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

MARIA JOÃO MADEIRA DUARTE, Instituto Superior Técnico, Portugal
PEDRO MANUEL MOREIRA VAZ ANTUNES DE SOUSA, Instituto Superior Técnico, Portugal

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

1 INTRODUCTION

Atlas is an enterprise architecture platform developed to help organizations visualize and manage their architectural artifacts and models. To facilitate data retrieval for analysis, Atlas includes the Query Designer, a graphical query editor that allows users to construct queries using a predefined syntax. However, this query mechanism relies on a translation process based on Extensible Markup Language (XML), which has proven to be slow and inefficient. Moreover, its construction requires extensive manual interaction, that makes complex queries time-consuming to build and modify.

To address these limitations, we developed a new system for users to write queries in Structured Query Language (SQL). This requires the design of a Virtual Schema that allows users to interact with an intuitive structure, that abstracts the complexities of the internal relational model. The solution also introduces a query translation mechanism, execution optimization techniques, and support for advanced features such as scenario verification and nested queries.

This SQL-based feature allows users to define complex queries more efficiently with intuitive syntax, while maintaining the performance expectations of the Atlas platform.

2 CONTRIBUTIONS

The work developed in this thesis provided the following contributions:

- A virtual schema that maps Atlas' classes, queries, and attributes to virtual tables and columns, abstracting the complexity of the relational model.
- A translation mechanism built using JSqlParser [1], responsible for parsing, validating, and transforming various types of user-written SQL queries into executable statements.
- Optimization techniques that eliminate redundant parsing, by introducing a new relational column that stores the pre-translated SQL queries, significantly improving query generation time and allowing support for nested query execution.
- Support for query execution within Atlas' scenarios.
- Performance evaluation through practical use cases, testing query accuracy and improvements in efficiency compared to the existing graphical query system.

3 BACKGROUND ON THE ATLAS PLATFORM

Atlas is an enterprise architecture tool [2] developed by **Link Consulting** to help organizations manage complex ecosystems by providing integrated views of their architectural layers and components. Inspired by the ArchiMate language, Atlas extends its capabilities by supporting customizable **classes** and **relationships**, allowing organizations to define their own metamodels using the classes and attributes best suited for their architecture. **Objects** in Atlas are instances of these classes, and can be connected with one another through **attributes**, which consist 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 (e.g., *Owner*).
(2) A **relation**: The type of relationship from one object to another, including both the direction from the origin to the destination and its inverse. Its name begins with a lowercase letter (e.g., *owns* and *owned by*).

To retrieve relevant data to display in views, Atlas includes the **Query Designer** canvas, which allows users to design **graphic queries**, used for extracting objects within the repository. Each repository contains its own distinct set of architectural elements, such as classes, views, and queries, and operates independently to reflect the specific characteristics of its architecture. Atlas also supports **scenarios**, which are isolated, editable views of the main repository, offering a safe environment for testing and experimentation without affecting the original data.

3.1 Atlas' Relational Data Model

To support data storage and retrieval, Atlas uses a complex relational schema composed of interconnected tables. Key tables include *t_object*, *t_property*, *t_base_type*, *t_object_relation*, and *t_query_type*, which track, respectively, object definitions, properties, relations,

Authors' addresses: Maria João Madeira Duarte, maria.j.duarte@tecnico.ulisboa.pt, Instituto Superior Técnico, Lisbon, Portugal; Pedro Manuel Moreira Vaz Antunes de Sousa, Instituto Superior Técnico, Lisbon, Portugal.

connections between objects, and queries. The tables *t_object_properties* and *t_object_relation* are associated with configuration containers, which reference either a repository or a scenario, through the *config_container_id* columns. Figure 1 shows the Entity-Relationship Diagram of Atlas' relational schema, highlighting the key tables and relationships for our work.

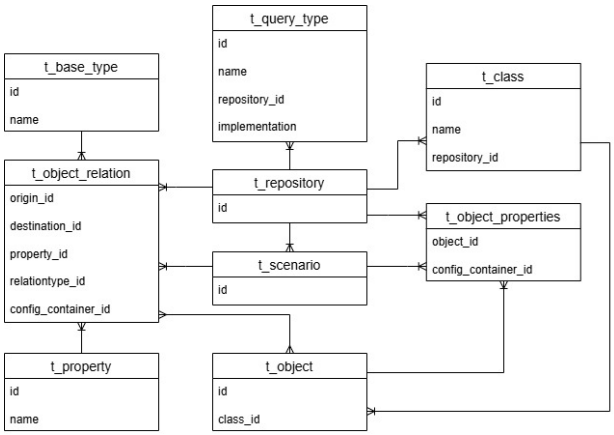

Fig. 1. Entity-Relationship Diagram of Atlas' Relational Schema.

3.2 Atlas' Graphic Querying System

Atlas' Query Designer allows users to build queries graphically using predefined symbols and a drag-and-drop canvas. It supports operations such as filtering, navigation, updates, and merging of object sets. Figure 2 illustrates a simple graphic query that retrieves the objects from the **class** *Business Actor* that have the **property** *Location* equal to Lisbon.

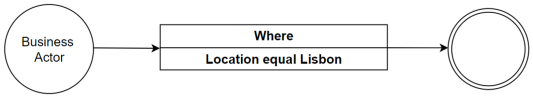

Fig. 2. Atlas' Graphic Query.

When a query is saved, its design is translated into an XML representation, which is then stored in the *implementation* column of the *t_query_type* table.

At execution time, this XML is interpreted by the backend Worker module to generate the corresponding executable database statements. This translation introduces inefficiencies and noticeable waiting times, and is what motivated the development of a more performant SQL-based query mechanism, presented in this thesis.

4 SOLUTION OVERVIEW

To incorporate direct SQL query input in Atlas, a solution comprising the following key components was developed:

(1) Designing a **virtual schema** to present an intuitive interface over Atlas' relational data model. This schema allows users to write SQL queries that reference familiar concepts, such as classes and attributes, as if they were traditional tables and columns. For example, users can write queries like `SELECT "Job Title" FROM "Business Actor"` without needing to understand the internal representation of those entities. In reality, classes like *Business Actor* are represented as entries 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 adapting 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:
      – **Nested queries**, allowing users to reference the result set of one query within another.
      – **Scenario verification**, so query results consider not only objects from the main repository but also those from loaded scenarios.

With this system, queries submitted in Atlas undergo the following transformation process:

```
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.
```

5 QUERY VALIDATION WITH JSQLPARSER

We adapted the **JSqlParser framework** [1] to validate user-written queries according to both standard SQL syntax and platform-specific rules. JSqlParser, built with JavaCC, parses SQL statements into a structured hierarchy of Java classes [3], allowing inspection and manipulation of query components.

A custom class, *QueryParser*, was developed to handle query validation. This class performs both **lexical** and **syntax analysis** using JSqlParser's `CCJSqlParserManager.parse` method. During **lexical analysis**, the query is tokenized, identifying SQL keywords, identifiers, and symbols. Invalid characters or unexpected tokens result in error messages.

Syntax analysis ensures that queries follow the expected structure. In our system, only `SELECT` statements are allowed, and queries are restricted to those represented by JSqlParser's **PlainSelect** class. This simplifies validation and prevents data modifications. The parser extracts key components, **SelectItem**, **FromItem**, and **Expression**, corresponding to the `SELECT`, `FROM`, and `WHERE` clauses.

Due to Atlas' flexible naming conventions, which allow spaces and special characters, the system requires all class, property, and query names in SQL to be enclosed in quotation marks. This rule is enforced throughout validation, and queries that do not comply are rejected with error messages.

Once validated, the extracted components are used in the translation process, transforming the user-written query into the Translated SQL Query, to match Atlas' relational schema. A separate custom class, *QueryHandler*, manages this transformation, beginning in the `interpreteQuery` method, described in later sections.

6 DESIGNING A VIRTUAL SCHEMA

In the virtual schema designed for our system, every **class** and **query** saved in Atlas is represented as a **virtual table**. These tables can optionally be prefixed with `"c="` or `"q="` to distinguish between classes and queries, and must be enclosed in quotation marks. Similarly, every **property** and **relation**, enclosed in quotation marks, is treated both as a **virtual column**, appearing in all virtual tables, and as a **virtual table** that can be queried directly.

The virtual schema is dynamic, based on the data stored in the Atlas database. Figure 3 illustrates how virtual tables are dynamically derived from the stored classes, queries, properties, and relations, while virtual columns are generated from the properties and relations.

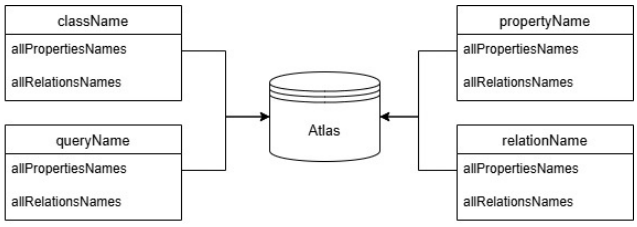

Fig. 3. Visualization of the Virtual Schema Dynamically Generated from Atlas' Data.

During query translation, references to virtual schema elements are mapped to Atlas' relational tables. For instance, class and query names in the `FROM` clause are translated against the *t_class* and *t_query_type* tables, respectively. Properties and relations in the `SELECT` and `WHERE` clauses are translated using joins across the tables *t_object_relation* and either *t_property* or *t_base_type*. Additionally, the translated queries can incorporate checks against the table *t_scenario* to include scenario objects.

7 DEFINING SQL-BASED QUERIES

The SQL-Based Query format follows a structured syntax that aligns with the Virtual Schema. It supports data retrieval through a simplified form of SQL, currently supporting the three main SQL clauses: `SELECT`, `FROM`, and `WHERE` clauses.

7.1 SELECT Clause

The system supports a single `SELECT` item per query. This can be:

(1) An **attribute name**, enclosed in quotation marks, referring to a property or relation connected to the dataset defined in the `FROM` clause. E.g.:

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

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

(2) An **asterisk (*)**, which returns the objects in the `FROM` clause directly. E.g.:

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

Retrieves all the objects from the *System Software* dataset.

Support for multiple `SELECT` items may be added in the future to enable selection of multiple connected attributes.

7.2 FROM Clause

The `FROM` clause accepts either a **class** or a previously defined **query**, treating both as virtual tables. Class names can be optionally prefixed with `"c="` and query names with `"q="`, and all names must be enclosed in quotation marks. The clause also supports **subqueries** and **union** operations.

Examples include:

(1)
```
SELECT * FROM "c=Business Actor";
```

Selects all objects from the *Business Actor* class. In the virtual schema, *"c=Business Actor"* is a virtual table.

(2)
```
SELECT * FROM "q=Employees in Lisbon";
```

Retrieves the result set of the previously defined query *Employees in Lisbon*. In the virtual schema, *"q=Employees in Lisbon"* is a virtual table.

(3)
```
SELECT "Components" FROM (SELECT "realizes"
    FROM "System Software");
```

Retrieves objects connected through *realizes*, then further connected through *Components*. In the virtual schema, *"Components"* is a column of the *"realizes"* table, and *"realizes"* is a column of the *"System Software"* table.

(4)
```
SELECT * FROM (SELECT * FROM "Business Actor
    " UNION SELECT * FROM "System Software")
    ;
```

Merges objects from the *Business Actor* and the *System Software* classes, using a `UNION` operator. In the virtual schema, both classes are virtual tables.

7.3 WHERE Clause

WHERE clauses are used to filter datasets based on their attributes. It supports:

(1) Conditions using **properties**, e.g.:

```
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.

(2) Conditions using **relations**, e.g.:

```
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.

(3) Multiple conditions using AND/OR operators, e.g.:

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

(4) Filtering using NULL/NOT NULL, e.g.:

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

8 INTRODUCING NESTED QUERIES

The graphic query language in Atlas does not support advanced features such as **nested queries**. To address this, the SQL-based system allows users to reuse defined queries as virtual tables in other queries.

This is achieved by referencing a previously defined query by its name in the FROM clause, 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";
```

9 INTRODUCING SCENARIO VERIFICATION

Another feature implemented in our system is **scenario verification**, which ensures that queries reflect the user's current working context by considering data from active scenarios. In Atlas, scenarios act as isolated views of the main repository, allowing users to modify data without affecting the original, thus providing a safe environment for testing and simulations.

Changes in the main repository propagate to its scenarios unless the objects have been modified or deleted within the scenario. In contrary, changes made in a scenario do not affect the main repository unless explicitly published.

When executing a SQL-Based Query, the system checks for an active scenario:

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

10 SQL-BASED QUERY PATTERNS

SQL-Based Queries were introduced in Atlas as an alternative to the graphical query language, giving users the flexibility to construct queries either visually or through SQL. This section highlights four SQL query patterns and their graphical equivalents.

10.1 Filtering by a Relation

The graphic query (Figure 4) demonstrates null attribute filtering. It selects all objects from the *Application Component* class that have the *released by* relation not empty.

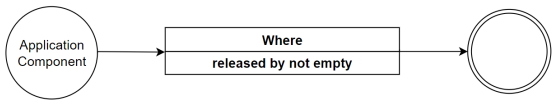

Fig. 4. Null Attribute Filtering in the Graphic Query Language.

The SQL equivalent is:

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

10.2 Subqueries

The graphic query (Figure 5) uses sequential navigation through two properties: it first retrieves objects connected through *Database Schema*, and then further navigates through *Technology Usage* to retrieve the final set of connected objects.

The SQL equivalent is:

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

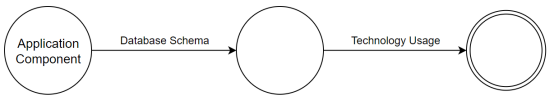

Fig. 5. Subqueries in the Graphic Query Language.

10.3 Union Operation

The graphic query (Figure 6) uses two parallel paths merged with a UNION operator. One path retrieves objects connected to the *Business Process* class through the *realizes* relation, while the other filters *Business Service* objects where the property *User* equals Link Consulting.

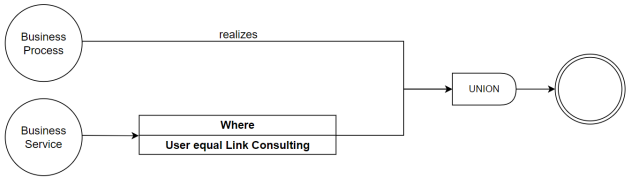

Fig. 6. Union Operator in the Graphic Query Language.

The SQL equivalent is:

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

10.4 Subqueries and Filtering

The graphic query (Figure 7) combines filtering and navigation. It selects objects connected to a filtered subset of *Application Component* objects through the *composed of* relation, and applies additional filtering conditions on the resulting set:

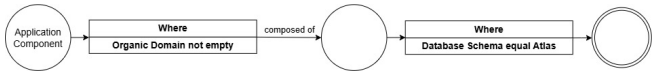

Fig. 7. Subqueries in the Graphic Query Language.

The SQL equivalent is:

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

11 STORING TRANSLATED SQL QUERIES

To improve execution performance of SQL-Based Queries, a new column named **output_query** was added to the *t_query_type* table. This column stores the executable translation generated by our system, named **Translated SQL Query**.

With this addition, when the user executes a query, its 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 execution time, which is more frequent and impactful for user experience.

This mechanism also supports **nested queries**. When translating a query that references another saved query, the system retrieves the inner query's pre-translated statement from the *output_query* column and wraps it in a Common Table Expression (CTE). The outer query is then translated around this structure and stored.

The pseudocode below illustrates how the system handles query creation and query execution:

(1) A user creates a query named *inner query* that retrieves data from a class <CLASS_NAME> based on a condition <CONDITION_1>.

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

(2) The query is translated into its Translated SQL Query, <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 <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 outer 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>;
```

12 MAPPING SQL-BASED QUERIES TO THE RELATIONAL SCHEMA

After validation, the SQL-Based Query is translated into the Translated SQL Query that aligns with Atlas' relational schema. This process is handled by two custom classes: the *QueryHandler*, which processes the FROM clause and constructs the statement to be wrapped in the CTE, and the *QueryBuilder*, which processes the SELECT and WHERE clauses and combines all transformations to produce the Translated SQL Query.

12.1 Processing the FROM Clause

The *QueryHandler* class interprets the `FROM` clause using the `interpreteQuery` method, which determines whether it refers to a class, a query, a `UNION` operation, or a subquery:

- If the name refers to a **class** (or has the `"c="` prefix), the class Identifier (ID) is retrieved and used to generate a query selecting all objects from that class.
- If the name refers to another **query** (or has the `"q="` prefix), the query ID is retrieved to return the inner query's pre-translated query from its *output_query* column.
- `UNION` and subqueries are recursively processed and wrapped in CTEs.

The result of this step becomes the base CTE for building the final Translated SQL Query.

12.2 Processing the SELECT and WHERE Clauses

The *QueryBuilder* class processes the rest of the query starting in its `build` method. It interprets:

12.2.1 SELECT Clause. The custom `handleSelectClause` method identifies whether the selected attribute is a property or a relation based on naming conventions (uppercase for properties, lowercase for relations):

- If it is a **property**, it uses the *t_property* and the *t_object_relation* tables to navigate to the objects connected to the ones in the base CTE through that property.
- If it is a **relation**, it uses the *t_base_type* and the *t_object_relation* tables to navigate to the objects connected to the ones in the base CTE through that relation.

If the clause contains an **asterisk**, the system selects all objects from the base CTE.

12.2.2 WHERE Clause. The custom `handleWhereClause` method processes logical expressions using stacks for operators and conditions. Each condition is translated according to whether it is:

- An **equality** condition: mapped similarly to the `SELECT` clause. It retrieves the objects where the specified attribute matches the given value.
- **Null** filtering (`IS NULL/IS NOT NULL`): determines whether the attribute is a property or a relation and retrieves the objects that either have or do not have a connection through that attribute.
- `AND/OR` predicate: separates its left and right expressions using JSqlParser methods, and combines them according to operator precedence.

The translated conditions are combined with the previously generated CTE to form the final Translated SQL Query, which is stored in the *output_query* column, while the original user-written query is stored in the *implementation* column of the *t_query_type* table.

12.3 Incorporating Scenario Verification

To support **scenario inclusion**, the system adapts queries based on the user's active environment, which can either be the main repository or a loaded scenario. In Atlas, objects edited in the main repository or in a scenario are distinguished using the *config_container_id* column of the *t_object_properties* and the *t_object_relation* tables, which links to either a repository or a scenario ID.

To support this functionality, all translated queries can be converted to a **SQL Query for Scenario Objects**, which is constructed using two CTEs:

- **REP**, that retrieves the objects edited in the **repository** that satisfy the 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>*.

To avoid returning outdated repository objects that have been changed in the scenario, the final query excludes the **REP** objects whose relevant attributes were overridden, and then performs a `UNION` with the objects from **SCEN**, as shown in the following pseudocode:

```
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;
```

At execution time, the system receives the *config_container_id* of the active environment and compares it with the main repository ID. If they match, the original Translated SQL Query is executed. If they differ, the SQL Query for Scenario Objects is constructed and executed instead. This is implemented in the custom `executeQuery` method.

13 TRANSLATED SQL QUERIES

This section illustrates how user-written SQL-Based Queries are translated into executable queries over Atlas' relational schema. The examples demonstrate how the system handles different patterns and scenario execution.

13.1 Selecting Objects from a Class

The following SQL-Based Query retrieves all objects from the class *Application Component*:

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

This query is translated into a Translated SQL Query that uses the class ID and repository ID to retrieve the corresponding objects:

```
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;
```

13.2 Selecting Objects from a Query

The following SQL-Based Query retrieves the result set of a previously defined query:

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

This query is translated into a Translated SQL Query by wrapping the inner query's pre-translated query in a new CTE:

```
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;
```

13.3 Filtering by a Relation

The following SQL-Based Query filters objects from a class according to a relation value:

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

This query is translated into a Translated SQL Query that uses the *t_base_type* and the *t_object_relation* tables to process the relation:

```
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")
    ;
```

13.4 Scenario Objects

When a scenario is loaded, queries are automatically adapted to return the objects that exist within the scenario. For example:

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

This query's SQL Query for Scenario Objects uses two CTEs: one for repository objects (`REP`) and one for scenario objects (`SCEN`) that uses the loaded scenario's ID. These are merged with a `UNION` operation, excluding overwritten repository 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;
```

14 SYSTEM ARCHITECTURE

The system is implemented in Java using the **Spring Boot** framework. SQL-Based Queries are parsed and validated using **JSqlParser**, after which a custom translation module maps virtual table and column references to Atlas' relational schema.

During query translation and execution, auxiliary queries are executed to retrieve class and query IDs or Translated SQL Queries. Since these follow fixed SQL patterns and differ only in parameter values, we use **PreparedStatement** from the Java SQL package. This allows the SQL to be precompiled with placeholders, allowing the reuse with different arguments without recompilation [4]. Additionally, PreparedStatement benefits from Hibernate's First-Level Cache, reducing query execution time [5].

The final Translated SQL Query is a plain SQL string with no placeholders and is executed using the **createNativeQuery** method provided by Hibernate's **EntityManager**. This approach allows Hibernate to automatically map the result set to the corresponding Entity class [6]. 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).

15 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 well-defined rules and do not introduce any unauthorized SQL operations besides the expected SELECT statement.
(2) Using **PreparedStatement** at early stages of query translation which validates the string literals against SQL injection [4].

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

16 USER INTERFACE INTEGRATION

SQL-Based Queries were integrated into the Atlas platform by extending the existing Query Explorer Menu, maintaining familiar workflows while allowing both graphical and SQL query creation.

16.1 Query Creation and Editing

Users can create new queries by clicking the + button in the Query Explorer Menu, which opens a menu to enter the query name and additional information. After saving, the Query Editing Menu appears in a modal window, where users can input the query implementation. Queries can be created or edited either graphically, via the Designer canvas, or through SQL-Based Queries, using the shared implementation text box.

The implementation text box supports only one type of query at a time: either a graphic query's XML translation or a user-written SQL query. After modeling a graphic query and saving it, the Worker module translates the design into an XML representation, which is stored in the *implementation* column of the *t_query_type* table and shown in the implementation text box (Figure 8).

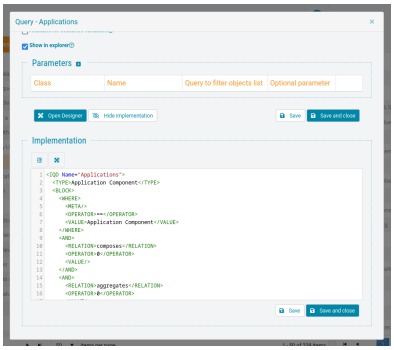

Fig. 8. Atlas' Implementation Text Box - XML Representation.

Now, in addition to displaying the XML code, users can write SQL queries directly in the same implementation text box (Figure 9). When saving a SQL query, the system validates it, stores it in the *implementation* column, and generates its corresponding Translated SQL Query, which is saved in the *output_query* column.

Because the text box can only support one query type at a time, inputting an SQL query will replace any previously stored XML, and switching back to the graphical interface will overwrite the SQL input with the updated XML translation.

16.2 Query Execution

To execute a query, the user selects it from the list. If the implementation starts with a `SELECT` statement, the system retrieves the corresponding pre-translated SQL query and executes it. Otherwise, the implementation corresponds to a graphical query and is interpreted by the Worker module, which translates the XML code into

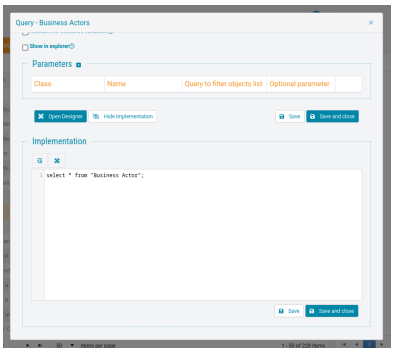

Fig. 9. Atlas' Implementation Text Box - SQL Query.

executable statements. The query results are then displayed in the right panel of the interface.

16.3 Scenario Verification

Users can activate scenarios through the UI's top bar. When a scenario is active, all queries are executed within that scenario's context. For SQL-Based Queries, the system receives the scenario ID along with the request and constructs a SQL Query for Scenario Objects accordingly. Once executed, the scenario results are displayed in the interface as usual.

17 EVALUATION

Given that the project is intended for commercial use within a product for clients, efficiency is a critical concern. Thus, the implemented SQL-Based Query system was evaluated using four key criteria: accuracy, performance, robustness, and effort.

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, and averaged. This reduced variability caused by the processor and background system activity. SQL queries were always generated before their graphical equivalents to avoid caching biases for SQL measurements. Additional tests for scenario and nested queries were executed afterward, which may slightly benefit from cached objects.

17.1 Performance Comparison

We constructed 70 pairs of equivalent graphic and SQL queries. Results showed:

- **Construction Time:** Graphic queries took on average 35.89 s to build, while SQL queries averaged 22.97 s—showing SQL was 36% faster to construct.
- **Saving Time:** Saving a graphic query averaged 51.7 ms, while SQL queries took 70.4 ms (36% slower), due to added parsing and translation overhead.
- **Generation Time:** SQL-Based Queries significantly outperformed graphical queries with an average generation time of 113.7 ms versus 3,079.3 ms—an improvement of 96.3%. SQL queries were up to 27× faster globally and 69× faster per query on average.

17.2 Scenario Queries

All SQL-Based Queries were evaluated in both repository and scenario contexts. When executed within a scenario, the system generated the SQL Query for Scenario Objects by adjusting the *config_container_id* and merging scenario and repository data. These scenario queries averaged 115.1 ms in generation time, only 1.2% slower than standard SQL queries.

17.3 Nested Queries

For nested queries, we recorded a mean **saving time** of 79.3 ms and a mean **generation time** of 109.3 ms. These results demonstrate that implementing support for nested queries did not introduce significant waiting time. The saving time is approximately 12.7% slower compared to standard queries, which aligns with expectations, since saving a nested query requires an additional database access to retrieve the pre-translated query of the referenced inner query.

On the other hand, the mean generation time for nested queries was 3.9% faster. Given that the generation process is identical for both nested and non-nested queries, this slight improvement is likely influenced by caching effects.

17.4 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.

17.5 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, with all failures attributable to user input errors.

17.6 Effort

SQL-Based Queries were easier and faster to construct for experienced users. On average, SQL query creation was 36% faster than using the graphical interface, which involves more manual interaction with a drag-and-drop canvas.

18 CONCLUSION

18.1 Query Structure, Data Volume, and Indexes

As part of our experimentation, we evaluated an alternative approach that **directly modified** the inner query structure instead of using CTEs. However, this method resulted in lengthy queries with multiple JOIN operations and late filtering, resulting in costly query execution, due to the processing of large amounts of unnecessary data.

However, **CTEs** provided performance benefits by reducing row volume at early stages in execution and simplifying complex query logic into more manageable building blocks, facilitating query interpretation and optimization.

We also investigated the the possibility of using **materialized views** or **temporary tables** and concluded that they would not offer advantages in our context. Given the dynamic nature of Atlas data, which is modified frequently, storing intermediate results in a database would be impractical, risking displaying outdated results.

Additionally, we investigated the use of **Statement** from Java Database Connectivity (JDBC) 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 [7].

Finally, we ensured that all columns used in JOIN conditions or filters were properly indexed, and we benefited from CTEs inheriting indexes from the underlying tables, improving query performance [8].

18.2 Overall Conclusions

The introduction of `"c="` and `"q="` **prefixes** in the `FROM` clause to distinguish between classes and queries removed the necessity for extra database accesses to verify whether a name corresponds to a class or a query, significantly improving **query saving** times.

Since searching for the ID of a class or query according to its name is an expensive operation, retrieving the IDs during the translation phase allowed the Translated SQL Query to reference them directly, eliminating the need for this costly search during query execution.

Our approach prioritized a fast query generation over query saving time, as execution has a greater impact on user experience. This requirement was satisfied giving that the system achieved a 96% improvement in generation time, with only a 36% increase in saving time, which remains acceptable.

19 LIMITATIONS AND FUTURE DEVELOPMENTS

A key limitation of the current system is that 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 later edited, those changes do not propagate to the dependent queries unless the user re-saves them, leading to outdated results.

To address this, two solutions are proposed:

(1) Instead of including the translated inner query in the CTE, we could save a **PreparedStatement** with a placeholder inside the CTE. Then, during execution, we would retrieve the current translation of the inner query and substitute it into the placeholder.
(2) Using the existing Atlas' **batch processing** functionality to periodically identify modified queries and automatically refresh the translations of any dependent queries.

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

(1) Introducing a **dedicated text box** for SQL input to prevent accidental overwriting by graphic query XML.

(2) Implementing a **feedback mechanism** to assist in detecting syntax errors, providing insight into which specific clause failed validation.
(3) Implementing a mechanism to populate the implementation text box with **query templates** containing empty clauses, with dropdown lists to select existing classes, queries, properties and relations.
(4) Once the user-written query is validated, iniciate the query translation and saving processes **asynchronously** in the background, to reduce the perceived saving time.

Although SQL Queries for Scenario Objects are not currently stored in the database due to not being used regularly, if that requirement changes, it is viable to create a **new column** in the *t_query_type* table to store scenario related query translations.

Lastly, future work could explore translating graphical queries directly into SQL-Based Queries, eliminating the inefficient XML representation. This would combine the user-friendly advantages of graphic query design with the performance benefits of the SQL-based execution model.

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

REFERENCES

[1] L. Francalanci and R. Manning, "Jsqlparser," accessed: 2024-05-31. [Online]. Available: https://github.com/JSQLParser/JSqlParser
[2] 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/
[3] L. Francalanci and R. Manning, "Java sql parser library," 2024, accessed: 2024-10-15. [Online]. Available: https://jsqlparser.github.io/JSqlParser/index.html
[4] Oracle, "Using prepared statements," 2024, accessed: 2025-04-04. [Online]. Available: https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html
[5] Pankaj, "Hibernate caching - first level cache," 2022, accessed: 2025-02-01. [Online]. Available: https://www.digitalocean.com/community/tutorials/hibernate-caching-first-level-cache
[6] 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)
[7] GeeksforGeeks. (2025) Hibernate native sql query with example. Accessed: 2025-05-01. [Online]. Available: https://www.geeksforgeeks.org/hibernate-native-sql-query-with-example/
[8] 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/