that meet the query conditions and have not been altered in the scenario in a way that would invalidate that match.
- If no scenario is loaded, the query returns solely the objects from the main repository.

4.5 SQL-Based Query Patterns

In this section, we explore how SQL-Based Queries have been introduced as an alternative to the existing graphic query language in the Atlas product, allowing users to choose between the graphical system and constructing queries using SQL. We demonstrate the various user-written query patterns supported, together with their graphical equivalents.

To keep the section concise, we present only one type of example, either using a property or a relation. However, queries involving properties follow the same structure as those using relations, with the difference being the replacement of the relation name with the property name and the relation value with the property value.

4.5.1 Simple Dataset Selection

4.5.1.A Selecting Objects From a Class

The graphic query example in Figure 4.2 selects all objects from the class *Application Component*.

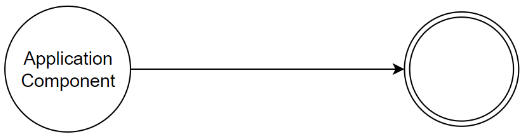


Figure 4.2: Simple Selection in the Graphic Query Language.

The equivalent SQL-Based Query uses the FROM clause to specify the class being queried, and is as follows:

```
SELECT * FROM "c=Application Component";
```

4.5.1.B Selecting the Result Set of Another Query

Since using the result set of a query as incoming objects is not supported in the graphic query language, we provide the designed SQL-Based Query for this purpose. It uses the FROM clause to specify the incoming set, which in this case is the result set of the query *All Applications*, and is as follows:

```
SELECT * FROM "q=All Applications";
```

4.5.2 Attribute Condition Filtering

4.5.2.A Single Relation Filtering

The graphic query example in Figure 4.3 selects all objects from the class *System Software*, from which it is applied a filter to retrieve the objects that have the relation *aggregated by* equal to *.NET Core*.

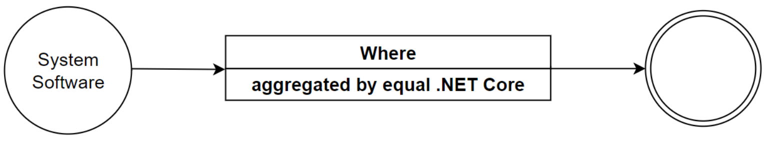


Figure 4.3: Single Attribute Filtering in the Graphic Query Language.

The matching SQL-Based Query uses the FROM clause to specify the class being queried and the WHERE clause to filter the objects based on the specified relation and value, and is as follows:

```
SELECT * FROM "c=System Software" WHERE "aggregated by" = ".NET Core";
```

Analogously, it is possible to select objects from the result set of another **query**. The SQL-Based Query below uses the FROM clause to specify the incoming set, which is the result set of the query *Employees in Lisbon*, and the WHERE clause to filter the objects based on having the property *Job Title* equal to *Manager*. The query is as follows:

```
SELECT * FROM "Employees in Lisbon" WHERE "Job Title" = "Manager";
```

4.5.2.B Multiple Condition Filtering

The graphic query example in Figure 4.4 applies multiple filters to the incoming objects, combined with AND and OR operators. It selects all objects from the class *Business Actor*, and filters those that have the property *Job Title* equal to *Manager* or equal to *Director*, while having the relation *located at* equal to *Lisbon*.

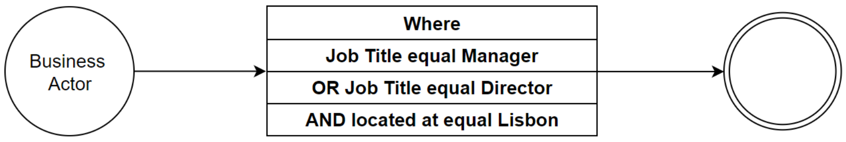


Figure 4.4: Multiple Attribute Filtering in the Graphic Query Language.

The equivalent SQL-Based Query uses the FROM clause to specify the class being queried and the WHERE clause to filter the objects based on the specified property and relation values, with appropriate parentheses to override rules of precedence. The query is as follows:

```
SELECT * FROM "Business Actor" WHERE ("Job Title" = "Manager" OR "Job Title"
    = "Director") AND "located at" = "Lisbon";
```

In our implemented system, we have the advantage of using parentheses to explicitly define operator precedence when combining conditions, which is not available in the graphic language, where operator precedence is fixed and the AND operator takes precedence over OR.

4.5.2.C Null/ Not Null Attribute Filtering

It is possible to filter a set of objects based on whether they do or do not have any objects connected through a specific property or relation.

The graphic query example in Figure 4.5 selects all objects from the class *Application Component* that have the relation *released by* not empty.

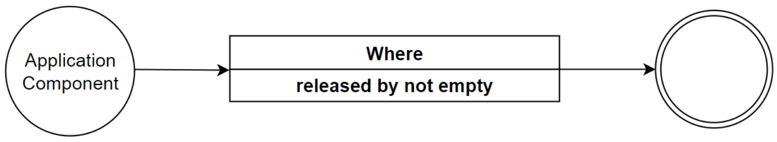


Figure 4.5: Null Attribute Filtering in the Graphic Query Language.

The equivalent SQL-Based Query uses the FROM clause to specify the class being queried and the WHERE clause to filter the objects based on having the specified relation not null, and is as follows:

```
SELECT * FROM "Application Component" WHERE "released by" IS NOT NULL;
```

Similarly, when the intention is to select objects that do not have any objects connected through a specific property or relation, the same methodology is followed but using IS NULL instead.

4.5.3 Attribute Connection Selection

It is possible to have an incoming set of objects and navigate to another set of objects connected to them through a connection.

The graphic query example in Figure 4.6 selects all objects from the class *Application Component*, and navigates the property *Components* to retrieve the connected objects.

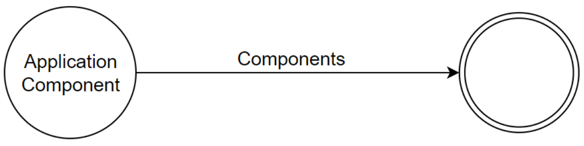


Figure 4.6: Selecting Connected Objects in the Graphic Query Language.

The equivalent SQL-Based Query uses the SELECT clause to retrieve the objects connected to the incoming set through the specified property and the FROM clause to specify the class being queried, and is as follows:

```
SELECT "Components" FROM "Application Component";
```

4.5.4 Attribute Selection With Filtering

It is possible to combine the concepts from the previous sections and consider a query that adds filtering after navigating a connection between objects.

Since using the result set of a query as incoming objects is not supported in the graphic query language, we provide the designed SQL-Based Query for this purpose. It uses the SELECT clause to retrieve the objects connected through the *contracted by* relation, the FROM clause to specify the incoming set *Employees in Lisbon*, and the WHERE clause to filter the connected objects based on having the property *Name* equal to *Confidentiality Agreement*. The query is as follows:

```
SELECT "contracted by" FROM "Employees in Lisbon" WHERE "Name" = "
    Confidentiality Agreement";
```

When an SQL-Based Query includes a WHERE clause, the conditions are always applied to the incoming set of objects, referenced in the FROM clause, unless a connection navigation is specified in the SELECT clause. In such cases, the connection takes precedence, and the filter is applied to the resulting set of connected objects.

If the intention is to first apply a filter to the incoming set of objects and then navigate through a specified connection, it is necessary to use a **subquery** within the FROM clause [8] to prioritize the filtering in the inner query. In the example below, the filter is applied in the inner query, and the connection is specified in the outer query:

```
SELECT "contracted by" FROM (SELECT * FROM "Employees in Lisbon" WHERE "Job
    Title" = "Business Analytic");
```

4.5.5 Union Operation

The graphic query example in Figure 4.7 has two paths and uses a UNION operator to merge them. The upper path selects all objects from the class *Business Process*, navigating through the *realizes* relation to retrieve the connected objects. The lower path selects all objects from the class *Business Service*, applying a filter to retrieve objects where the property *User* is equal to *Link Consulting*. The resulting sets are then combined using a UNION symbol.

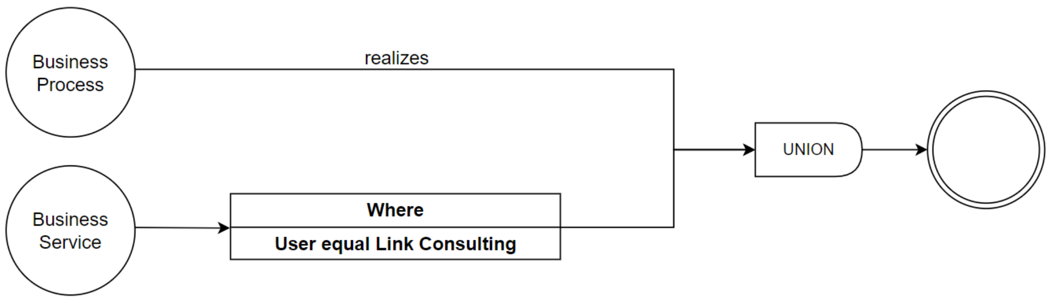


Figure 4.7: Union Operator in the Graphic Query Language.

The equivalent SQL-Based Query uses a subquery in the FROM clause, specifying the two queries being combined. In the inner query, we use a UNION operator to combine the objects connected to the class *Business Process* through the *realizes* relation with the objects from the class *Business Service* that have the property *User* equal to *Link Consulting*. The query is as follows:

```
SELECT * FROM (SELECT "realizes" FROM "c=Business Process" UNION SELECT *
    FROM "c=Business Service" WHERE "User" = "Link Consulting");
```

4.5.6 Subqueries

The graphic query example in Figure 4.8 selects all objects from the class *Application Component*, first navigating through the *Database Schema* property to retrieve connected objects, and then using the result set to navigate through the *Technology Usage* property to retrieve the final connected objects.

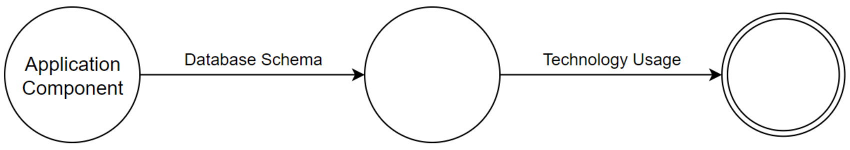


Figure 4.8: Subqueries in the Graphic Query Language.

Each navigation step in the graphic model corresponds to a SELECT clause in the SQL-Based Query, resulting in a nested query structure. The concept of subqueries in SQL allows for these actions to be performed sequentially, where the result of one query becomes the input for the next [8]. In an SQL-Based Query, subqueries follow the principle that the incoming set of objects is defined in the FROM clause of the innermost query.

The example below illustrates the use of subqueries in our model. The outer query's SELECT clause specifies the second action - which is retrieving objects connected through the *Technology Usage* property - while the FROM clause contains the inner query. In the inner query, the FROM clause specifies the incoming objects, the class *Application Component*, and the SELECT clause defines the first action - retrieving objects connected through the *Database Schema* property.

```
SELECT "Technology Usage" FROM (SELECT "Database Schema" FROM "Application
    Component");
```

Following this structure, the inner query executes first, retrieving the objects connected to the *Application Component* class through the *Database Schema* property, and the main query uses that result to further retrieve the objects connected to them through the *Technology Usage* property.

4.5.7 Combining Query Capabilities

It is possible to combine the concepts introduced in the previous sections to construct a query that leverages multiple capabilities.

The graphic query example in Figure 4.9 illustrates the complexity that can be found in realistic Atlas queries.

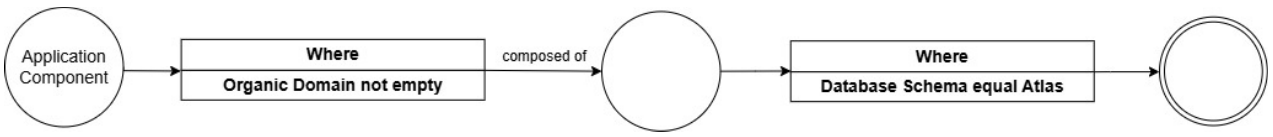


Figure 4.9: Combining Functionalities in the Graphic Query Language.

The equivalent SQL-Based Query is as follows:

```
SELECT "composed of" FROM (SELECT * FROM "Application Component" WHERE "
    Organic Domain" IS NOT NULL) WHERE "Database Schema" = "Atlas";
```

5 Transforming SQL Queries for Relational Execution

Contents

This chapter begins by introducing a **new column** in the *t_query_type* table of the relational schema and explaining how it optimizes the execution of **nested queries**. Next, we demonstrate how user-written queries are mapped to Atlas' existing relational schema, providing explanations and examples of the translation process from **SQL-Based Queries** to **Translated SQL Queries**. Following this, we present the **system architecture**, outlining the technologies behind our implementation, and then discuss the security measures implemented to protect against **SQL injection**. Finally, we present the **user interface** adapted for the new feature, covering the entire lifecycle of SQL-Based Queries, including query creation, editing, execution, and visualizing the results.

5.1 New Relational Column for Query Translations

In Atlas, queries created through the graphical system are translated into an XML query language, which is stored in the *implementation* column of the *t_query_type* table. To support SQL-Based Queries and improve query execution efficiency, we introduce a new column in the same table named **output_query**, designed to store the **Translated SQL Query**.

With this addition, when a user-written query is saved, the system translates it into a Translated SQL Query and stores it in the *output_query* column, so that when the user executes the query, the pre-translated query is retrieved and run, instead of having to re-parse and re-translate the input query. Although this approach increases query saving time, it significantly reduces query generation time, which is done more frequently and has greater impact on user experience.

Additionally, when translating a query that contains a nested query, the system does not translate the entire query from scratch. Instead, it retrieves the pre-translated query of the inner query from the *output_query* column, wraps it in a Common Table Expression (CTE), and then proceeds with translating the outer query, integrating the CTE into its final structure.

Below is a pseudocode example illustrating how our system handles query creation and query execution:

1. A user creates a query named *inner query* that retrieves data from a specified class, represented as <CLASS_NAME>, based on a condition, <CONDITION_1>.

```
# User-Written Query 1
SELECT * FROM "<CLASS_NAME>" WHERE <CONDITION_1>;
```

2. This query is automatically translated into its Translated SQL Query, represented as <TRANSLATED_INNER_QUERY>, which is stored in the *output_query* column of the *t_query_type* table.

```
# Query 1 Translation
<TRANSLATED_INNER_QUERY>;
```

3. A user executes the same query *inner query*. Its pre-translated query, <TRANSLATED_INNER_QUERY>, is retrieved from the *output_query* column of the *t_query_type* table to be executed.

4. A user creates another query that references the previously created *inner query* and applies additional filtering, represented as <CONDITION_2>.

```
# User-Written Query 2
SELECT * FROM "inner query" WHERE <CONDITION_2>;
```

5. During query translation, the system retrieves the pre-translated query of the *inner query*, wraps it in a CTE, and integrates it into the translation of the current query. The final translation is stored in the *output_query* column of the *t_query_type* table, for query execution.

```
# Query 2 Translation
WITH CTE AS (<TRANSLATED_INNER_QUERY>)
SELECT CTE.* FROM CTE WHERE <CURRENT_QUERY_TRANSLATION>;
```

5.2 Mapping SQL-Based Queries to a Relational Schema

In the query validation phase, explained in Chapter 3, the parser tokenizes the components of the user-written query, to ensure that they conform to the expected syntax before proceeding to query translation. The translation process is managed by the custom Java classes: **QueryHandler** and **QueryBuilder**. The **QueryHandler** class, introduced in Chapter 3, is responsible for interpreting the FROM clause and constructing or retrieving the query to be wrapped in the CTE, and the **QueryBuilder** class processes the SELECT and WHERE clauses and combines the CTE and all transformations to produce the Translated SQL Query.

5.2.1 Processing the FROM Clause

The interpretation process for the input query begins in the *interpreteQuery* method of the **QueryHandler** class, which validates the **FROM** clause to determine whether the referenced entity is a valid class or query in Atlas or a UNION statement or a subquery:

- If the entity in the FROM clause has a *"c="* prefix or the name matches a **class**, the *interpreteClassName* method is called to retrieve the ID of the specified class using its name and the *repository_id*, which is automatically determined from the query context. The method then uses the *class_id* to generate a query that retrieves all objects belonging to the specified class.

- If the entity in the FROM clause has a *"q="* prefix or the name matches a **query**, the *interpreteQueryName* method is called to retrieve the ID of the specified query using its name and the *repository_id*. The method then uses the ID to retrieve the inner query's pre-translated SQL query from the *output_query* column of the *t_query_type* table.

- If the FROM clause contains a **UNION statement** enclosed in parentheses, the system separates each subquery and recursively invokes the *interpreteQuery* method for each one. Each transformation is wrapped in a CTE and merged using a UNION operator.

- If the FROM clause contains a **subquery** enclosed in parentheses, the system retrieves it and invokes the *interpreteQuery* method to generate its translation.

If the entity does not match a class or a query and is not a UNION statement or a subquery, the query is deemed invalid and an error message is displayed in the Atlas interface.

Once the FROM clause is successfully interpreted, its transformation is wrapped in a final CTE to serve as the base structure for the Translated SQL Query.

5.2.2 Processing the SELECT and WHERE Clauses

After processing the FROM clause, the system invokes the *build* method of the **QueryBuilder** class, which constructs the Translated SQL Query after processing additional query components. In this step, the *handleSelectClause* method is called to handle the attribute in the **SELECT** clause, and the *handleWhereClause* method is responsible of analyzing the attributes and corresponding values that compose the filtering condition as well as the relational operations in the **WHERE** clause.

5.2.2.A HandleSelectClause method

The *handleSelectClause* method identifies whether the attribute specified in the **SELECT** clause corresponds to a property or a relation. If the attribute is a **property**, i.e. starts with an uppercase letter, the system translates the query using the *t_property* and *t_object_relation* tables to navigate to the objects connected to the ones in the base CTE through that property, and if the attribute is a **relation**, i.e. starts with a lowercase letter, the system translates it against the *t_base_type* and *t_object_relation* tables to retrieve the objects connected to the ones in the base CTE through that relation.

If the **SELECT** clause contains an asterisk (*) instead of a specific attribute, the translated query selects all objects from the base CTE.

In the Java code snippet in A.3 is the custom method *handleSelectClause*, which handles the translation of the SELECT clause.

5.2.2.B HandleWhereClause method

The *handleWhereClause* method processes each condition and logical expression in the **WHERE** clause. To achieve this, we implemented two Java stacks, one for logical operators and one for expressions, to analyze each expression sequentially and determine whether they use **=** or **IS NULL** expressions and whether they are connected through **AND** or **OR** operators, in order to correctly organize and combine them.

The system first pushes the entire WHERE clause onto the expression stack, then pops and processes each expression sequentially until the stack is empty, analyzing the type of each expression using the JSqlParser:

- If it is an **AND** or **OR** predicate, the system separates its right and left expressions using JSqlParser's *getRightExpression()* and *getLeftExpression()* methods. The right expression is an elementary expression and is pushed first, and the left expression includes everything to the left and may contain other AND and OR predicates. The corresponding logical operator is pushed into the logical operators stack.

- If it is an **equality** predicate, the system determines whether the referenced attribute is a property or a relation. If it is a property, it is translated against the *t_property* and *t_object_relation* tables, and if it is a relation, it is translated against the *t_base_type* and *t_object_relation* tables. The resulting transformation retrieves the objects where the specified attribute matches the given value, and is combined into the Translated SQL Query using the operator popped from the logical operators stack, that can be an AND or OR.
- If it is a **null** predicate, the system determines if the referenced attribute is a property or a relation and whether the operator is **IS NULL** or **IS NOT NULL**. Based on this analysis, the expression is translated against either the *t_property* or *t_base_type* table to retrieve objects that either have or do not have connections to the specified attribute, and is combined into the Translated SQL Query using the operator popped from the logical operators stack, that can be an AND or OR.

In the Java code snippet in A.4 is the custom *handleWhereClause* and its related methods, which handle the translation of the WHERE clause.

All these transformations are combined with the base CTE previously constructed from the FROM clause, forming the Translated SQL Query to be executed against the Atlas database. Additionally, a new record is saved in the *t_query_type* table, storing the **SQL-Based Query** in the **implementation** column and the **Translated SQL Query** in the **output_query** column.

5.2.3 Incorporating Scenario Verification

5.2.3.A Creating the SQL Query for Scenario Objects

To support **scenario inclusion** in SQL-Based Queries and ensure that the objects relevant to the user's active context are shown, we pay attention to the attribute *config_container_id* from the *t_object _properties* and the *t_object_relation* tables, which links to either a repository or a scenario ID. In Atlas' relational model, filtering objects by having the *config_container_id* equal to a *t_repository* ID, returns the objects that have been edited in the main repository and meet the criteria, while filtering objects with *config_container_id* equal to a *t_scenario* ID, returns the objects that have been modified within the scenario, with changes that match the conditions.

However, in order to show the data from a scenario that meets the criteria, combining these two queries would incorrectly include repository objects that originally met the conditions but were modified in the scenario in a way that no longer match the conditions. To address this, we exclude the objects edited in the repository that have had relevant attributes overridden in the scenario. Our approach for constructing a **SQL Query for Scenario Objects** uses two CTEs:

- **REP**, that retrieves the **objects edited in the repository** that satisfy the query conditions. This corresponds to the query's original Translated SQL Query.

- **SCEN**, that retrieves the **objects edited in the scenario** that meet the criteria, which is equal to the Translated SQL Query with the condition *config_container_id = <repository_id>* replaced with *config_container_id = <scenario_id>*.

We then combine all the objects from **SCEN**, with those from **REP** that **are not** in the group of objects whose relevant attributes have been modified within the scenario.

Below is a pseudocode example illustrating the construction of a **SQL Query for Scenario Objects**, making use of the query's original **Translated SQL Query**, as <TRANSLATED_SQL_QUERY>, its modified version where the repository ID is replaced with the scenario ID, represented as <TRANSLATED_SQL_QUERY_WITH_SCENARIO_ID>, and a NOT IN statement with a condition block that outputs objects that have had relevant attributes modified in the scenario.

```
WITH REP AS (<TRANSLATED_SQL_QUERY>),
SCEN AS (<TRANSLATED_SQL_QUERY_WITH_SCENARIO_ID>)
SELECT * FROM REP WHERE id NOT IN (
    SELECT o.id FROM t_object o LEFT JOIN t_object_properties op ON o.id
    = op.object_id AND op.congif_container_id = <SCENARIO_ID>)
UNION SELECT * FROM SCEN;
```

5.2.3.B Query Execution Based on Context

When there is a request to generate a query, the backend receives a *config_container_id*, indicating the active context, corresponding to a repository or a scenario. Our system compares it with the ID of the main repository. If they match, the original **Translated SQL Query** is executed as is. If they differ, the system constructs the **SQL Query for Scenario Objects**. In the Java code snippet in A.5 is the custom method *executeQuery* that illustrates the construction process.

5.3 Translated SQL Queries

Next, we delve into the translations of user-written patterns supported by our system. These patterns align with the ones introduced in the previous chapter, and their equivalent translations are compatible with Atlas' relational schema.

5.3.1 Simple Dataset Selection

5.3.1.A Selecting Objects From a Class

The SQL-Based Query shown below selects all objects from the **class** *Application Component*:

```
SELECT * FROM "c=Application Component";
```

This user-written query is translated by our system into the following Translated SQL Query:

```
WITH `All Applications` AS (
SELECT o.*
FROM t_object o JOIN t_object_properties op ON o.id = op.object_id AND op.
    config_container_id = 552992
WHERE o.class_id = 553778)
SELECT o.* FROM `All Applications` o;
```

5.3.1.B Selecting the Result Set of Another Query

The SQL-Based Query shown below selects all objects from the result set of the **query** *All Applications*:

```
SELECT * FROM "q=All Applications";
```

This user-written query is translated by our system into the following Translated SQL Query:

```
WITH `rs` AS (
WITH `All Applications` AS (
    SELECT o.*
    FROM t_object o JOIN t_object_properties op ON o.id = op.object_id AND
        op.config_container_id = 552992
    WHERE o.class_id = 553778)
    SELECT DISTINCT o.* FROM `All Applications` o)
SELECT DISTINCT o.* FROM `rs` o;
```

5.3.2 Attribute Condition Filtering

5.3.2.A Single Relation Filtering

The SQL-Based Query shown below selects objects from the **class** *System Software*, from which it is applied a filter to retrieve the objects that have the **relation** *releases* equal to *.NET Core*:

```
SELECT * FROM "c=System Software" WHERE "releases" = ".NET Core";
```

This user-written query is translated by our system into the following Translated SQL Query:

```
WITH `srf` AS (
SELECT o.*
FROM t_object o JOIN t_object_properties op ON o.id = op.object_id AND op.
    config_container_id = 552992
WHERE o.class_id = 553491)
SELECT DISTINCT o.*
FROM `srf` o LEFT OUTER JOIN t_object_relation orel ON o.id=orel.origin_id
     LEFT OUTER JOIN t_base_type bt ON orel.relationtype_id = bt.id LEFT
     OUTER JOIN t_object dest ON orel.destination_id = dest.id
WHERE (orel.config_container_id = 552992 AND bt.name = "releases" AND dest.
    name = ".NET Core");
```

5.3.2.B Multiple Condition Filtering

The SQL-Based Query shown below selects all objects from the **class** *Business Actor*, where it is applied three conditions to filter the objects where the **property** *Job Title* is equal to *Manager* or is equal to *Director*, while the **relation** *located at* is equal to *Lisbon*:

```
SELECT * FROM "Business Actor" WHERE ("Job Title" = "Manager" OR "Job Title"
     = "Director") AND "located at" = "Lisbon";
```

This user-written query is translated by our system into the following Translated SQL Query:

```
WITH `mcf` AS (
SELECT o.*
FROM t_object o JOIN t_object_properties op ON o.id = op.object_id AND op.
    config_container_id = 552992
WHERE o.class_id = 553009)
SELECT DISTINCT o.*
FROM `mcf` o LEFT OUTER JOIN t_object_relation orel0 ON o.id=orel0.origin_id
     LEFT OUTER JOIN t_property p0 ON orel0.property_id = p0.id LEFT OUTER
    JOIN t_object dest0 ON orel0.destination_id = dest0.id LEFT OUTER JOIN
    t_object_relation orel1 ON o.id=orel1.origin_id LEFT OUTER JOIN
    t_property p1 ON orel1.property_id = p1.id LEFT OUTER JOIN t_object dest1
     ON orel1.destination_id = dest1.id LEFT OUTER JOIN t_object_relation
    orel2 ON o.id = orel2.origin_id LEFT OUTER JOIN t_base_type bt2 ON orel2.
    relationtype_id = bt2.id LEFT OUTER JOIN t_object dest2 ON orel2.
    destination_id = dest2.id
WHERE ((orel0.config_container_id = 552992 AND p0.name = "Job Title" AND
    dest0.name = "Manager") OR (orel1.config_container_id = 552992 AND p1.
    name = "Job Title" AND dest1.name = "Director")) AND (orel2.
    config_container_id = 552992 AND bt2.name = "located at" AND dest2.name =
     "Lisbon");
```

5.3.2.C Null/ Not Null Attribute Filtering

The SQL-Based Query shown below selects objects from the **class** *Application Component*, that have the **relation** *releases* not empty.

```
SELECT * FROM "Application Component" WHERE "releases" IS NOT NULL;
```

This user-written query is translated by our system into the following Translated SQL Query:

```
WITH `nnaf` AS (
SELECT o.*
FROM t_object o JOIN t_object_properties op ON o.id = op.object_id AND op.
    config_container_id = 552992
WHERE o.class_id = 553778)
SELECT DISTINCT o.*
FROM `nnaf` o
WHERE (o.id IN (
    SELECT o0.id
    FROM `nnaf` o0 LEFT OUTER JOIN t_object_relation orel0 ON o0.id=orel0.
        origin_id LEFT OUTER JOIN t_base_type bt0 ON orel0.relationtype_id =
        bt0.id
    WHERE orel0.config_container_id = 552992 AND bt0.name = "releases"));
```

5.3.3 Attribute Connection Selection

The SQL-Based Query shown below selects objects from the **class** *Application Component*, navigating through the *Components* **property** to retrieve the connected objects. The query is as follows:

```
SELECT "Components" FROM "Application Component";
```

This user-written query is translated by our system into the following Translated SQL Query:

```
WITH `acs` AS (
SELECT o.*
FROM t_object o JOIN t_object_properties op ON o.id = op.object_id AND op.
    config_container_id = 552992
WHERE o.class_id = 553778)
SELECT DISTINCT dest.*
FROM `acs` o LEFT OUTER JOIN t_object_relation orel ON o.id = orel.origin_id
     LEFT OUTER JOIN t_property p ON orel.property_id = p.id LEFT OUTER JOIN
    t_object dest ON orel.destination_id = dest.id
WHERE orel.config_container_id = 552992 AND p.name = "Components";
```

5.3.4 Attribute Selection With Filtering

The SQL-Based Query shown below selects objects from the result set of the **query** *Employees in Lisbon*, navigates to the set of objects connected through the **relation** *contracted by*, and then applies a filter to the resulting objects based on having the **relation** *has goal* equal to *Confidentiality Agreement*:

```
SELECT "contracted by" FROM "Employees in Lisbon" WHERE "has goal" = "
    Confidentiality Agreement";
```

This user-written query is translated by our system into the following Translated SQL Query:

```
WITH `asf` AS (
WITH `Employees in Lisbon` AS (
    SELECT o.*
    FROM t_object o JOIN t_object_properties op ON o.id = op.object_id AND
        op.config_container_id=552992
    WHERE o.class_id = 553009)
    SELECT DISTINCT o.*
    FROM `Employees in Lisbon` o LEFT OUTER JOIN t_object_relation orel0 ON
        o.id=orel0.origin_id LEFT OUTER JOIN t_property p0 ON orel0.
        property_id = p0.id LEFT OUTER JOIN t_object dest0 ON orel0.
        destination_id = dest0.id
    WHERE (orel0.config_container_id = 552992 AND p0.name = "Location" AND
        dest0.name = "Lisbon"))
SELECT DISTINCT dest0.*
FROM `asf` o LEFT OUTER JOIN t_object_relation orel0 ON o.id = orel0.
    origin_id LEFT OUTER JOIN t_base_type bt0 ON orel0.relationtype_id=bt0.id
     LEFT OUTER JOIN t_object dest0 ON orel0.destination_id = dest0.id LEFT
    OUTER JOIN t_object_relation orel1 ON dest0.id = orel1.origin_id LEFT
    OUTER JOIN t_base_type bt1 ON orel1.relationtype_id = bt1.id LEFT OUTER
    JOIN t_object dest1 ON orel1.destination_id = dest1.id
```

```
WHERE orel0.config_container_id = 552992 AND bt0.name = "contracted by" AND
    (orel1.config_container_id = 552992 AND bt1.name = "has goal" AND dest1.
    name = "Confidentiality Agreement");
```

5.3.5 Union Operation

The SQL-Based Query shown below uses a UNION operator to combine the objects connected to the **class** *Business Process* through the *realizes* **relation** with the objects from the **class** *Business Service* that have the **property** *User* equal to *Link Consulting*:

```
SELECT * FROM (SELECT "realizes" FROM "Business Process" UNION SELECT *
    FROM "Business Service" WHERE "User" = "Link Consulting");
```

This user-written query is translated by our system into the following Translated SQL Query:

```
WITH `unop` AS (
WITH CTE0 AS (
    WITH `CTE00` AS (
    SELECT o.*
    FROM t_object o JOIN t_object_properties op ON o.id = op.object_id AND
        op.config_container_id = 552992
    WHERE o.class_id = 552999)
    SELECT DISTINCT dest0.*
    FROM `CTE00` o LEFT OUTER JOIN t_object_relation orel0 ON o.id = orel0.
        origin_id LEFT OUTER JOIN t_base_type bt0 ON orel0.relationtype_id =
        bt0.id LEFT OUTER JOIN t_object dest0 ON orel0.destination_id = dest0
        .id
    WHERE orel0.config_container_id = 552992 AND dest0.name = "realizes" ),
CTE1 AS (
    WITH `CTE01` AS (
    SELECT o.*
    FROM t_object o JOIN t_object_properties op ON o.id = op.object_id AND
        op.config_container_id = 552992
    WHERE o.class_id = 554312)
    SELECT DISTINCT o.*
    FROM `CTE01` o LEFT OUTER JOIN t_object_relation orel0 ON o.id=orel0.
        origin_id LEFT OUTER JOIN t_property p0 ON orel0.property_id = p0.id
        LEFT OUTER JOIN t_object dest0 ON orel0.destination_id = dest0.id
    WHERE (orel0.config_container_id=552992 AND p0.name ="User" AND dest0.
        name = "Link Consulting"))
SELECT *
FROM CTE0 UNION SELECT * FROM CTE1)
SELECT DISTINCT o.* FROM `unop` o;
```

5.3.6 Subqueries

The SQL-Based Query shown below selects all objects from the **class** *Application Component*, first navigating through the *Database Schema* **property** to retrieve connected objects, and then using the result set to navigate through the *Technology Usage* **property** to retrieve the final connected objects:

```
SELECT "Technology Usage" FROM (SELECT "Database Schema" FROM "Application
    Component");
```

This user-written query is translated by our system into the following Translated SQL Query:

```
WITH `sub` AS (
    WITH `INNER` AS (
        SELECT o.*
        FROM t_object o JOIN t_object_properties op ON o.id = op.object_id
            AND op.config_container_id = 552992
        WHERE o.class_id = 553778)
    SELECT DISTINCT dest0.*
    FROM `INNER` o LEFT OUTER JOIN t_object_relation orel0 ON o.id = orel0.
        origin_id LEFT OUTER JOIN t_property p0 ON orel0.property_id = p0.id
        LEFT OUTER JOIN t_object dest0 ON orel0.destination_id = dest0.id
    WHERE orel0.config_container_id = 552992 AND p0.name = "Database Schema"
        )
SELECT DISTINCT dest0.*
FROM `sub` o LEFT OUTER JOIN t_object_relation orel0 ON o.id = orel0.
    origin_id LEFT OUTER JOIN t_property p0 ON orel0.property_id = p0.id LEFT
     OUTER JOIN t_object dest0 ON orel0.destination_id = dest0.id
WHERE orel0.config_container_id = 552992 AND p0.name = "Technology Usage";
```

5.3.7 Combining Capabilities

The SQL-Based Query shown below leverages multiple query capabilities, illustrating the complexity that can be found in Atlas queries:

```
SELECT "composed of" FROM (SELECT * FROM "Application Component" WHERE "
    Organic Domain" IS NOT NULL) WHERE "Database Schema" = "Atlas";
```

This user-written query is translated by our system into the following Translated SQL Query:

```
 WITH `cc` AS (
    WITH `INNER` AS (
        SELECT o.*
        FROM t_object o JOIN t_object_properties op ON o.id = op.object_id
            AND op.config_container_id = 552992
        WHERE o.class_id = 553778)
    SELECT DISTINCT o.*
    FROM `INNER` o
    WHERE (o.id IN (
        SELECT o0.id
        FROM `INNER` o0 LEFT OUTER JOIN t_object_relation orel0 ON o0.id =
            orel0.origin_id LEFT OUTER JOIN t_property p0 ON orel0.
            property_id = p0.id
        WHERE orel0.config_container_id = 552992 AND p0.name = "Organic
            Domain")))
SELECT DISTINCT dest0.*
FROM `cc` o LEFT OUTER JOIN t_object_relation orel0 ON o.id = orel0.
    origin_id LEFT OUTER JOIN t_base_type bt0 ON orel0.relationtype_id = bt0.
    id LEFT OUTER JOIN t_object dest0 ON orel0.destination_id = dest0.id LEFT
     OUTER JOIN t_object_relation orel1 ON dest0.id = orel1.origin_id LEFT
    OUTER JOIN t_property p1 ON orel1.property_id = p1.id LEFT OUTER JOIN
    t_object dest1 ON orel1.destination_id = dest1.id
WHERE orel0.config_container_id = 552992 AND bt0.name = "composed of" AND (
    orel1.config_container_id = 552992 AND p1.name ="Database Schema" AND
    dest1.name = "Atlas");
```

5.3.8 SQL Query for Scenario Objects

All SQL-Based Queries can be executed within a scenario, meaning that the system is capable of translating them into queries that retrieve objects from the user's currently active scenario.

The SQL-Based Query shown below, previously introduced in this section, selects the objects from the **class** *Application Component*:

```
SELECT * FROM "c=Application Component";
```

This user-written query is translated by our system into the following SQL Query for Scenario Objects:

```
WITH REP AS (
    WITH `All Applications` AS (
        SELECT o.*
        FROM t_object o JOIN t_object_properties op ON o.id = op.object_id
            AND op.config_container_id = 552992
        WHERE o.class_id = 553778)
    SELECT DISTINCT o.* FROM `All Applications` o),
SCEN AS (
    WITH `All Applications` AS (
        SELECT o.*
        FROM t_object o JOIN t_object_properties op ON o.id = op.object_id
            AND op.config_container_id = 1248313
        WHERE o.class_id = 553778)
    SELECT DISTINCT o.* FROM `All Applications` o)
SELECT r.*
FROM REP r
WHERE r.id NOT IN (
    SELECT op.object_id
    FROM t_object_properties op
    WHERE op.config_container_id = 1248313)
UNION SELECT * FROM SCEN;
```

5.4 The System Architecture

The system is implemented using **Java** and the **Spring Boot** framework. To handle SQL-based input, we use **JSqlParser** to parse and validate the query structure. The parsed query is then passed to our custom translation module, which maps it to the relational schema by converting references to virtual tables and columns into actual database entities.

Each query translation process involves the execution of auxiliary queries that assist in the translation, which include retrieving the ID of a specified class, retrieving the ID of a specified query, and obtaining a Translated SQL Query. Since these queries share the same structure and differ only in their arguments, we use SQL statements with placeholders, which are compiled using the **PreparedStatement** from the **Java SQL package**. Once compiled, the PreparedStatement object holds the precompiled statement with empty placeholders, allowing it to be executed for different cases without recompilation [9]. Additionally, PreparedStatement benefits from Hibernate's First Level Cache, which helps reduce query execution time [10]. A Java code snippet illustrating the use of PreparedStatement to retrieve a Translated SQL Query is provided in A.6.

The Translated SQL Query generated by our model is a plain SQL string, without any placeholders. To execute raw queries, we use the **createNativeQuery** method from **EntityManager**, which is supported by **Hibernate**, an Object-Relational Mapping (ORM) framework. This approach allows Hibernate to automatically map the results to the corresponding Entity class [11]. The translated query is executed through Hibernate's query Application Programming Interface (API), and the resulting data is formatted and returned to the Atlas platform to display in the User Interface (UI).

5.5 SQL Injection Prevention

It is critical to ensure the security of Atlas and user-written SQL queries in our system. To prevent SQL injection attacks, we use the following mechanisms:

1. Using **JSqlParser**, which parses and validates the structure of user-written queries before they are processed, ensuring that all identifiers, keywords, and clauses adhere to expected rules and do not introduce any unauthorized SQL operations besides the expected SELECT statement.

2. Using **PreparedStatement** at early stages of query translation to construct and execute auxiliary queries. This validates the string literals against SQL injection [9].

3. Having a preconstructed query template for the Translated SQL Query, with defined placeholders, where each recognized and validated SQL element is inserted, ensuring that the statements that access the database are controlled and predictable.

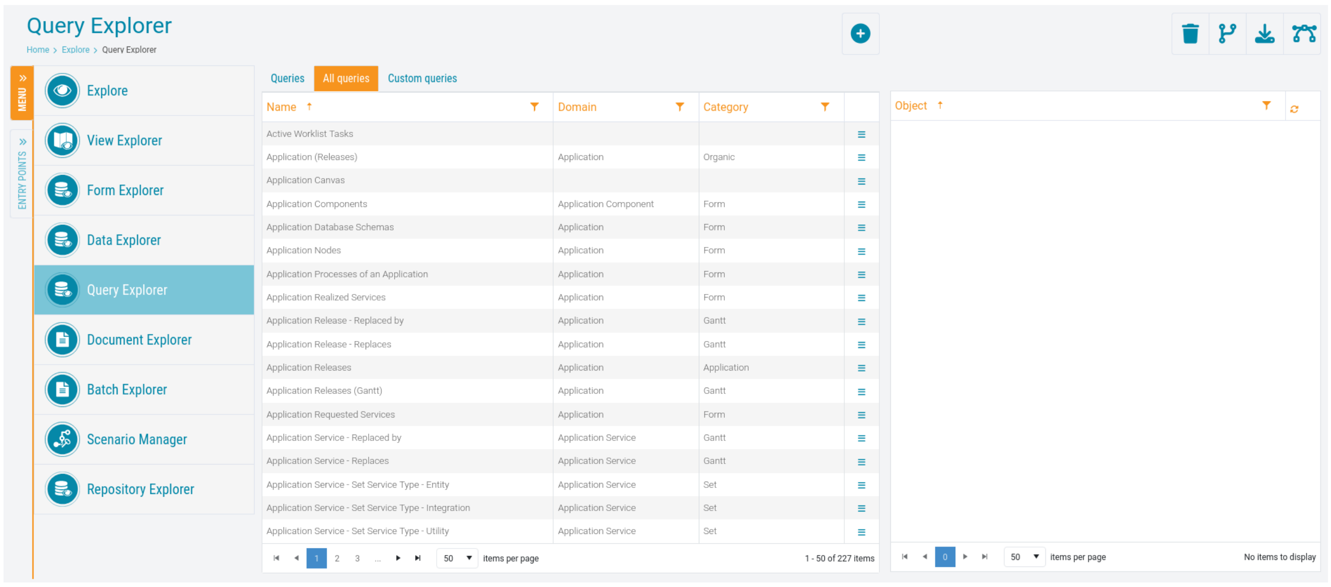
5.6 User Interface Integration

This section describes how the existing UI in the Atlas product has been adapted to support SQL-Based Queries, building on the already established **Query Explorer Menu**. The integration maintains familiar workflows while adding new functionality for managing and executing SQL queries alongside the graphical query language.

5.6.1 Query Creation and Editing

In the left-hand side of the **Query Explorer Menu**, illustrated in Figure 5.1, users can view a list of existing queries in the repository and manage them or create new ones.

To create a new query, users click the **+** button, which opens a menu where they can input the query's name and other information, as shown in Figure 5.2. After clicking the **Save and edit** button, the system sends a request to the backend to create the new query. If there are no errors, the query is saved, and the **Query Editing Menu** appears in a modal window with all fields empty except the ones previously filled, as illustrated in Figure 5.3. Users can then enter or edit information as needed. Clicking **Save** or **Save and close** sends an update request to the backend to save the changes.



Name	Domain	Category
Active Worklist Tasks		
Application (Releases)	Application	Organic
Application Canvas		
Application Components	Application Component	Form
Application Database Schemas	Application	Form
Application Nodes	Application	Form
Application Processes of an Application	Application	Form
Application Realized Services	Application	Form
Application Release - Replaced by	Application	Gantt
Application Release - Replaces	Application	Gantt
Application Releases	Application	Application
Application Releases (Gantt)	Application	Gantt
Application Requested Services	Application	Form
Application Service - Replaced by	Application Service	Gantt
Application Service - Replaces	Application Service	Gantt
Application Service - Set Service Type - Entity	Application Service	Set
Application Service - Set Service Type - Integration	Application Service	Set
Application Service - Set Service Type - Utility	Application Service	Set

1 2 3 ... 50 items per page 1 - 50 of 227 items

Figure 5.1: Atlas' Query Explorer Menu - Queries.

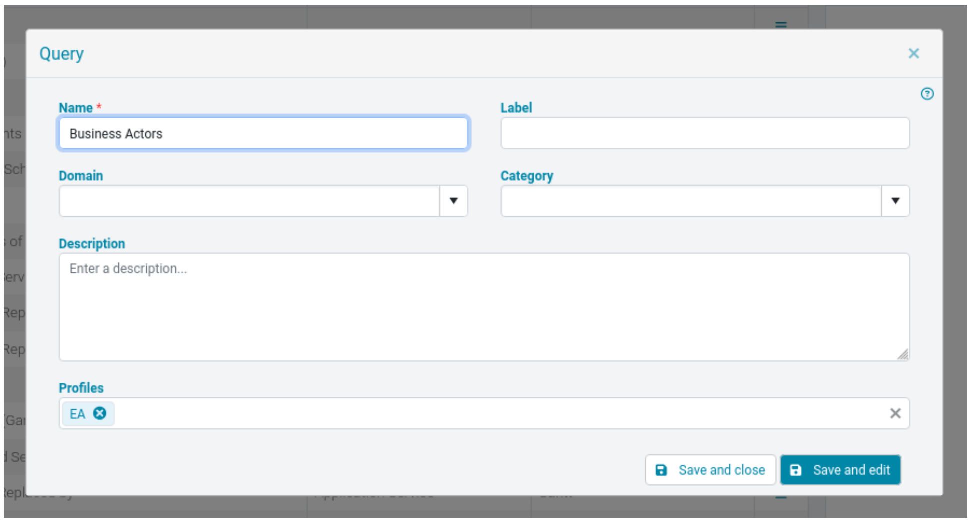
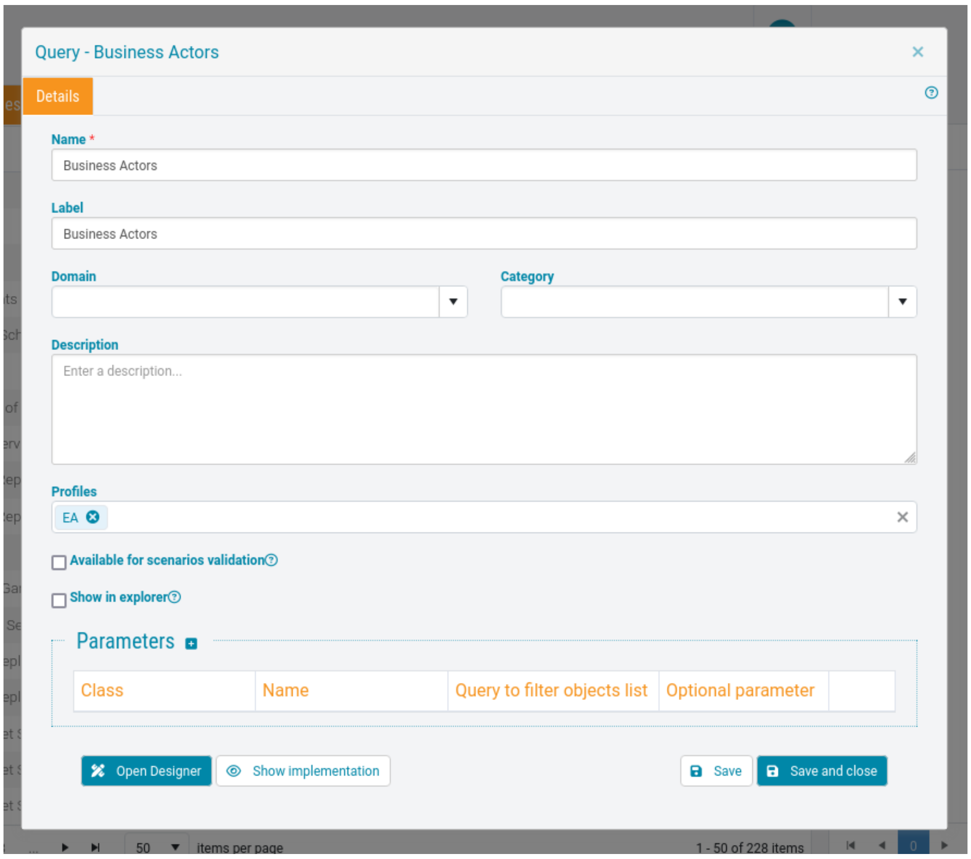


Figure 5.2: Atlas' Query Creation Menu.

To edit a query, users select it from the list in the **Query Explorer Menu** and click the **Edit** option. This opens the same **Query Editing Menu** in a modal window, where the fields are pre-filled with the query's current information. Users can make changes as needed, and after clicking **Save** or **Save and close**, the system sends an update request to the backend to save the changes.

The Atlas product provides a graphic query language interface for designing queries. In the **Query Editing Menu**, users can click the **Open Designer** button to access a full-page canvas where they can use the available symbols to model a graphic query, as shown in Figure 5.4. After completing and saving the design, an update request is sent to the backend, where the graphic query is automatically translated to a XML query language, and the user is returned to the **Query Editing Menu**. By clicking in **Show Implementation**, users can check the generated implementation code of the graphic query in a text box, illustrated in Figure 5.5.

To integrate SQL-Based Queries into the Atlas platform, we chose to reuse the implementation text


Query - Business Actors

Details

Name *

Business Actors

Label

Business Actors

Domain

Category

Description

Enter a description...

Profiles

EA

☐ Available for scenarios validation

☐ Show in explorer

Parameters

Class	Name	Query to filter objects list	Optional parameter

Figure 5.3: Atlas' Query Editing Menu.

box. Now, in addition to displaying the XML language when using the graphic query language, users who prefer to write SQL queries can enter them into the same implementation text box, as presented in Figure 5.6. However, the text box can only be used for one type of query at a time: if users input a SQL query, it will replace any graphical query, as the XML translation will no longer be shown. Similarly, if they switch to using the graphic query language, the XML translation will replace any existing SQL query.

After editing a query, either graphical or SQL based, users can click **Save and close** to return to the **Query Explorer Menu**, sending an update request to the backend. During this process:

1. If a graphic query was designed, the Worker module translates it into an XML representation that is saved in the implementation text box .
2. The contents of the implementation text box, either graphical or SQL based, are stored in the **implementation** column of the *t_query_type* table.
3. If the content starts with a SELECT statement, our parser is executed to validate the user's SQL query and generate its translation, which is then saved in the **output_query** column of the *t_query_type* table.
4. The user is returned to the Query Explorer Menu once the query is successfully translated and saved.

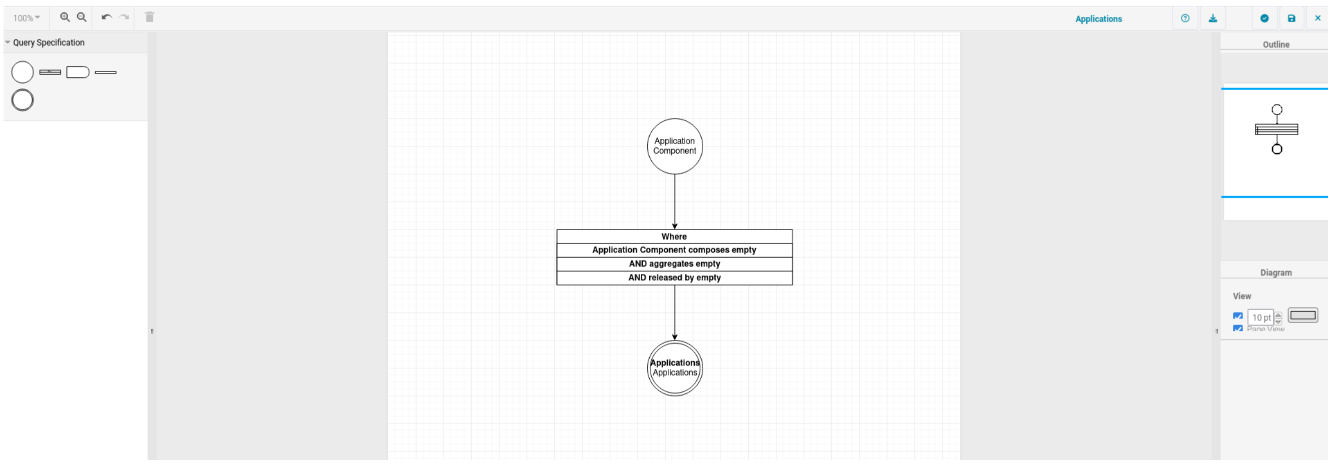


Figure 5.4: Atlas' Query Designer Menu.

5.6.2 Query Execution

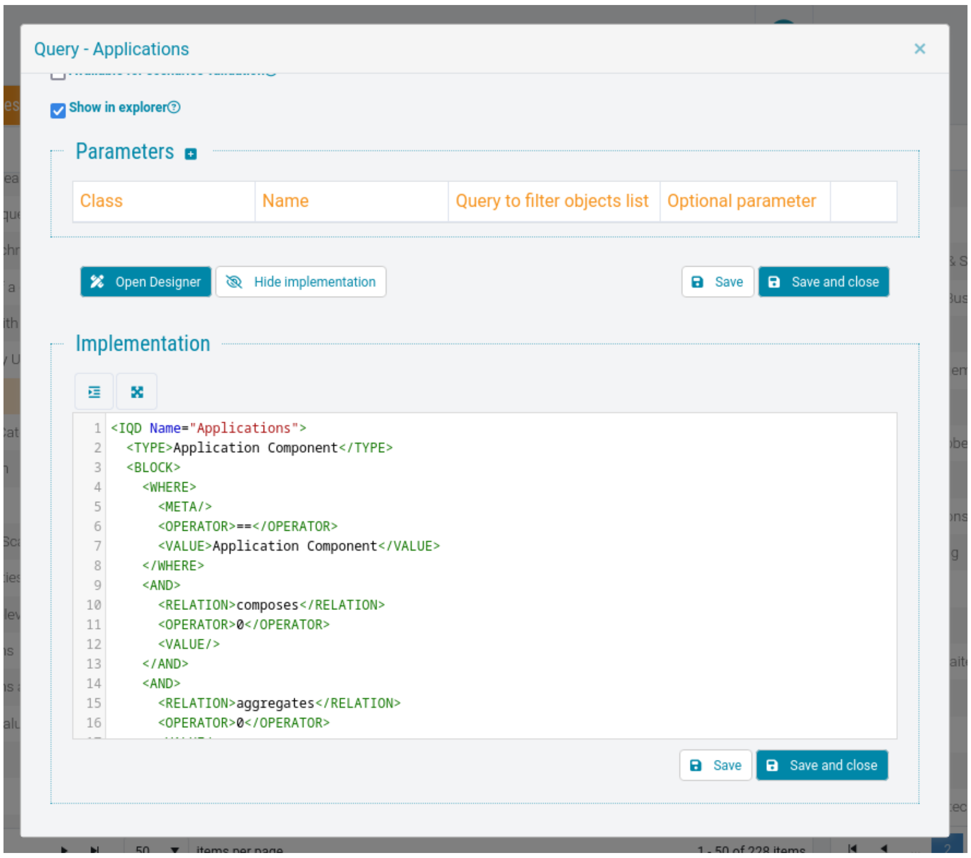
In the **Query Explorer Menu**, the created query will appear in the list on the left-hand side. When the user clicks on it, a request is sent to generate the query and the system interprets the textual information saved in the **implementation** column of the query table. If it starts with a SELECT statement, the query execution is handled by our implemented system. It retrieves the pre-translated Translated SQL Query from the **output_query** column and executes it against the database to retrieve the relevant objects. Otherwise, if the implementation is in XML language, the traditional Atlas Worker mechanism is triggered to interpret the XML code and retrieve the corresponding objects.

Once the browser receives the response, all objects returned by the query are displayed on the right-hand side of the **Query Explorer Menu**, as illustrated in Figure 5.7.

5.6.3 Scenario Verification

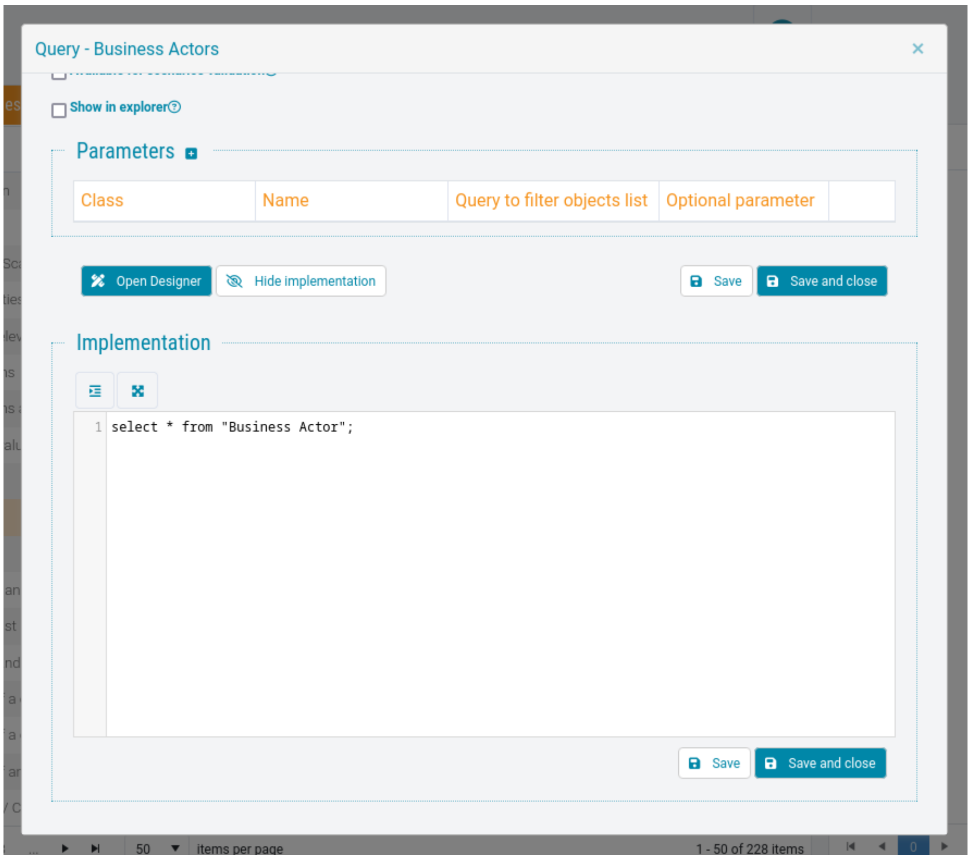
The user can activate a scenario through the top bar of the interface. Once a scenario is active, the interface switches to a darker theme, though the same queries remain accessible, as shown in Figure 5.8. Queries can be executed in the same way as in the main repository, by clicking on them. When this occurs, a request is sent to generate the query, along with the ID of the active scenario.

If the query begins with a SELECT statement, the system retrieves the Translated SQL Query from the **output_query** column and uses the scenario ID to construct the SQL Query for Scenario Objects. This query is then executed against the database to return the scenario results.



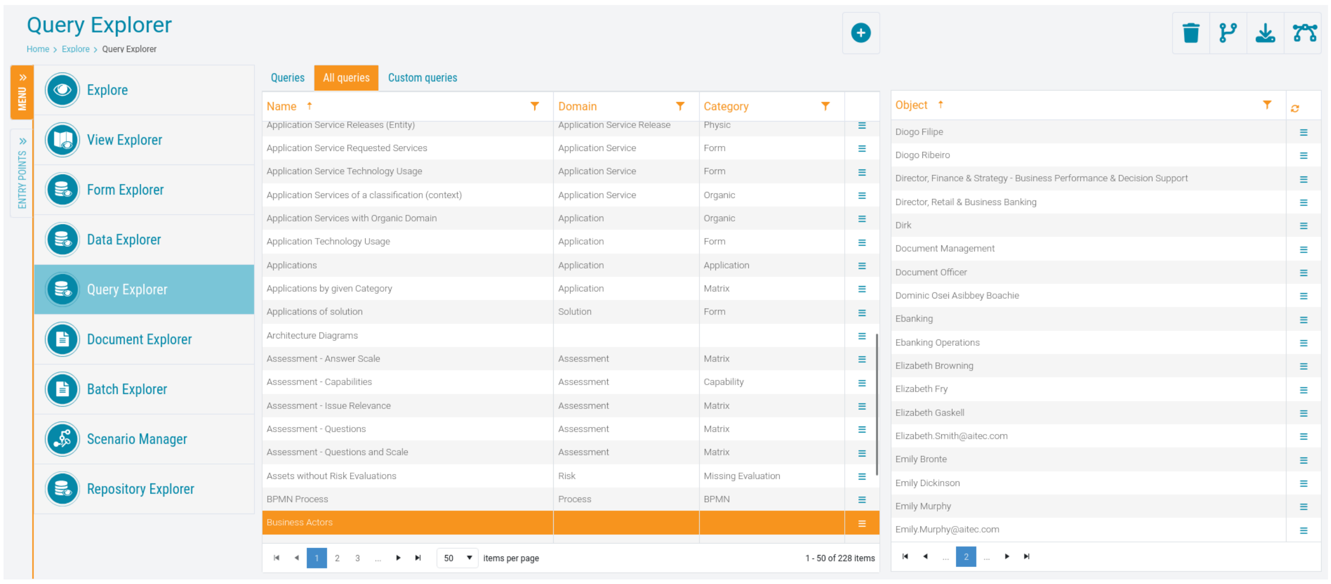
```
<IQD Name="Applications">
  <TYPE>Application Component</TYPE>
  <BLOCK>
    <WHERE>
      <META/>
      <OPERATOR>==</OPERATOR>
      <VALUE>Application Component</VALUE>
    </WHERE>
    <AND>
      <RELATION>composes</RELATION>
      <OPERATOR>0</OPERATOR>
      <VALUE/>
    </AND>
    <AND>
      <RELATION>aggregates</RELATION>
      <OPERATOR>0</OPERATOR>
```

Figure 5.5: Atlas' Implementation Text Box - XML Translation.



```
select * from "Business Actor";
```

Figure 5.6: Atlas' Implementation Text Box - SQL Query.



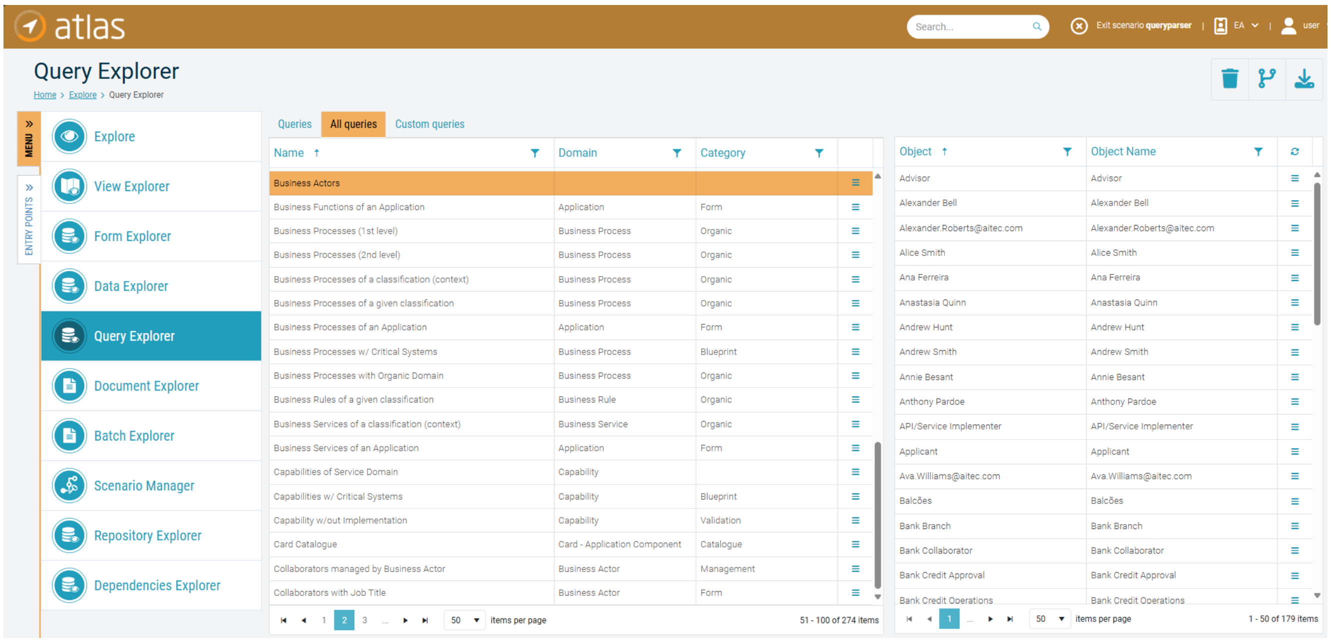
Name	Domain	Category
Application Service Releases (Entity)	Application Service Release	Physic
Application Service Requested Services	Application Service	Form
Application Service Technology Usage	Application Service	Form
Application Services of a classification (context)	Application Service	Organic
Application Services with Organic Domain	Application	Organic
Application Technology Usage	Application	Form
Applications	Application	Application
Applications by given Category	Application	Matrix
Applications of solution	Solution	Form
Architecture Diagrams		
Assessment - Answer Scale	Assessment	Matrix
Assessment - Capabilities	Assessment	Capability
Assessment - Issue Relevance	Assessment	Matrix
Assessment - Questions	Assessment	Matrix
Assessment - Questions and Scale	Assessment	Matrix
Assets without Risk Evaluations	Risk	Missing Evaluation
BPMN Process	Process	BPMN
Business Actors		

1 2 3 ... 50 items per page 1 - 50 of 228 items

Object
Diogo Filipe
Diogo Ribeiro
Director, Finance & Strategy - Business Performance & Decision Support
Director, Retail & Business Banking
Dirk
Document Management
Document Officer
Dominic Osei Asibbey Boachie
Ebanking
Ebanking Operations
Elizabeth Browning
Elizabeth Fry
Elizabeth Gaskell
Elizabeth.Smith@aitec.com
Emily Bronte
Emily Dickinson
Emily Murphy
Emily.Murphy@aitec.com

... 2 ...

Figure 5.7: Atlas' Query Explorer Menu - Query Results.



Name	Domain	Category
Business Actors		
Business Functions of an Application	Application	Form
Business Processes (1st level)	Business Process	Organic
Business Processes (2nd level)	Business Process	Organic
Business Processes of a classification (context)	Business Process	Organic
Business Processes of a given classification	Business Process	Organic
Business Processes of an Application	Application	Form
Business Processes w/ Critical Systems	Business Process	Blueprint
Business Processes with Organic Domain	Business Process	Organic
Business Rules of a given classification	Business Rule	Organic
Business Services of a classification (context)	Business Service	Organic
Business Services of an Application	Application	Form
Capabilities of Service Domain	Capability	
Capabilities w/ Critical Systems	Capability	Blueprint
Capability w/out Implementation	Capability	Validation
Card Catalogue	Card - Application Component	Catalogue
Collaborators managed by Business Actor	Business Actor	Management
Collaborators with Job Title	Business Actor	Form

1 2 3 ... 50 items per page 51 - 100 of 274 items

Object	Object Name
Advisor	Advisor
Alexander Bell	Alexander Bell
Alexander.Roberts@aitec.com	Alexander.Roberts@aitec.com
Alice Smith	Alice Smith
Ana Ferreira	Ana Ferreira
Anastasia Quinn	Anastasia Quinn
Andrew Hunt	Andrew Hunt
Andrew Smith	Andrew Smith
Annie Besant	Annie Besant
Anthony Pardoe	Anthony Pardoe
API/Service Implementer	API/Service Implementer
Applicant	Applicant
Ava.Williams@aitec.com	Ava.Williams@aitec.com
Balcões	Balcões
Bank Branch	Bank Branch
Bank Collaborator	Bank Collaborator
Bank Credit Approval	Bank Credit Approval
Bank Credit Operations	Bank Credit Operations

1 ... 50 items per page 1 - 50 of 179 items

Figure 5.8: Atlas' Query Explorer Menu - Scenario Interface.

6 Results

Contents

This chapter presents the evaluation of the implemented system. It begins by introducing the **evaluation criteria** and the **measurement methodologies** used. In addition, it presents the results obtained from the **current system**, followed by the results achieved with our **implemented solution**. Finally, the chapter concludes with a discussion on the **impact** our solution has on the Atlas product.

6.1 Evaluation Criteria

It is crucial to do a meticulous evaluation of the implemented solution, given that the project is intended for commercial use within a product for clients, where system efficiency greatly impacts user experience and client satisfaction.

To evaluate the efficiency of the implemented solution, we applied the following criteria:

1. **Accuracy** — Assessing the correctness of the SQL-Based Query results by meticulously comparing them to the results produced by their equivalent graphic queries.
2. **Performance** — Measuring query update times and query generation times of SQL-Based Queries and graphic queries, to compare performances.
3. **Robustness** — Measuring how often the system produces errors, such as invalid SQL queries that the database cannot execute, or queries that are syntactically correct but fail during execution due to misalignments.
4. **Effort** — Evaluating the effort required to construct queries using the graphic interface versus writing SQL queries.

To ensure precise measurements of **query saving times** and **query generation times**, each operation was executed 100 times, recording the total duration in nanoseconds using Java's *nanoTime* method. The **mean** value was then calculated and converted into milliseconds. Performing 100 repetitions per operation averaged out fluctuations caused by the processor and background system activity, providing a more accurate measurement. Moreover, the query construction times were recorded in seconds and obtained from four users experienced in SQL.

Additionally, for each measurement, the original SQL-Based Queries were generated **before** the corresponding graphic queries to ensure that the performance of SQL-Based Queries was not misleadingly improved by benefiting from cached database objects. After these executions, we performed the generation of SQL-based nested queries and scenario queries, meaning their performance measurements may reflect slight improvements compared to the original SQL-Based Queries due to caching effects.

6.2 Performance of the Current Solution

To evaluate the performance of the graphic queries, we constructed 70 different graphical queries and measured, in seconds, the time required for each query's **construction** from opening the canvas to completing the query design.

For each query, we measured the **saving time** using the previously defined methodology. The saving process consists of translating the graphic design into an XML representation, and storing the record in the database.

Additionally, we measured the **generation time** for each query based on the methodology described. The generation process consists of translating the XML representation into executable statements, using them to access the database and returning the corresponding results.

The following metrics were obtained (across 70 graphic queries):

– Mean query construction time: 35.886 s.

– Mean query saving time: 51.723 ms.

– Mean query generation time: $3,079.320$ ms.

6.3 Performance of the Implemented Solution

To evaluate the performance of the SQL-based input, we created equivalent SQL queries for the 70 graphic queries and measured the time required for each query's **construction**, in seconds, from opening the implementation text box to writing a valid query.

For each SQL query, we measured the **saving time** using the previously defined methodology. The saving process consists of parsing the user-written query using the custom parser, translating it into a Translated SQL Query, and storing it in the database.

Additionally, for each query, we measured the **generation time** based on the methodology described. The generation process involves retrieving the pre-translated SQL query, executing it to retrieve the results, and returning them.

The following metrics were obtained (across 70 SQL-Based Queries):

– Mean query construction time: 22.971 s.

– Mean query saving time: 70.389 ms.

– Mean query generation time: 113.670 ms.

6.4 Scenario Queries

To evaluate the performance of queries executed within a **scenario**, we generated each of the 70 user-written queries inside a loaded scenario and measured the **query generation** time using the same methodology described. The generation process in this context involves receiving the ID of the active scenario, retrieving the pre-translated SQL query to transform it into the SQL Query for Scenario Objects, using it to access the database, and returning the corresponding results.

The following metric was obtained (across 70 SQL-Based Queries executed within a scenario):

– Mean query generation time: 115.064 ms.

6.5 Nested Queries

To evaluate the implemented **nested queries** feature, each of the 70 queries was transformed into two queries: the inner query referencing a class in the FROM clause, and the outer query using the result set of the first query as its base in the FROM clause, in a way that when the outer query is executed, the combined conditions of both queries are equivalent to those of the original query.

For each outer query, we measured the **saving time** using the previously defined methodology. The saving process consists of parsing the user-written query using the custom parser, retrieving the pre-translated SQL query of the nested query, merging it with the translation of the current query to form the final Translated SQL Query, and storing it in the database.

Additionally, we measured the **generation time** based on the methodology previously described. The generation process involves retrieving the Translated SQL Query from the database, executing it to retrieve the corresponding objects, and returning them.

The following metrics were obtained (across 70 SQL-Based Queries referencing a result set):

– Mean query saving time: 79.312 ms.

– Mean query generation time: 109.293 ms.

6.6 Impact on the Product

This section presents the key remarks on the impact of the implemented solution compared to the current system in Atlas.

Accuracy — Across all evaluated queries, we meticulously compared the number of results produced by graphic queries and SQL-Based Queries and manually inspected the list of returned objects. Our evaluation confirmed that the SQL-Based Queries returned results with **100% accuracy**, exactly matching the outputs of the equivalent graphic queries.

Performance — The mean query saving time for graphic queries was 51.723 ms, compared to 70.389 ms for SQL-based input. This indicates that saving SQL queries is approximately **36.1% slower**.[1]. This is expected, given that SQL queries require additional database accesses to retrieve class or query IDs and to generate the Translated SQL Query.

In contrast, query generation showed significant improvement. The mean query generation time for graphic queries was 3,079.320 ms, while for SQL queries it was 113.670 ms. This demonstrates that query generation with the implemented solution is approximately **96.3% faster**.[2].

To provide a more comprehensive view of the performance gains for query generation, we conducted additional analyses. When comparing the mean generation times directly, to reflect the global improvement, the implemented solution is **approximately 27 times faster**[3]. Additionally, to reflect the relative improvement experienced per query, we calculated the speedup for each query individually and then the mean of the ratios. This analysis showed that, on average, the generated queries are **approximately 69 times faster**[4].

[1]Calculated as $(51.723 - 70.389)/51.723 \times 100 \approx -36.1\%$
[2]Calculated as $(3,079.320 - 113.670)/3,079.320 \times 100 \approx 96.3\%$
[3]Calculated as $3,079.320/113.670 \approx 27.1$
[4]Calculated as the mean of all (graphic query time/SQL query time) ≈ 69.0

Effort — The mean time for constructing a graphic query was 35.886 seconds, compared to 22.971 seconds for writing the corresponding SQL-Based Query. This shows that constructing SQL queries is approximately **36% faster**.[5].

Robustness — Throughout the evaluation, the system never generated invalid SQL queries. Out of 70 queries, 2 failed due to **incorrect user input**, where attribute names were misspelled, resulting in translations that attempted to verify connections with nonexistent attributes and returned empty result sets. This corresponds to a **97.1% robustness rate**[6], with all failures attributable to user input errors.

6.6.1 Scenario Queries

For the implemented **scenario** feature, we manually inspected the list of returned objects based on the manual alterations performed within the scenario. Our evaluation confirmed that queries executed within scenarios achieved **100% accuracy** compared to the original results.

In terms of performance, queries executed within a scenario had a mean generation time of 115.064 ms, compared to 113.670 ms when executed directly in the main repository. This represents a performance difference of approximately **1.2% slower**[7], indicating that the additional adjustments made to handle scenario-specific data do not significantly impact efficiency.

6.6.2 Nested Queries

For the implemented **nested queries** feature, we compared the number of results produced by queries referencing result sets with the ones returned by their corresponding SQL-based originals. Manual inspection of the returned objects confirmed **100% accuracy**, with results exactly matching the ones returned by the equivalent graphic queries.

Regarding performance, the mean query saving time for queries referencing result sets was 79.312 ms, compared to 70.389 ms, for queries referencing classes. This represents a performance difference of approximately **12.7% slower**[8]. This is expected, as nested queries require an additional database access to retrieve the Translated SQL Query of the inner query during the saving process.

On the other hand, the mean query generation time for nested queries was 109.293 ms, compared to 113.670 ms for non-nested queries. Although slightly faster (**approximately 3.9% faster**[9]), this result is likely influenced by caching effects, as nested queries executed after graphic query tests could benefit from object data still residing in cache memory. Since the generation process is identical for nested and non-nested queries, no inherent performance difference is expected under normal conditions.

All detailed measurements for the 70 evaluated queries are provided in Table B.1.

[5]Calculated as $(35.886 - 22.971)/35.886 \times 100 \approx 36.0\%$
[6]Calculated as $(70 - 2)/70 \times 100 \approx 97.1\%$
[7]Calculated as $(113.670 - 115.064)/113.670 \times 100 \approx -1.2\%$
[8]Calculated as $(70.389 - 79.312)/70.389 \times 100 \approx -12.7\%$
[9]Calculated as $(113.670 - 109.293)/113.670 \times 100 \approx 3.9\%$

6.7 Discussion

To validate the consistency and accuracy of the original performance measurements, we conducted an additional experiment using a different execution strategy. Instead of executing each query 100 times sequentially, we created a loop that executed all 70 SQL-Based Queries sequentially three times, followed by the 70 graphic queries, also executed sequentially three times. This full batch was repeated 15 times, resulting in 45 executions per query. This approach ensured that queries were not repeatedly executed in isolation, reducing the influence of database caching.

The results of this test confirmed the validity of our previous findings. The mean total generation time for graphic queries across all execution batches was 6,718.125 ms, while for SQL-Based Queries it was 194.535 ms. This reflects an improvement of approximately **97.1%**[10], aligning with the original 96.3% result obtained from the initial evaluation. The difference of less than 1% indicates a narrow margin of variability, affirming the robustness of the initial performance measurements.

All detailed measurements for this additional experiment are provided in Table B.2.

[10]Calculated as $(6,718.125 - 194.535)/6,718.125 \times 100 \approx 97.1\%$

7 Conclusion

Contents

This chapter presents a reflection on the work developed. First, we discuss important considerations regarding **query structure**, **data volume**, and the use of **indexes** to optimize query efficiency. Then, we summarize the **main conclusions** drawn from the implementation and evaluation of the solution. Finally, we outline the **limitations** identified in the current system and propose potential directions for **future developments**.

7.1 Considering Query Structure, Data Volume, and Indexes

In parallel with our implementation, we explored an alternative strategy where, instead of creating a CTE for each query, we attempted to **directly modify** the inner query by adding additional translations. We made further adaptations to the JSqlParser framework to parse the inner query's translation and modify the SELECT, FROM and WHERE clauses, directly. However, this method resulted in increasingly lengthy queries with multiple JOIN operations and late filtering, resulting in costly query execution, due to the heavy processing of large amounts of unnecessary data.

However, adopting CTEs provided performance benefits, by reducing the volume of rows processed at earlier stages of query execution, decreasing the data volume involved in subsequent JOIN operations. Additionally, CTEs simplified complex query logic into smaller and more manageable building blocks, facilitating query interpretation and optimization.

We also investigated the possibility of using **materialized views** or **temporary tables**, and concluded that they would not offer advantages in our context. Given the continuous evolution of Atlas' data, whose objects are modified daily, storing intermediate results in the database would be impractical, risking displaying outdated results.

In addition, we investigated using the method **Statement** from the Java SQL Package, instead of the chosen method createNativeQuery from Java Persistence API (JPA), to execute the translated queries, since they both execute raw SQL with no placeholders. However, by using createNativeQuery, we benefited from Hibernate's support for entity mapping, while still taking advantage of caching, session management, and efficient query execution [12].

In our implemented model, all columns involved in JOIN conditions or filtering criteria were **indexed** appropriately, and we benefited from CTEs inheriting indexes from the underlying tables, improving query performance [13].

7.2 Overall Conclusions

The performance of saving SQL-based queries significantly improved after our system provided a mechanism for users to distinguish between class and query names using the prefixes "c=" or "q=" in the FROM clause, removing the necessity for unnecessary database accesses to verify whether a name corresponds to a class or a query.

Additionally, retrieving the IDs corresponding to class and query names during the translation phase proved to be beneficial. Since searching for a ID by its name is an expensive operation, retrieving it during the translation phase allowed the generation of a Translated SQL Query that directly references the *class_id*, eliminating the need of performing the costly search during query execution.

The approach adopted in this work prioritized optimizing query generation performance, even at the cost of slightly increased query saving times. This trade-off was considered acceptable, as saving operations occur less frequently and have minimal impact on user experience compared to query execution times.

The results obtained validate the success of this approach. The implemented system achieved a **96% performance improvement** in query generation times compared to the existing solution, fulfilling the goal of delivering efficient query generation. Although saving SQL-Based Queries is approximately **36% slower** than saving graphic queries, this overhead is acceptable given the context of the broader objectives.

In summary, the implemented system successfully meets its goals, delivering substantial improvements in operations critical to user interaction speed while maintaining high accuracy and robustness.

7.3 Limitations

When we save a query containing a nested query, the translation of the inner query is retrieved and combined with the outer query's translation, and the generated Translated SQL Query is then saved in the database. Consequently, if the inner query is edited afterward, the changes do not propagate to the translations of the queries that depend on it. As a result, unless the user re-saves those dependent queries, to update the Translated SQL Query with the new translation, they continue using an outdated version of the inner query.

7.4 Future Developments

To address this limitation, we propose two solutions:

1. When saving a translated query in the database, rather than including the translated inner query within the CTE, we could instead save a **PreparedStatement** for the outer query translation using a placeholder within the CTE, such as `WITH <CTE_NAME> AS (?)`. Subsequently, when executing the outer query, we would first retrieve the current translation of the inner query and execute our query using a PreparedStatement, substituting the placeholder with the retrieved translation. While this approach ensures query results are current, it could increase query execution time due to additional database accesses.

2. An alternative solution involves using the existing **batch** processing capabilities in Atlas. We could create a batch process that periodically identifies modified queries and automatically updates the translations of dependent queries. This method would ensure updates are consistently applied without affecting query execution performance.

To improve user experience, future work on the UI includes:

1. Currently, the implementation text box is reused for SQL-Based Queries, which is useful for leveraging the existing Atlas UI, however, this can lead to errors, as the text in the box is not persistent and is automatically replaced by the XML translation if a graphic query is saved. We plan to keep the **Open Designer** and the **Show Implementation** buttons dedicated to graphic queries and introduce a separate, clearly labeled text box for direct SQL query input. If both graphic and SQL implementations are submitted for the same query, the user can be prompted to choose which one to execute.

2. At present, there is no feedback mechanism to help users detect syntax errors in user-written queries. We intend to implement a **feedback** system to assist in detecting syntax errors, providing insight into which specific clause failed validation.

3. Instead of requiring users to write the full query from scratch, we want the UI to provide guided query construction. Users may select the **query template** they want to create which automatically

populates the query box with the corresponding empty clauses. From there, users can choose relations, properties, classes, or queries from **dropdown lists**, reducing the likelihood of errors, speeding up the query creation process, and also allowing distinction between classes and queries in the from clause.

4. When a user edits and saves a query, the system could exit the Query Editing Menu once the parser validates the input, and the query translation and saving processes could be handled asynchronously in the background. This would reduce the perceived saving time from the user's perspective.

Currently, scenario queries aren't saved in the database, as it isn't considered worthwhile since scenarios aren't used regularly. Although, if that requirement changes, it is viable to create a new column in the *t_query_type* table to store scenario related query translations, with a placeholder in the *config_container_id* condition to populate it with the *scenario_id* of the loaded scenario.

Lastly, the results of this work demonstrate that SQL-based queries are substantially more efficient than the XML translation mechanism used for graphic queries. As future work, it can be valuable to explore the possibility of translating graphic queries directly into SQL-Based Queries and corresponding Translated SQL Queries, eliminating the inefficient XML representation. Such an approach could combine the user-friendly advantages of graphic query design with the performance benefits of the SQL-based execution model.

All diagrams in this thesis were created using Drawio (available at `https://www.drawio.com`).

Bibliography

[1] A. Moreira, "Link launches cybersecurity solution based on atlas," 2024, accessed: 2024-01-07. [Online]. Available: https://linkconsulting.com/blog/link-launches-cybersecurity-solution-based-on-atlas-cybersecurity-from-infrastructure-to-business-services/

[2] S.-Y. Hsieh, C. Chang, P. Mongkolwat, W. W. Pilch, and C.-C. Shih, "Capturing the objected-oriented database model in relational form," in *Proceedings of 1993 IEEE 17th International Computer Software and Applications Conference COMPSAC '93*, 1993, pp. 202–208.

[3] Y.-F. Huang and W.-F. Kuo, "The query translation between object-oriented and relational databases," *Journal- Chinese Institute of Engineers*, vol. 26, pp. 715–720, 2003.

[4] J. Fong and S. K. Cheung, "Translating oodb method to rdb routine," *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 03, pp. 329–355, 2001.

[5] L. Francalanci and R. Manning, "Jsqlparser," accessed: 2024-05-31. [Online]. Available: https://github.com/JSQLParser/JSqlParser

[6] ——, "Java sql parser library," 2024, accessed: 2024-10-15. [Online]. Available: https://jsqlparser.github.io/JSqlParser/index.html

[7] ——, "Jsqlparser," 2024, accessed: 2024-10-15. [Online]. Available: https://jsqlparser.sourceforge.net/

[8] w3resource, "Sql subqueries," 2024, accessed: 2024-10-15. [Online]. Available: https://www.w3resource.com/sql/subqueries/understanding-sql-subqueries.php

[9] Oracle, "Using prepared statements," 2024, accessed: 2025-04-04. [Online]. Available: https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html

[10] Pankaj, "Hibernate caching - first level cache," 2022, accessed: 2025-02-01. [Online]. Available: https://www.digitalocean.com/community/tutorials/hibernate-caching-first-level-cache

[11] Hibernate, "Queryproducer interface documentation," accessed: 2025-02-01. [Online]. Available: https://docs.jboss.org/hibernate/orm/6.2/javadocs/org/hibernate/query/QueryProducer.html#createNativeQuery(java.lang.String,java.lang.String,java.lang.Class)

[12] GeeksforGeeks. (2025) Hibernate native sql query with example. Accessed: 2025-05-01. [Online]. Available: https://www.geeksforgeeks.org/hibernate-native-sql-query-with-example/

[13] A. Zanini, “Sql server cte: Everything you need to know,” 2024, accessed: 2025-04-04. [Online]. Available: https://www.dbvis.com/thetable/sql-server-cte-everything-you-need-to-know/

A

Code of Project

In this appendix, we present examples of code snippets from the implemented system.

```java
public class QueryParser {
    private PlainSelect plainSelect;

    public QueryParser(String query) {
        Statement statement;

        try {
            statement = CCJSqlParserUtil.parse(query);
        } catch (JSQLParserException e) {
            // Query is invalid: throw exception
        }
        if (!(statement instanceof PlainSelect)) {
            // Query is not a SELECT statement: throw exception
        }
        plainSelect = (PlainSelect) statement;
    }
    public String getSelectItem() {

        String str = plainSelect.getSelectItem(0).toString();
        if (str.equals("*")) {
            return str;
        } else if (str.length() > 2 && str.startsWith("\"") && str.endsWith("\"")) {
            return str.substring(1, str.length() - 1);
        } else if (str.length() == 2 && str.startsWith("\"") && str.endsWith("\"")) {
            // SELECT item is empty: throw exception
        } else {
            // SELECT item is not enclosed in quotation marks: throw exception
        }
    }
    public String fromClausetoString() {

        String str = plainSelect.getFromItem().toString();
        if (str.length() > 2 && str.startsWith("\"") && str.endsWith("\"")) {
            return str.substring(1, str.length() - 1);
        } else if (str.length() == 2 && str.startsWith("\"") && str.endsWith("\"")) {
            // FROM item is empty: throw exception
        } else {
            // FROM item is not enclosed in quotation marks: throw exception
        }
    }
    public Expression getWhereClause() {
        return plainSelect.getWhere();
    }
}
```

Listing A.1: QueryParser Class - Lexical and Syntax Analysis of SQL Queries.

```java
public class QueryHandler {
    public String interpreteQuery(String inputQuery, String variableName, Long repositoryId) {

        QueryBuilder qb = new QueryBuilder();

        QueryParser parser = new QueryParser(inputQuery);
        String selectClause = parser.getSelectItem();
        FromItem fromClause = parser.getFromClause();
        Expression whereClause = parser.getWhereClause();
        //Additional query processing
    }
}
```

Listing A.2: QueryHandler Class - Token Collection for Query Interpretation.

```java
public void handleSelectClause(String selectItem, String variableName, Long
    repositoryId) {

    if (selectItem.equals("*")) {
    // Select objects from CTE
        select += "o.* ";
        from += "`" + variableName + "` o ";
    } else if (Character.isLowerCase(selectItem.charAt(0))) {
    // Select objects connected to CTE through the Relation
        select += "DISTINCT dest" + counter + ".*";
        from += // JOINS until connected objects
        + "LEFT OUTER JOIN t_object dest" + counter + " ON orel" + counter +
            ".destination_id = dest" + counter + ".id ";
        // Additional clause construction

    } else if (Character.isUpperCase(selectItem.charAt(0))) {
    // Select objects connected to CTE through the Propery
        select += "DISTINCT dest" + counter + ".*";
        from += // JOINS until connected objects
        + "LEFT OUTER JOIN t_object dest" + counter + " ON orel" + counter +
            ".destination_id = dest" + counter + ".id ";
        // Additional clause construction
    }
}
```

Listing A.3: HandleSelectClause Method - Translation of the Select Clause.

```java
public void handleAndExpression(Expression expression) {
    AndExpression andExpression = (AndExpression) expression;
    logicalExpressions.push(andExpression.getRightExpression());
    logicalExpressions.push(andExpression.getLeftExpression());
    operators.push("AND "); // Logical operators stack
}

public void handleOrExpression(Expression expression) {
    OrExpression orExpression = (OrExpression) expression;
    logicalExpressions.push(orExpression.getRightExpression());
    logicalExpressions.push(orExpression.getLeftExpression());
    operators.push("OR "); // Logical operators stack
}
public void handleEqualsToExpression(Expression expression, Long
    repositoryId) {

    String query = "";
    EqualsTo equalsTo = (EqualsTo) expression;

    // Verify if WHERE clause names are enclosed in quotation marks, if not
        throw exception

    if (Character.isLowerCase(equalsTo.getLeftExpression().toString().charAt
        (1))) {
    // Condition refers to a Relation
        from += // JOINS to relation table bt and connected objects dest
        where += " AND bt" + counter + ".name=" + equalsTo.getLeftExpression
            ().toString() + " AND dest" + counter + ".name=" + equalsTo.
            getRightExpression().toString() + " ";
    } else if (Character.isUpperCase(equalsTo.getLeftExpression().toString()
        .charAt(1))) {
   // Condition refers to a Property
        from += // JOINS to property table p and connected objects dest
        where += " AND p" + counter + ".name=" + equalsTo.getLeftExpression
            ().toString() + " AND dest" + counter + ".name="+ equalsTo.
            getRightExpression().toString() + " ";
    }
    if (!operators.isEmpty()) {
        query += operators.pop();
    }
    where += query;
}
public void handleNullExpression(Expression expression, String variableName,
     Long repositoryId) {

IsNullExpression isNull = (IsNullExpression) expression;
String query = "";

// Verify if WHERE clause names are enclosed in quotation marks, if not
    throw exception
```

```java

    if (isNull.isNot() == false) { // Operator IS NULL
        if (Character.isLowerCase(isNull.getLeftExpression().toString().charAt(1))) { // Relation
            where += "o.id NOT IN (SELECT ob.id FROM "+ variableName + " ob LEFT OUTER JOIN t_object_relation" + " r ON ob.id=r.origin_id LEFT OUTER JOIN t_base_type b ON r.relationtype_id = b.id WHERE b.name=" + isNull.getLeftExpression().toString()+")"; // Objects that are in the group of those that have no connection to that Relation
        } else if (Character.isUpperCase(isNull.getLeftExpression().toString().charAt(1))) {// Property
            // Objects that are in the group of those that have no connection to that Property
        }
    } else { // Operator IS NOT NULL
        if (Character.isLowerCase(isNull.getLeftExpression().toString().charAt(1))) { // Relation
            // Objects that are in the group of those that have a connection to that Relation
        } else if (Character.isUpperCase(isNull.getLeftExpression().toString().charAt(1))) { // Property
            // Objects that are in the group of those that have a connection to that Property
        }
    }
    if (!operators.isEmpty()) {
        query += operators.pop();
    }
    where += query;
}
public void handleWhereClause(Expression whereClause, String variableName, Long repositoryId) throws MinervaQueryParserException {

    // Verify if WHERE clause is not null

    logicalExpressions.push(whereClause); // Expressions stack

    while (!logicalExpressions.isEmpty()) {
        Expression expression = logicalExpressions.pop();
        switch (expression.getClass().getSimpleName()) {
            case "AndExpression":
                handleAndExpression(expression);
                break;
            case "OrExpression":
                handleOrExpression(expression);
                break;
            case "EqualsTo":
                handleEqualsToExpression(expression, repositoryId);
                break;
            case "IsNullExpression":
            handleNullExpression(expression, variableName, repositoryId);
                break;
            default:
                break;
        }
    }
    return;
}
```

Listing A.4: HandleWhereClause and Related Methods - Translation of the Where Clause.

```java
public List<Object> executeQuery(String query, Long configContainerId, Long
    repositoryId) {
    try {
        if (query.length() > 0) {
            String newQuery = "";

            if (configContainerId != null && !configContainerId.equals(
                repositoryId)) {
                // The query is being executed in a scenario
                // Construction of SQL Query for Scenario Objects

            } else if (configContainerId != null && configContainerId.equals
                (repositoryId)) {
                // The query is being executed in the main repository
                newQuery = query;
            }
            Query compQuery = em.createNativeQuery(newQuery, Object.class);
            return compQuery.getResultList();

        } else {
            // Translated query is empty: throw exception
        }
    } catch (PersistenceException e) {
        // Error executing query: throw exception
    }
}
```

Listing A.5: ExecuteQuery Method - Modifying the Translated Query Based On Scenario Context.

```
public String getSQLOutputQuery(Long id, Long repositoryId) {
        Session session = em.unwrap(Session.class);
        return session.doReturningWork(connection -> {
            String query = "SELECT output_query FROM t_query_type WHERE id =
                 ? AND repository_id = ?";
            try (PreparedStatement ps = connection.prepareStatement(query))
                {
                ps.setLong(1, id);
                ps.setLong(2, repositoryId);
                try (ResultSet rs = ps.executeQuery()) {
                    if (rs.next()) {
                        return rs.getString("output_query");
                    }
                }
            } catch (SQLException e) {
                // Error retrieving query: throw exception
            }
            return null;
        });
    }
```

Listing A.6: Retrieving a Translated SQL Query using PreparedStatement.

B

Query Performance Measurements

In this appendix, we present Table B.1 containing the query performance measurements used to evaluate the implemented system. We created 70 different query types, each corresponding to a row in the table. For each query type, we constructed both a graphic query and an equivalent SQL-Based Query, measuring the construction, saving, and execution times, as well as the number of results returned, which is identical for both approaches. For SQL-Based Queries, we measured the time to generate queries within a scenario, and the times required to save and generate nested queries. The last row of the table presents the mean values for all measured times.

Additionally, Table B.2 presents the results of an alternative execution strategy designed to validate the main performance measurements. In this approach, all 70 SQL-Based Queries were executed sequentially three times, followed by the 70 graphic queries executed three times. This sequence was repeated 15 times, and each row in the table represents one of these execution batches. The table reports the total generation time for each query type per batch, with the final row showing the mean values.

For both tables, the times measured in nanoseconds are displayed in microseconds for easier readability.

Table B.1: Evaluation Results for 70 Queries.

Query ID	Constructing Graphic (s)	Constructing SQL (s)	Saving Graphic (µs)	Saving SQL (µs)	Generating Graphic (µs)	Generating SQL (µs)	Saving Nested (µs)	Generating Nested (µs)	Scenario Generating (µs)	Result Count
1	45	27	75,136	87,945	1,303,265	28,624	238,423	27,952	11,762	2
2	35	21	121,485	119,780	1,417,533	69,431	136,959	151,208	88,529	34
3	47	31	52,028	62,087	1,072,068	21,064	156,965	66,825	17,493	10
4	45	31	49,623	83,267	5,694,470	32,670	70,507	87,649	26,592	21
5	59	40	53,257	88,139	7663,960	42,806	124,811	49,783	59,873	1
6	48	22	48,612	67,416	881,044	89,741	93,796	98,722	82,657	58
7	62	43	50,825	70,161	5,828,036	124,810	146,077	262,108	126,830	513
8	48	28	43,535	57,469	1,576,545	86,572	89,584	131,627	91,870	1
9	52	30	46,876	80,087	2,015,661	364,470	168,438	364,917	308,729	9
10	20	13	33,992	57,244	10,634,684	33,872	85,677	95,027	68,494	514
11	115	54	81,605	147,388	1,095,549	59,533	64,184	58,758	60,990	17
12	83	52	28,723	51,912	1,122,983	46,408	87,844	45,028	27,921	17
13	33	23	80,587	84,947	1,078,272	87,578	92,655	135,707	179,527	81
14	34	20	83,133	142,984	1,845,187	250,575	84,066	200,721	113,702	443
15	85	65	50,895	93,040	7,131,621	1,847,251	78,293	777,820	1,462,453	22
16	49	36	40,615	74,913	71,0236	14,165	98,082	22,795	15,496	1
17	99	48	64,402	128,456	2,965,636	50,519	207,714	59,408	122,490	7
18	41	31	46,886	92,758	10,265,349	121,809	86,914	171,784	113,650	7
19	69	48	42,694	82,523	6,843,709	199,726	95,148	193,426	98,763	0
20	73	53	93,229	228,927	7,581,281	158,425	52,587	175,669	127,488	1
21	63	40	43,554	72,463	6,022,325	150,514	105,533	140,332	282,059	42
22	33	19	51,563	72,432	6,495,070	150,521	85,825	168,010	125,595	45

Continued on next page

Table B.1 – continued from previous page

Query ID	Constructing Graphic (s)	Constructing SQL (s)	Saving Graphic (µs)	Saving SQL (µs)	Generating Graphic (µs)	Generating SQL (µs)	Saving Nested (µs)	Generating Nested (µs)	Scenario Generating (µs)	Result Count
23	33	21	34,845	71,739	683,641	22,249	72,990	46,386	15,492	28
24	35	22	77,923	62,889	466,618	33,878	94,070	12,458	20,577	26
25	35	13	39,705	88,767	69,300	12,399	84,019	11,364	18,390	10
26	33	19	45,834	60,164	3,110,209	15,640	62,263	33,585	33,527	8
27	34	22	48,362	84,398	2,088,074	10,159	58,463	73,276	26,059	30
28	43	27	39,473	57,424	1,011,151	19,636	58,593	53,775	41,122	60
29	54	33	64,282	64,305	1,776,327	16,017	119,898	44,012	31,553	1
30	28	19	47,025	62,338	1,761,097	15,645	72,337	82,727	16,627	1
31	31	22	49,444	59,648	1,779,113	11,691	73,703	73,915	20,245	1
32	13	8	65,718	65,492	2,914,136	66,968	59,584	45,002	92,701	317
33	50	32	49,204	59,809	1,772,317	14,445	69,874	120,803	33,568	8
34	27	19	31,212	46,831	1,379,931	102,122	60,037	191,940	274,849	61
35	32	25	40,879	39,947	1,509,798	10,832	47,867	123,580	13,917	12
36	49	33	44,566	45,536	1,869,520	1,509,312	65,694	481,774	1,346,411	65
37	34	22	46,199	34,301	9,362,525	20,514	69,061	46,754	36,263	5
38	33	14	43,404	49,052	6,763,654	158,400	64,968	247,344	86,402	34
39	27	17	45,865	48,347	7,265,307	120,505	58,504	148,754	158,392	120
40	19	14	34,180	40,234	2,132,788	139,040	70,153	144,558	113,317	45
41	22	14	30,162	44,161	10,074,957	113,536	42,662	104,739	123,818	253
42	23	15	45,720	60,835	7,399,088	86,256	131,071	269,075	76,099	108
43	19	13	40,259	46,963	6,251,767	168,804	66,796	179,421	187,899	63
44	21	14	39,576	55,685	988,124	38,362	57,111	85,302	27,381	58
45	23	14	39,439	56,299	2,599,285	50,725	84,259	34,459	44,687	58

Continued on next page

Table B.1 – continued from previous page

Query ID	Constructing Graphic (s)	Constructing SQL (s)	Saving Graphic (µs)	Saving SQL (µs)	Generating Graphic (µs)	Generating SQL (µs)	Saving Nested (µs)	Generating Nested (µs)	Scenario Generating (µs)	Result Count
46	17	8	50,792	48,237	793,315	14,205	54,866	12,613	15,375	68
47	20	13	39,506	56,326	55,0145	14,692	48,678	17,488	73,550	53
48	20	11	31,547	75,004	442,866	20,327	84,831	55,923	73,583	45
49	27	17	62,315	63,830	964,319	32,021	93,427	39,503	47,959	51
50	23	14	45,188	62,564	3,811,577	85,364	96,943	141,384	42,188	39
51	22	20	67,846	61,750	3,993,885	24,968	51,502	64,447	22,618	24
52	23	15	37,108	52,796	3,363,852	18,666	57,763	83,809	47,505	4
53	19	13	45,135	59,511	3,799,013	95,204	70,457	66,409	28,748	3
54	22	17	65,194	55,171	3,133,281	31,056	48,474	97,102	26,767	1
55	30	23	32,350	56,869	2,663,195	76,418	86,629	140,507	64,812	182
56	30	20	97,295	46,659	1,004,123	131,911	44,893	66,739	48,346	31
57	27	19	67,275	115,411	3,106,079	468,81	50,942	73,608	73,078	81
58	31	19	71,541	76,292	1,380,998	49,039	46,126	57,427	142,278	232
59	24	18	50,832	76,352	826,702	35,882	50,122	116,467	160,397	263
60	24	19	65,234	73,835	2,772,641	68,542	69,330	51,830	127,558	259
61	19	13	66,541	65,328	1,746,841	34,632	53,051	46,498	55,669	5
62	19	15	46,338	59,825	1,269,487	20,298	43,311	122,514	34,966	13
63	28	16	57,903	57,180	1,416,427	81,923	40,641	44,608	36,444	6
64	29	15	36,933	59,882	1,477,011	59,949	54,686	43,449	147,181	289
65	24	18	43,118	62,833	2,254,509	104,725	55,036	38,848	45,777	6
66	25	16	39,454	57,736	1,852,282	66,469	56,077	51,472	163,206	289
67	14	12	31,186	56,744	330,380	7,419	44,786	11,124	15,583	14
68	13	11	46,655	58,184	3,314,312	22,529	62,006	41,429	33,377	284

Continued on next page

Table B.1 – continued from previous page

Query ID	Constructing Graphic (s)	Constructing SQL (s)	Saving Graphic (µs)	Saving SQL (µs)	Generating Graphic (µs)	Generating SQL (µs)	Saving Nested (µs)	Generating Nested (µs)	Scenario Generating (µs)	Result Count
69	14	11	58,045	54,619	546,830	8,959	42,431	9,632	20,556	13
70	12	8	38,770	62,791	2,465,091	16,580	50,756	15,405	54,676	135
Mean	**35.886**	**22.971**	**51,723**	**70,389**	**3,079,320**	**113,670**	**79,312**	**109,293**	**115,064**	

Table B.2: Execution Times of 70 Graphic and SQL-Based Queries Over 15 Batches.

Execution Batch	Generating Graphic (μs)	Generating SQL (μs)
1	7,302,045	338,822
2	6,671,329	194,072
3	6,525,536	176,813
4	6,901,691	175,610
5	6,805,839	196,562
6	6,884,951	176,273
7	6,652,580	158,670
8	6,677,927	200,870
9	6,698,828	167,380
10	6,694,286	190,867
11	6,569,962	205,669
12	6,559,885	191,559
13	6,831,136	195,824
14	6,544,750	170,686
15	6,451,133	178,355
Mean	**6,718,125**	**194,535**