TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Edvard Paas, 221490IABM

Terry, a Datalog to SQL Interpreter

Master's thesis

Supervisor: Sadok Ben Yahia, PhD
Bruno Rucy Carneiro Alves De Lima, MSc

Tallinn 2024

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Edvard Paas, 221490IABM

Terry, Datalog-SQL interpretaator

Magistritöö

Juhendaja: Sadok Ben Yahia,
Professor
Bruno Rucy Carneiro
Alves De Lima, MSc

Tallinn 2024

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Edvard Paas

12.05.2024

Abstract

This thesis focuses on the development and benchmarking of a Datalog to SQL interpreter, designed to integrate the logic programming language Datalog with SQL, the predominant language used in relational database management systems. Datalog is used for its ability to succinctly express complex recursive queries, which are typically verbose and complicated in SQL. By integrating Datalog with SQL, this work aims to leverage the data manipulation capabilities of SQL with the advantages offered by Datalog, facilitating more efficient and expressive data querying. The primary aim of this master's thesis is to develop a Datalog to SQL interpreter and to subsequently assess its practicality and performance across various database engines. The interpreter is benchmarked to evaluate its efficiency in translating Datalog queries into SQL.

This thesis is written in English and is 42 pages long, including 6 chapters, 13 figures and 1 table.

Annotatsioon

Terry, Datalog-SQL interpretaator

See lõputöö keskendub loogilise programmeerimiskeele Datalog integreerimiseks SQL-iga, mida valdavalt kasutatakse relatsioonilistes andmebaasihaldussüsteemides, läbi Datalog-SQL interpretaatori arendamise ja jõudluse testimise. Datalogi kasutatakse selle võime tõttu lühidalt väljendada keerulisi rekursiivseid päringuid, mis on SQL-is tavaliselt keerulised. Integreerides Datalogi SQL-iga, on selle töö eesmärk kasutada SQL-i andmetega manipuleerimise võimalusi koos Datalogi pakutavate eelistega, hõlbustades tõhusamat ja väljendusrikkamat andmepäringut. Selle magistritöö esmane eesmärk on välja töötada Datalog to SQL interpretaator ning seejärel hinnata selle praktilisust ja toimivust erinevates andmebaasimootorites. Osa tööst on ka uurida interpretaatori jõudlust, et hinnata selle tõhusust Datalogi päringute SQL-i käivitamisel.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 42 leheküljel, 6 peatükki, 13 joonist, 1 tabelit.

List of abbreviations and terms

RDBMS	Relational database management system
SQL	Structured Query Language
ACID	Atomicity, consistency, isolation, durability
IDB	Intensional database
EDB	Extensional database
LAMP	Linux, Apache, MySQL, PHP
CTE	Common Table Expression
PyPi	Python Package Index
ORM	Object-Relational Mapping
SPJ	Select-Project-Join
vCPU	Virtual Central Processing Unit
RDF	Resource Description Framework
KDE	Kernel density estimation
DDL	Data Definition Language

Table of contents

List of figures

List of tables

1 Introduction

Datalog, as a declarative logic programming language, has regained prominence in the realm of database querying and programming languages due to its expressive power and simplicity. The relevance of Datalog extends beyond traditional database queries; its ability to efficiently handle recursive queries and perform complex pattern matching makes it invaluable in areas requiring sophisticated data manipulation and inference capabilities, such as artificial intelligence [1], knowledge representation, and data integration [2].

The utility of Datalog manifests across various use cases: in the realm of big data, Datalog is used for large-scale data processing [3], leveraging its recursive capabilities to navigate and analyse deeply nested or hierarchical data structures like social networks [4], genealogical trees [5]. Furthermore, in cybersecurity, Datalog helps in the formulation and enforcement of security policies and access control decisions [6], where its rules can succinctly express complex conditions and relationships among data entities.

Despite its strengths, there remains a gap in the integration of Datalog into mainstream application development, primarily due to the lack of robust tools that can translate Datalog's theoretical advantages into practical, deployable solutions in standard relational database management systems (RDBMS). Many existing systems either do not support recursive queries natively or do so with significant limitations. This demonstrates the need for an interface between classical relational database systems and a logical programming language with an accessible recursion syntax such as Datalog. Developing an interpreter that can efficiently translate Datalog queries into optimized SQL should broaden Datalog's acceptance in the industry. Addressing this gap forms the core challenge for this research.

1.1 Research Questions

This thesis aims to answer two research questions:

1. **How would the interpreter's performance be affected by different database engines, including SQLite, PostgreSQL, MySQL, and DuckDB?** This aspect is important for understanding how different database engines handle expected loads from a Datalog program.

2. **How efficient would such a Python implementation be compared to other languages?** In order to gauge whether a solution should instead be developed in a different programming language, it would be valuable to analyse the efficiency of the current solution in terms of estimated energy consumption.

1.2 Contributions

Our contribution to this field with the development of Terry includes:

1. Implementing a Datalog to SQL interpreter in Python, including a Datalog parser, rule evaluator and program materializer.

2. Added integration for several popular databases like SQLite, PostgreSQL, MySQL, DuckDB.

3. The modular design which allows future integration of other SQL database engines in the future.

4. Benchmarks empirically evaluate Terry's effectiveness and performance; I conducted comprehensive benchmarks. These tests are designed to measure the interpreter's execution speed when operating across the supported databases under various datasets and query complexities.

Section 2 gives a background overview of the theory behind relational databases and SQL, Datalog and deductive databases, and the concepts behind interpreters and translating Datalog to SQL. Section 3 describes the conceptual background behind Terry, the software architecture and implementation. Section 4 explains the datasets and Datalog programs used in the benchmarks, and discusses its results. Section 5 draws conclusions based on the completed work.

2 Background

This section offers a brief theoretical overview of the three fundamental pillars that this work relies on: the relational model and the SQL databases that implement it, Datalog and the concepts behind interpreting Datalog into SQL. The aim is to introduce the ideas which are integral to understanding how Terry functions.

In the spirit of transparency, we are disclosing that AI assistance was used in this chapter. Its role was to better our understanding of the concepts necessary for grasping the thesis, as well as to partially aid in improving the writing process.

2.1 Relational Databases

Relational algebra is a theory that defines an algebraic structure to model data. It can be seen as a set of rules to manipulate relations, which are sets of tuples S

$$S = \{(s_{i1}, s_{i2}, \dots, s_{in} | \ i \in 1 \dots m\}$$

with a fixed-arity of n and containing m number of rows.

Relational algebra defines operations that are used to manipulate data, such as selection, projection, cartesian product, union, etc. However, only the relevant operations will be introduced.

Selection is used to filter rows from a relation based on a specified condition. Given a relation R, and a condition ϕ (for example, $A = 5$), selection operation selects all tuples t in R where ϕ holds true:

$$\sigma_{A=5}(R) = \{t \mid t \in R \ \wedge t.A = 5\}$$

This selection corresponds to a SQL `SELECT * FROM R WHERE A=5` statement.

Projection is used to select certain columns from a relation, reducing the number of attributes in the result set to only those specified. It is denoted by $\Pi_{a_1,\ldots,a_n}(R)$, where $a_1, \ldots, a_n$ is a list of attributes kept in the result. The result of a projection is a relation that contains tuples stripped of any attributes not specified in the project list, e.g.,

$$\Pi_{A,B}(R)$$

produces a new relation with only the columns A and B from the relation R. An analogical statement in SQL is `SELECT A, B FROM R`.

Equijoin (**Join**) is a join operation that combines tuples from two relations based on equality conditions between specified attributes of those relations. Equijoin merges rows from two relations, when the specified attributes have matching values. The syntax can be expressed as

$$R \bowtie_{A=B} S$$

where R and S are relations, and A and B are attribute names from R and S respectively. The condition $A = B$ specifies that only those tuples in the resulting relation should be included where the value of attribute A in relation R matches the value of attribute B in relation S.

Relational database is a database that uses the relational model to describe data as named relations of labelled values. For example, a relation to describe a graph's set of edges could be written as

$$Edge: \{< (SrcNode, 1), (DstNode, 2) >, < (SrcNode, 2), (DstNode, 3) >\}$$

where the relation's name is $Edge$, commonly referred to as a table, and tuples of labelled value pairs $< (SrcNode, 1)$ ... $>$ are seen as rows of a table. [7].

SQL (Structured Query Language) is the standard language for interacting with relational databases [8]. SQL dialects refer to variations of the SQL language that are adapted and extended by different database systems like MySQL, PostgreSQL, SQLite or DuckDB. Each dialect has its unique features and syntax nuances, which makes the goal of creating a compatible interpreter more challenging.

2.1.1 Embedded Databases

An embedded database is integrated into the application that uses it, rather than being a standalone service. The demand for embedded databases has grown significantly since the rise in popularity of embedded devices, whose applications tend to be of limited scope and do not require an enterprise-level relational database to manage its data [9]. Many applications are not complex and require only a single database to satisfy their storage needs [10]. This kind of database has the benefit of easier deployment, as it typically utilizes the device's memory or disk space for storage.

SQLite is a library, which implements a self-contained SQL database engine. The lack of having to manage a dedicated connection to a separate, possibly remote, process makes it easier to deploy, as the database instance exists within the same memory space as the application. [11] It also offers exceptional portability, as the only requirements for its use are a couple of routines from the standard C library.

DuckDB is an embedded database meant for analytical purposes. It is similar to SQLite, except that it uses a columnar query execution engine. This greatly reduces overhead present in traditional systems such as SQLite which process each row sequentially [12].

2.1.2 Client-Server Architecture Databases

PostgreSQL is an open-source object-relational database system that extends SQL with advanced features for handling complex data workloads. Originating from the POSTGRES project at the University of California, Berkeley in 1986, it has evolved over 35 years into a highly respected platform known for its architecture, reliability, and rich feature set including ACID compliance since 2001 and third-party extensions. PostgreSQL is also notable for its broad compliance with the SQL standard, conforming to 170 out of 179 mandatory features [13].

MySQL is an open-source relational database management system (RDBMS) that has grown to be one of the most popular RDBMSs [14] widely used in web applications, particularly as a result of having been included in the LAMP (Linux, Apache, MySQL, PHP/Python/Perl) stack.

2.2 Datalog

Datalog is a declarative logical programming language. It is a Turing incomplete subset of Prolog. Datalog is known for its simplicity and expressiveness, particularly in representing complex queries involving recursion, which is a challenge in many traditional database systems [15].

A Datalog **program** is a list of rules. **Rules** are Horn clauses in the form of

$$H(x, y) \leftarrow B(x, y)$$

where H, B are relations (predicates). H is also the head of the rule and is part of the intensional database (IDB), while b is the body of the rule, and is part of the extensional database (EDB). The arguments of the relations are **terms**, which can either be **variable** or **constant**, in this case both are variable. **Facts** are derived from evaluating rules. Facts in themselves are rules without a body.

Datalog has three main definitions for the semantics of its programs: model-theoretic, proof-theoretic and fixpoint semantics. This work focuses on **fixpoint semantics** because bottom-up evaluation is based directly on it [7]. Fixpoint semantics involves using an immediate consequence operator, which is a function that applies rules to derive new facts based on the conditions defined within those rules. The operator can be used to derive new data that conforms to the Datalog program's logic. Recursive programs may be applied multiple times to accumulate results, before reaching the least fixpoint, meaning that no new facts can be derived.

Bottom-up evaluation in Datalog is a strategy that repeatedly applies the immediate consequence operator starting from the base data (initial facts). This approach iteratively computes the results from the ground up until a point is reached where no new data emerges, indicating that a fixpoint has been reached. The **naïve** version of this strategy involves a lot of redundancy as it recomputes all facts, including those derived in previous steps, while the **semi-naïve** approach optimizes this by only considering new facts produced in the most recent step.

For example, computing the transitive closure using the Datalog program:

$$T(x, y) \leftarrow E(x, y)$$

$$T(x, y) \leftarrow T(x, z), E(z, y)$$

We initialize relation T with the edges from E with $T^0 \leftarrow E$, thus evaluating the first rule. To evaluate the $(i + 1)$ iteration, we compute the following:

$$T^{i+1} \leftarrow \Pi_{x,y}(T^i \bowtie E)$$

The computation reaches the fixpoint when $T^{i+1} = T^i$, which means that no new facts were derived last iteration. In a semi-naïve implementation, we maintain a delta relation $\Delta T^i = T^i - \Delta T^{i-1}$ where we store the results of the i-th iteration only. In this case, we the $(i + 1)$ iteration is evaluated as $T^{i+1} \leftarrow \Pi_{x,y}(\Delta T^i \bowtie E)$ [16].

A Datalog reasoner has three primary functions:

1. Parsing. The reasoner takes a Datalog program as input, which consists of a set of rules and possibly some facts. The reasoner parses these rules to understand the logical implications they represent.

2. Rule evaluation. The core functionality of a Datalog reasoner is to evaluate these rules. It applies logical inference based on the rules provided to derive new facts from given facts. This is often done using various evaluation strategies like naive, semi-naive, or magic sets, depending on the implementation and the specific requirements of the application.

3. Query resolution. In addition to rule evaluation, a Datalog reasoner can answer queries about the data. These queries are answered by checking whether data derived or existing within the database satisfies the query conditions.

2.3 SQL to Datalog Translation

High-level programming language source code needs to be translated into machine language that the computer's processor can understand. This translation can happen in two main ways: through compilation or through interpretation.

An **interpreter** is a program that directly executes instructions written in a programming language without requiring them to have been compiled into a machine language program.

An interpreter parses the source code of a program written in a high-level programming language statement by statement. Interpreted source code is platform independent, assuming that the interpreter itself is compiled for the target platform.

Writing recursive queries in SQL, particularly with extensions such as Common Table Expressions (CTEs), can be complex and verbose as it requires explicit definition of both the anchor and recursive parts within the CTE. SQL's recursive capabilities are not as inherently integrated as in Datalog, leading to potential errors and cumbersome constructions in complex recursion scenarios. In contrast, Datalog naturally supports recursion through its syntax and semantics, allowing recursive queries to be expressed more succinctly and declaratively. Datalog's execution engines are specifically optimized for recursive operations, potentially offering more efficient query processing than SQL databases where recursion is not a primary design focus.

Terry is a Datalog to SQL interpreter that translates Datalog programs into SQL statements that can be executed on traditional relational database management systems. This translation allows the powerful, declarative querying capabilities of Datalog to be applied to structured data stored in SQL databases. Section 4 presents its architecture and implementation in detail.

In general, the first step in interpreting Datalog into SQL is parsing the input Datalog program. This involves breaking down the logical rules and queries written in Datalog into a format that the interpreter can understand. This process checks the syntax of the Datalog program for correctness and then constructs an internal representation of the program.

Once parsed, the program needs to be evaluated in the correct order, which requires sorting its rules based on a dependency analysis. The executed rules must be translated into an intermediary format that has a close relationship with SQL, and this thesis utilizes the Select-Project-Join (SPJ)-stack to achieve that. Once an instruction stack based on the rule is created, the relational algebra instructions can be easily interpreted in the context of SQL. Afterwards, the results of the evaluation are materialized by the interpreter, which completes one iteration of the program.

An interpreter that could transform a Datalog program to a SQL query would allow us to make use of complex recursive queries without having to deal with the verbosity of such queries in SQL.

Python is an interpreted high-level programming language, where the emphasis is put on having readable, and therefore more maintainable, source code [17]. Due to this, Python has been in the recent years been considered to be the most popular programming language [18], and therefore has a large developer community and a diverse selection of libraries and applications available through its Python Package Index (PyPi).

SQLAlchemy is an open-source SQL toolkit and Object-Relational Mapping (ORM) system for Python, designed to facilitate high-level interaction between Python programs and databases using SQL language constructs. The library abstracts common database interactions, allowing developers to write Python code instead of SQL to create, read, update, and delete data and schemas in their database. It supports multiple database engines, including PostgreSQL, MySQL, and SQLite, providing a consistent interface to manage database operations across different types of databases. [19].

SQLGlot is an open-source Python library designed to parse and transform SQL into various dialects. It enables the conversion of SQL queries from one dialect to another, supporting multiple database platforms like PostgreSQL, MySQL, and SQLite, thus facilitating cross-database compatibility and migration efforts. Additionally, SQLGlot offers features for optimizing and generating SQL queries [20].

3 Terry

Terry owes its name to P. D. Terry, who pitched a comprehensive way to envision interpretation as a syntax-directed processing unit [21]. A core feature of Terry, rule evaluation, uses a similar approach to parse Datalog rules into an intermediate form of an SPJ stack, which is then translated into SQL statements. The source code is available at the author's GitHub page: https://github.com/edvardpaas/pyterry.

3.1 Architecture

The interpreter utilizes a strategy of bottom-up evaluation, which means iteratively applying the rules until no more new facts are derived. As the interpreter utilizes semi-naïve evaluation, rule execution happens using all the facts produced so far. For example, computing the transitive closure using the Datalog program [16]:

```
T(x, y) <- E(x, y)
T(x, y) <- T(x, z), E(z, y)
```

We would initialize relation T with the edges from E with $T^0 \leftarrow E$, thus evaluating the nonrecursive program. To compute the $(i + 1)$ iteration, we compute the following:

$$T^{i+1} \leftarrow \pi_{x,y}(T^i \bowtie E)$$

The computation reaches the fixpoint when $T^{i+1} = T^i$, which means that no new facts were derived last iteration [16].

The language chosen to implement the compiler is Python (3.12), as it possessed the best selection of libraries and documentation to make the development faster. Namely, a library called SQLGlot allows easily to build SQL statements and specify a dialect to output them in. The wide selection of dialects compared to other similar engines in other languages made it the most attractive module to use. [20]

The code was developed using test-driven development and object-oriented design principles. The interpreter is sectioned into classes that have minimal responsibilities, and different database engines are covered by integration tests.

This section describes the codebase, including the documentation and unit tests of the Naïve compiler. It consists of the Datalog library, relational algebra instruction stack, Datalog rule evaluator, Datalog program splitting and sorting, and the Naïve compiler class. The code is deliberately commented to be self-documenting alongside the unit tests.

Terry's functionality can be abstracted into three primary components: Datalog Parser, Program Materializer, and the Rule Evaluator. Terry makes use of a shared connection pool between all its components, all of which need to interact with the SQL database. This section will provide an overview for each component. See Figure 1 for the general architecture of the interpreter.

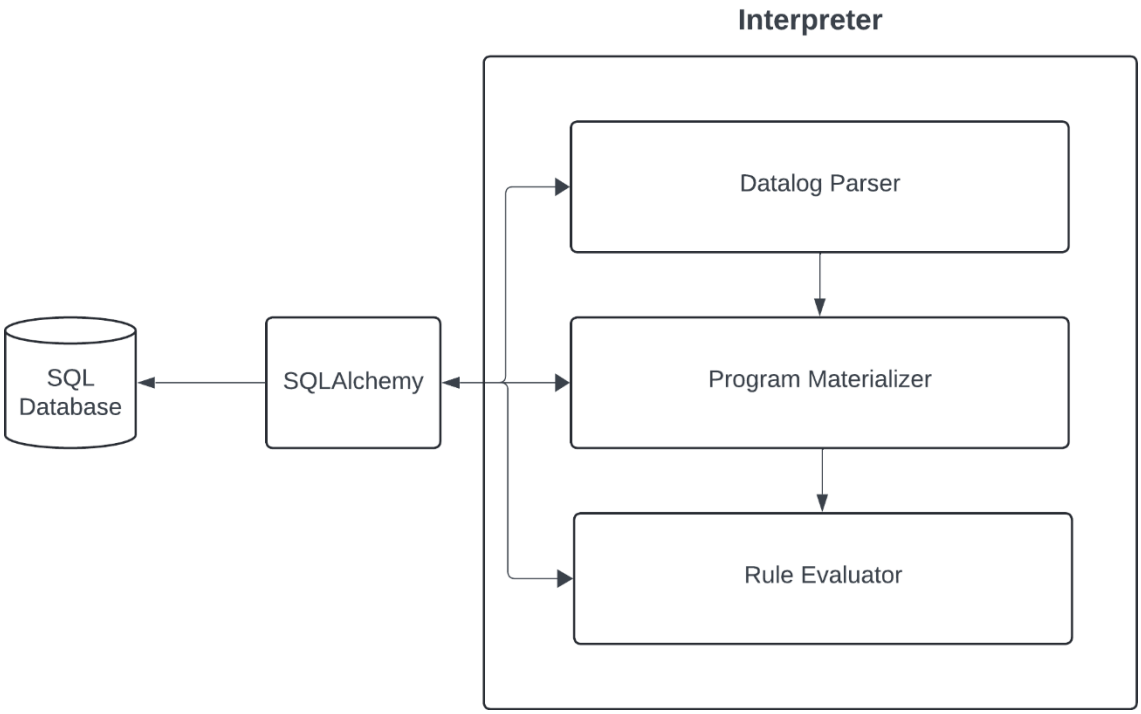

Figure 1. Architecture of Terry

3.2 Datalog Parser

The aim of the parser is to take an input in the form of a Datalog program, process it and prepare the database for materializing newly derived facts and rule evaluation (activity diagram of the algorithm on Figure 2). It achieves this by splitting the program into nonrecursive and recursive parts and deriving delta programs from them, sorting the

recursive delta program, establishing the connection with the database, and initializing it with delta relation and evaluation result tables.

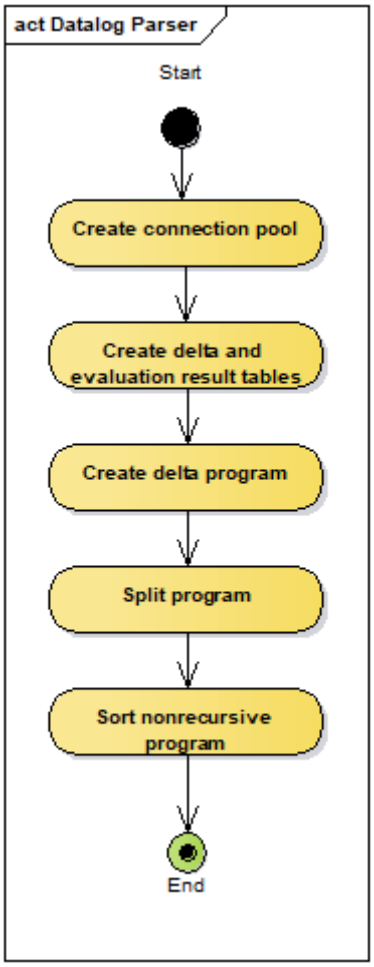

Figure 2. Datalog Parser Activity Diagram

In order to parse a Datalog program, we need a way to operate with Datalog concepts in the interpreter. This was accomplished by creating a library, derived from Micro Datalog, which provides a structured and type-safe way to represent and manipulate Datalog constructs: programs, rules, atoms and terms. The class model of the Datalog library is depicted by Figure 3.

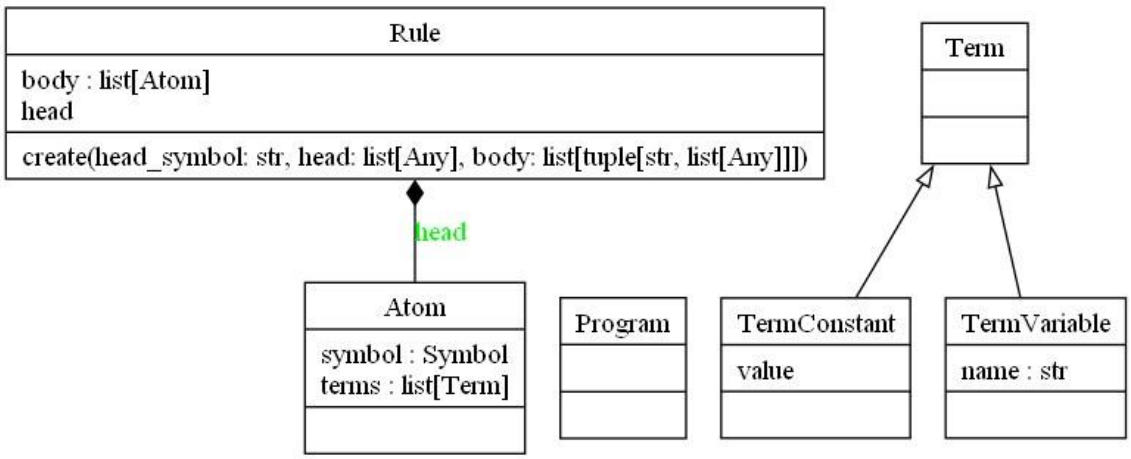

Figure 3. Datalog Library Class Model

The connection pool is created using SQLAlchemy, and this is propagated to rest of the interpreter's components. The interpreter allows the user to choose the database engine, and currently supports SQLite, DuckDB, Postgres, MySQL.

Delta relation and rule evaluation result tables are used by delta programs, and the interpreter assumes that these tables are used solely for the purpose of program evaluation, meaning that it takes responsibility for creating and dropping them when necessary.

Delta programs are Datalog programs where IDB relations, i.e. relations that appear in the head of the program's rules, are replaced with delta relations. The transformation of a program into a delta program entails swapping the IDB relation table name with the equivalent delta relation table name created prior.

For example, a simple program to insert all tuples of one relation into another, and assuming the extensional database containing the relation E has been updated with new facts.

$$T(x, y) \leftarrow E(x, y)$$

Gets transformed into

$$\Delta E(x, y) \leftarrow E(x, y)$$
$$\Delta T(x, y) \leftarrow \Delta E(x, y)$$

Note that in this case the body relation E has also been *deltafied*, due to it containing new data that needs to be processed.

Program splitting needs to occur in order to separate recursive rules from nonrecursive ones. This is important as nonrecursive rules need to be run only once and sequentially. Splitting means creating two programs: recursive and nonrecursive, and filling them with rules from the original. Recursive rules are rules which contain the IDB relation (head) inside the body.

After splitting, the nonrecursive program needs to be sorted according to the dependencies between the rules. A sorted program is a Datalog program, where rules are executed in a sequential order without violating a relational dependency. A relation is dependent on another when a rule r_2 has an EDB relation, which is present in the head of another rule r_1,then means that we must first derive the facts from rule $r_1$before evaluating r_2.

For example, in the program

$$r_1: r_1(x, y) :- r_0(x, y)$$
$$r_2: r_2(x, y) :- r_1(x, y)$$

From code nr x, we can clearly see that r_2 depends on r_1 to be evaluated, which means that r_1 must be evaluated first. There is also r_0, which is an independent relation.

Sorting a Datalog program requires generating a rule dependency graph and stratifying it. A dependency graph is a directional graph, where rules are represented by nodes, and edges imply a dependency of the source rule on the destination rule, where the body of the source rule contains a relation name, or the symbol, which is present in another rule's head.

A **strongly connected component** is a maximal subgraph in which any vertex is reachable from any other vertex within the same component. Essentially, for every pair of vertices u and v in a strongly connected component, there exists a directed path from u to v and a directed path from v to u.

This dependency graph needs to be stratified into strongly connected components using an appropriate algorithm, such as Kosaraju's algorithm. The strongly connected components in this context are sets of dependent rules that serve as the basis for the evaluation order. Finally, the sets are merged into a program and the order reversed as

required to handle the side-effect of Kosaraju's algorithm. With this, the interpreter has the two programs necessary for complete evaluation.

3.3 Program Materializer

Materializing a program has to be done in order to store intermediate results, from where the next iterations of the program can continue. What sets semi-naïve implementation apart from naïve is that the interpreter materializes only the change in data since the last iteration, which significantly reduces the amount of processed data. Figure 4 presents the general algorithm for evaluating the entire program with materialization.

In a SQL database, materialization occurs with the help of auxiliary tables that store intermediate results. After a rule in the program is evaluated, the results are stored in a temporary table, e.g. $\Delta\Delta T$. The antijoin (implemented through SQL's $EXCEPT$) of this temporary table and the corresponding delta table ΔT is inserted into T, to be able to track whether fixpoint has been reached, and ΔT, thereby starting a new increment for the next evaluation. In relational algebra terms, the operation sequence for materialization is:

$$T \leftarrow \Delta\Delta T \ \triangleright \Delta T$$
$$\Delta T \leftarrow \Delta\Delta T \ \triangleright \Delta T$$

Materialization in either recursive or nonrecursive variants is very similar, but with the key difference being that the nonrecursive program rules need to be evaluated sequentially, while recursive program rules are independent and can be evaluated in any order. As a result, evaluation and materialization of nonrecursive programs happens one rule at a time, while with recursive programs the interpreter evaluates all rules first, and then materializes the results in bulk.

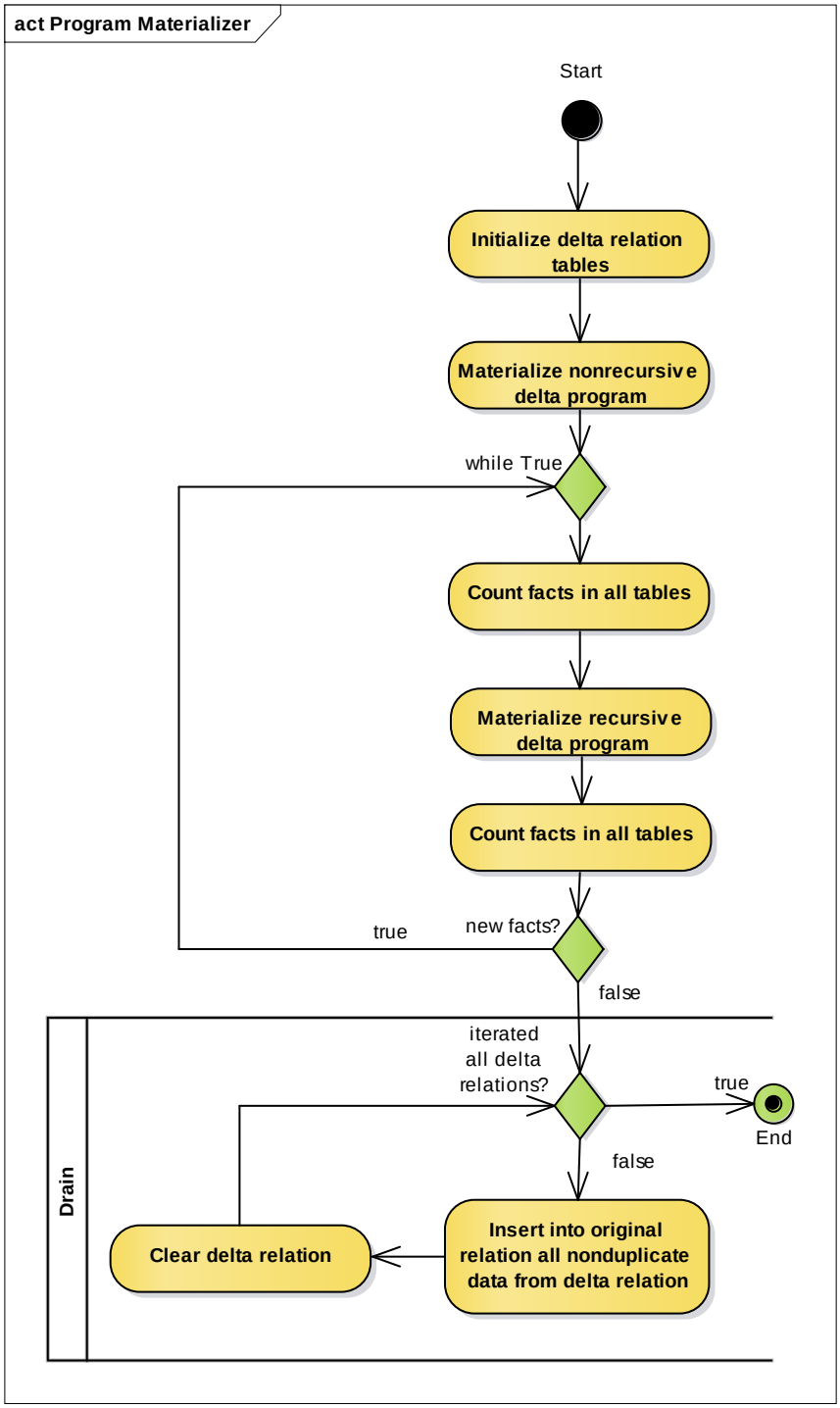

Figure 4. Program Materializer Activity Diagram.

Prior to program evaluation, the delta relation tables have to be initialized with the data contained by the respective original relation tables. The sorted nonrecursive program must be evaluated and materialized first, and then the recursive program is evaluated until no new facts can be derived. After evaluation, the remaining data in all delta relations needs to be drained into the respective original relations.

3.4 Rule Evaluator

Rule Evaluator is responsible for evaluating a Datalog rule within a SQL database. It executes a sequence of instructions derived from a given rule, using the Select-Project-Join stack, to produce the desired output in the database.

SQL queries are generated using the SQLGlot library. However, the next iteration of the compiler will allow the user to choose a specific SQL dialect and database connection.

This (Instruction Stack) library serves as a bridge between Datalog and SQL, allowing for the transformation of Datalog rules into SQL queries using relational algebra principles. It provides a structured representation of relational algebra operations as Instruction subclasses that are collected into a list (the `Stack`), which is later used to evaluate them in SQL by Rule Evaluator.

The instruction stack module is split into several classes. The ProjectionInput class model is shown by Figure 5.

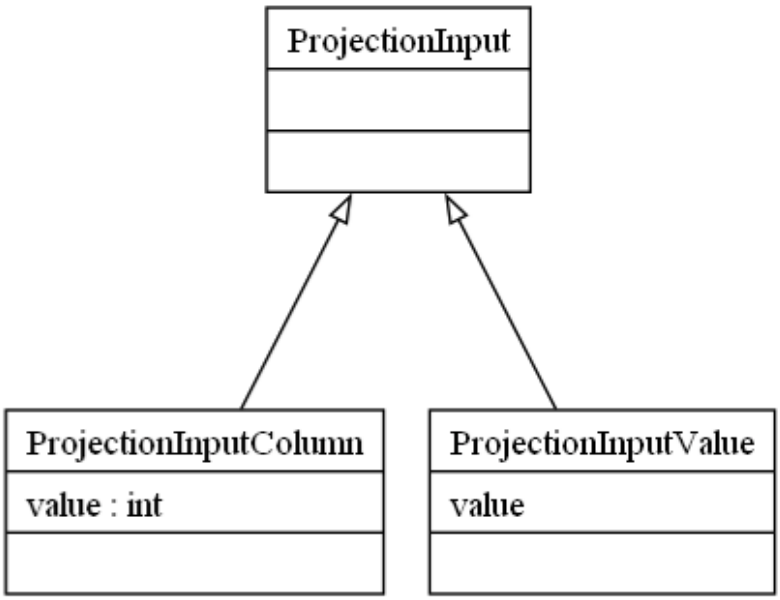

Figure 5. ProjectInput Class Model

The Instruction class model as depicted by Figure 6 shows different subclasses of instructions available to the Stack.

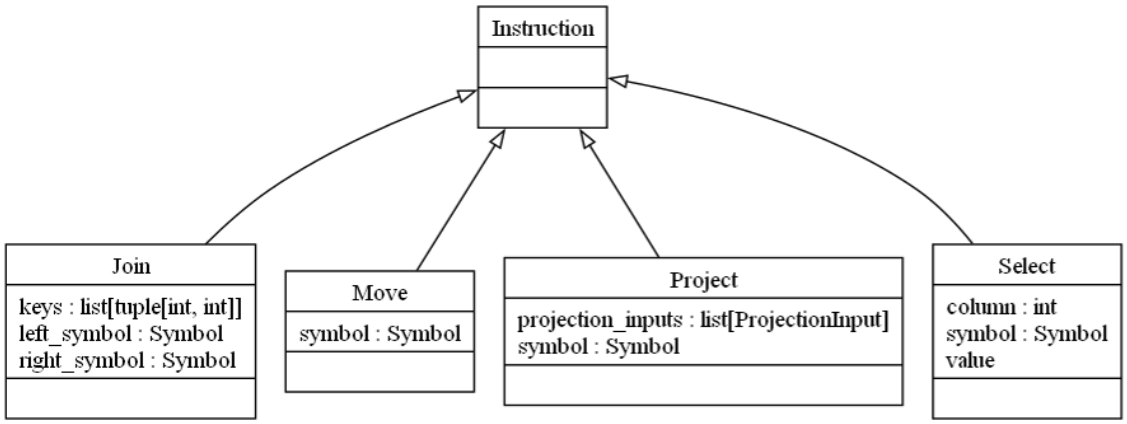

Figure 6. Instruction Class Model

Finally, the Rule Evaluator class has a very simple design shown by Figure 7, where the primary functionality is in the step() function. This function is responsible for building and executing SQL statements according to the instruction stack of the evaluated rule.

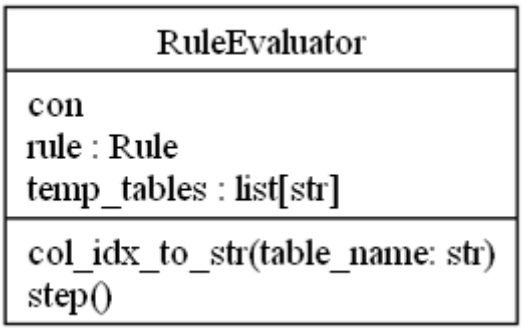

Figure 7. Rule Evaluator Class Model

For the recursive part, it follows the bottom-up evaluation strategy as described in point 2.1.

4 Experimental Results

This section elaborates on how Terry was benchmarked. First, we introduce the programs and the datasets, and then we look at the performance observed during testing.

The interpreter was benchmarked on a Google Cloud Compute Engine (16 vCPU, 128 GB, Intel Broadwell platform) running a Debian 12 virtual machine. Performance was profiled using three graph datasets: a dense graph, a sparse graph and the Lehigh University Benchmark [22]. Every benchmark was allowed 100 test runs, where each data point was recorded at the successful execution of a SQL statement. The recorded variables were iteration number, statement type, evaluated rule (if applicable) and elapsed time of the SQL statement.

4.1 Benchmark Programs and Datasets

A **dense graph** is a graph where the number of edges is close to the maximum number of edges it can possibly have. For a simple graph with n vertices, the maximum number of edges in a directed graph $n \cdot (n-1)$, which occurs when every vertex is connected to every other vertex. Therefore, a graph is considered dense if the number of edges $|E|$ approaches this maximum value.

The dense graph dataset contains less data in the form of vertices and edges, but it is very interconnected, giving more data per iteration for the Datalog program. The expectation in this case is that the interpreter might struggle with an increased amount of data to process, but it has to do less hops overall.

Conversely, a **sparse graph** has relatively few edges compared to the number of vertices $|V|$, meaning most pairs of vertices are not directly connected. In other words, if

$$|E| \ll n(n-1)$$

Then the graph is considered sparse.

The sparse graph dataset contains less data per iteration than the dense dataset, but has more iterations in total, which means that the interpreter will spend more time materializing fewer facts per hop.

The following program was run on the dense and sparse graph datasets

$$T(x, y) \leftarrow E(x, y)$$
$$T(x, z) \leftarrow T(x, y), E(y, z)$$

Figure 8. Datalog transitive closure program

An **RDF** (Resource Description Framework) graph is a structured way of representing information about resources in the form of a graph. RDF is a foundational technology of the semantic web, which aims to make data on the internet machine-readable. RDF represents information using a set of triples, each consisting of a subject, a predicate, and an object. This structure is analogous to the basic sentence structure in natural language, where the subject and object represent the entities or concepts, and the predicate represents the relationship or property linking them. In RDF, a collection of such triples is typically visualized as a graph, where each triple forms an edge from a subject node to an object node, labeled by the predicate.

The following program was run on the RDF dataset:

$$T(s, p, o) \leftarrow RDF(s, p, o)$$
$$T(y, 0, x) \leftarrow T(a, 3, x), T(y, a, z)$$
$$T(y, 0, x) \leftarrow T(a, 4, x), T(y, a, z)$$
$$T(x, 2, z) \leftarrow T(x, 2, y), T(y, 2, z)$$
$$T(x, 1, z) \leftarrow T(x, 1, y), T(y, 1, z)$$
$$T(z, 0, y) \leftarrow T(x, 1, y), T(z, 0, x)$$
$$T(x, b, y) \leftarrow T(a, 2, b), T(x, a, y)$$

Figure 9. Datalog RDF program

In total there are three perspectives from which the benchmarked data was analysed.

1. The general performance of the interpreter among the tested database engines was measured by observing the time it took to evaluate a given program until reaching the fixed point.

2. Rule performance for each database engine. The goal was to capture the average execution time per rule for three specific computational rules evaluated across the four different database types. Each bar represents the average execution time for a given rule-database combination, measured in milliseconds.

3. Comparing distribution of operations

The central inquiry about the benchmark revolves around what impacts performance more significantly: a smaller number of large queries (dense dataset) or a higher volume of very small queries (sparse dataset).

The evaluation of RDF data presents a unique case within this context. RDF is characterized by query scenarios that are significantly challenging, predominantly because each rule is recursive and typically requires only two iterations to complete. This scenario represents a "worst-case" in terms of query complexity and recursion.

4.2 Overall Performance

Figure 10 contains a set of density plots that visualize the evaluation speed of an interpreter across different database engines for three types of graph data: Dense, Sparse, and RDF. Each plot shows the distribution of execution times in milliseconds for each database type as it processes queries on these graph structures.

Each plot displays the kernel density estimation (KDE) which is a way to estimate the probability density function of a random variable. The x-axis represents the time in milliseconds, and the y-axis represents the density.

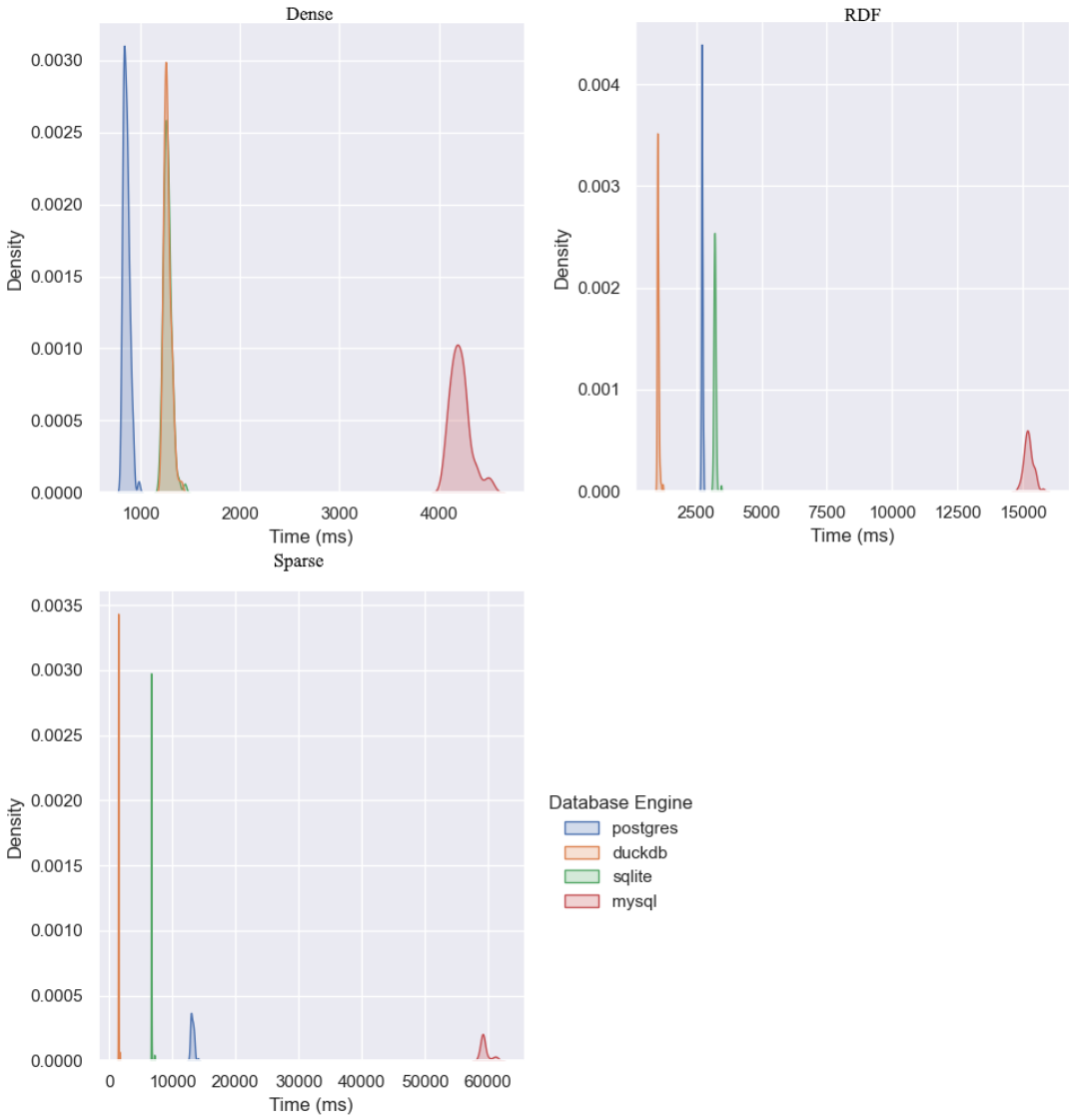

Figure 10. Dense, Sparse and RDF Graph Evaluation Speed

The variability in execution times can be seen with broader peaks for MySQL, which indicates less consistency in performance. Postgres appears to have an edge over the other database engines with dense graphs, consistently performing evaluation under 1000 ms. On the other extreme, MySQL is lagging far behind the others overall in every single dataset. DuckDB seems to handle the RDF graph the best out of four, with Postgres and SQLite being nearly even in their performance. We can also see an answer to our inquiry: it is clear that the sparse dataset turned out to be the most challenging, though, DuckDB appears to be performing well. It is also notable that both of the embedded databases outperform the others.

4.3 Rule Performance

Before analysing the performance of rule evaluation, let us clarify the rules:

0. $dE(x, y) \leftarrow E(x, y)$ initializing nonrecursive rule that is run implicitly by the interpreter.

1. $dT(x, y) \leftarrow dE(x, y)$ nonrecursive rule that inserts all edges from E into T.

2. $dT(x, z) \leftarrow T(x, y), dE(y, z)$ nonrecursive rule which finds the first hop of TC.

3. $dT(x, z) \leftarrow dT(x, y), E(y, z)$ recursive rule which finds connected nodes by connecting two different edges.

In terms of rule performance, on Figure 11 we can see a couple of trends:

- MySQL consistently shows the longest execution times across all rules, following the already established pattern.

- Postgres generally exhibits the shortest execution times, when it comes to the dense dataset, but struggles more at the last rule of the sparse dataset, which is likely due to the heavier projection that occurs, as other databases seem to suffer a performance hit as well.

- In the sparse dataset DuckDB appears to have a clear advantage over other database engines, likely due to its column-based ordering.

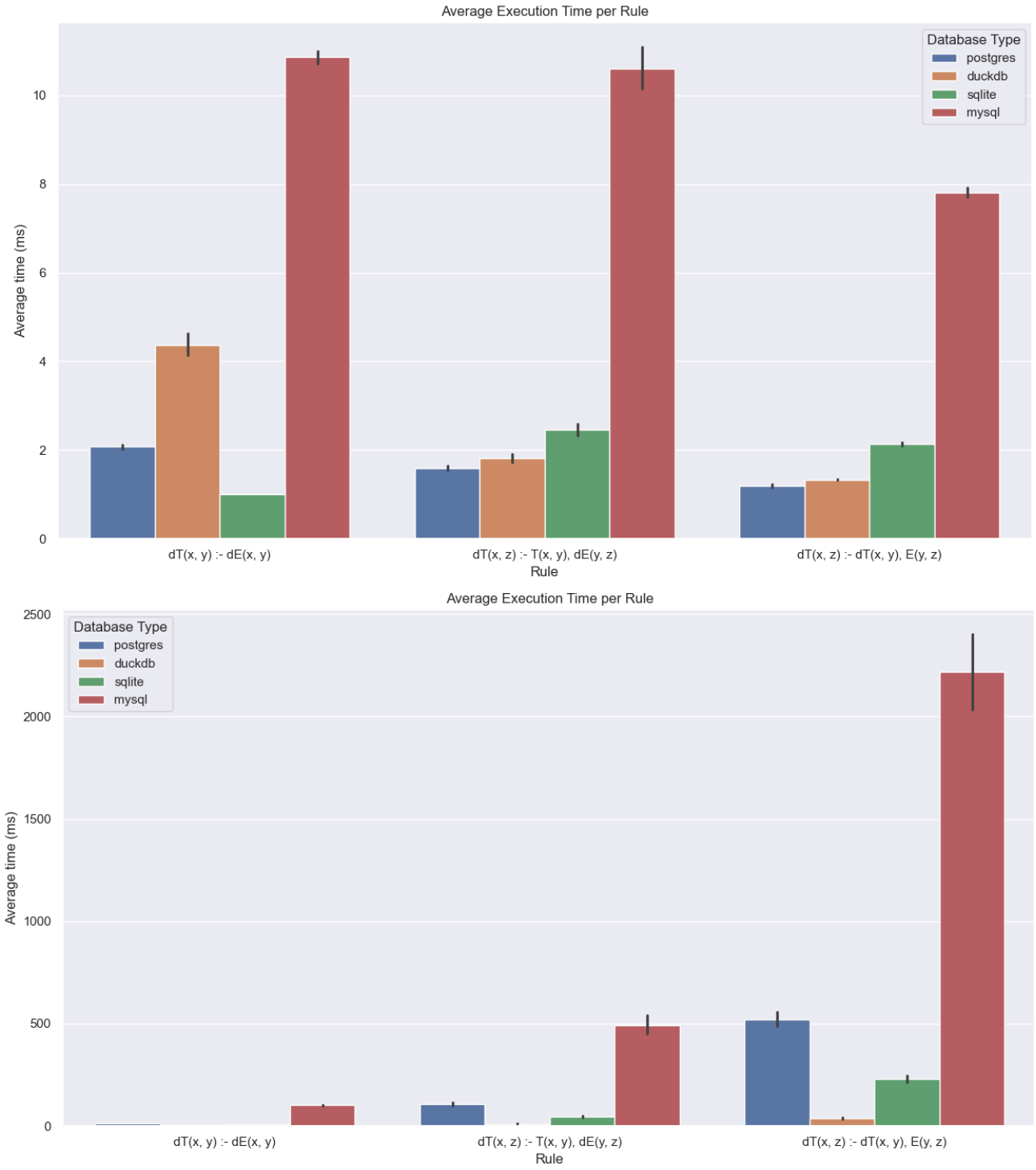

Figure 11. Dense (Top) and Sparse (Bottom) Graph Rule Performance

When it comes to the RDF dataset on Figure 12, the graph indicates that while MySQL still struggles significantly, and with the nonrecursive rule especially, all databases manage recursive computations efficiently. The notable increase in execution time for later recursive rules across all databases suggests that these rules might be due to the growing number of derived facts. This increase appears to begin at rule Recursive_5

onward, and there is a gradual increase in execution times across all databases, peaking at Recursive_10, and going down significantly afterwards.

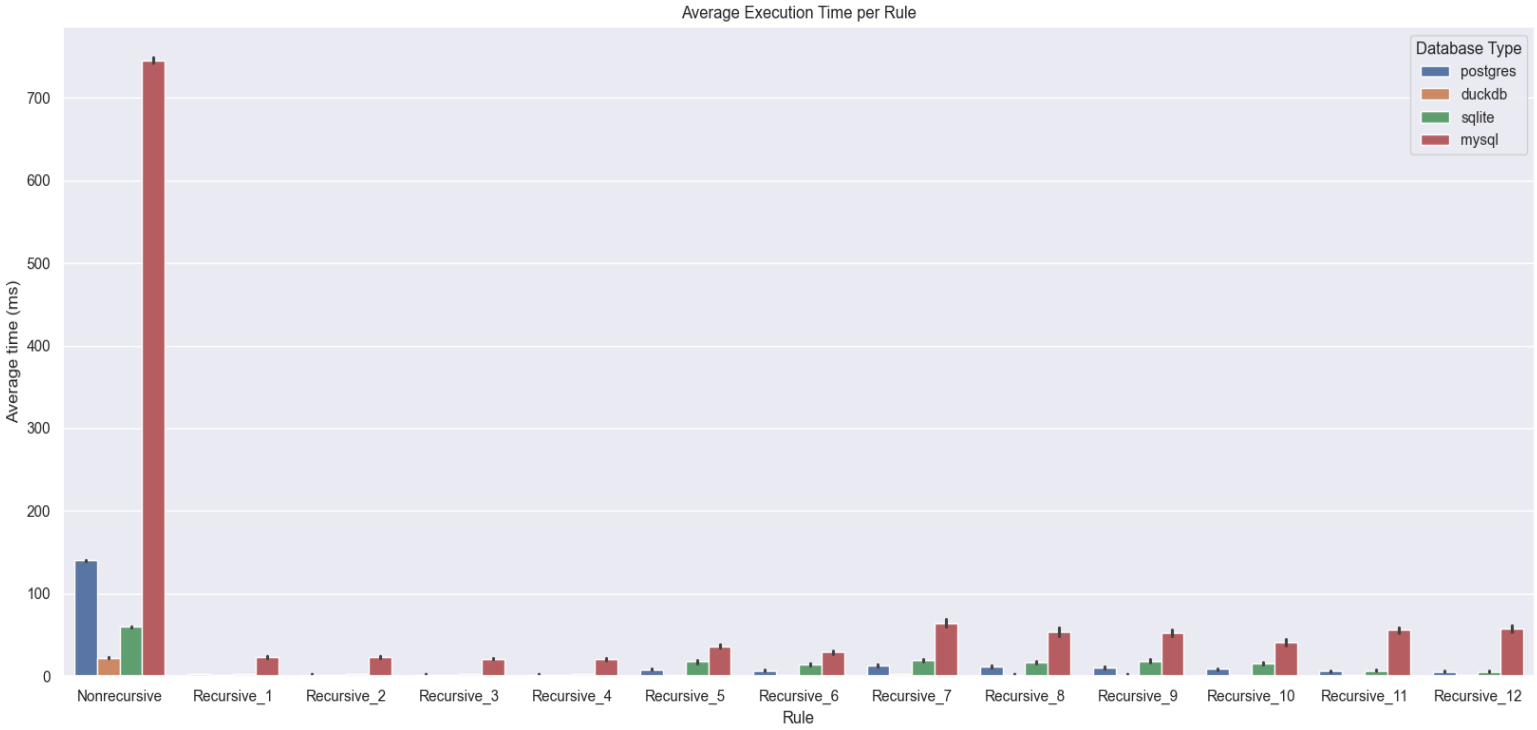

Figure 12. RDF Graph Rule Performance

4.4 Statement Type Performance

On Figure 13 we can see a plot for each dataset, which shows the average execution time per statement type. In total, there are 5 types of statements.

1. COMPILER_INIT operations initialize the tables before program evaluation.
2. SPJ_PROJECT, SPJ_SELECT, SPJ_JOIN, SPJ_CLEAR are operations related to rule evaluation.
3. MAT_NONREC, MAT_REC are operations that are responsible for materializing evaluation results.

4. DRAIN is the operation responsible for moving the last remaining facts from the delta relations into their respective original relations, and dropping the temporary delta tables.

5. FACT_COUNT is responsible for counting rows inside a table.

Some of the results are notable:

- SQLite clearly seems to struggle with DDL statements responsible for dropping tables, seeing how disproportionate the amount of time is spent in SPJ_CLEAR. In addition, SQLite also appears to struggle relatively more with table creation statements under COMPILER_INIT.

- In the case of sparse and RDF datasets, DuckDB did not appear to struggle with any particular operation, meaning that the performance effect that its columnar engine has is uniform across all operations.

- In the sparse dataset, Postgres spends more time relative to others in SPJ_PROJECT and SPJ_JOIN, which could suggest that embedded databases perform better due to their close integration and serverless design, which means almost no latency between the database and the application.

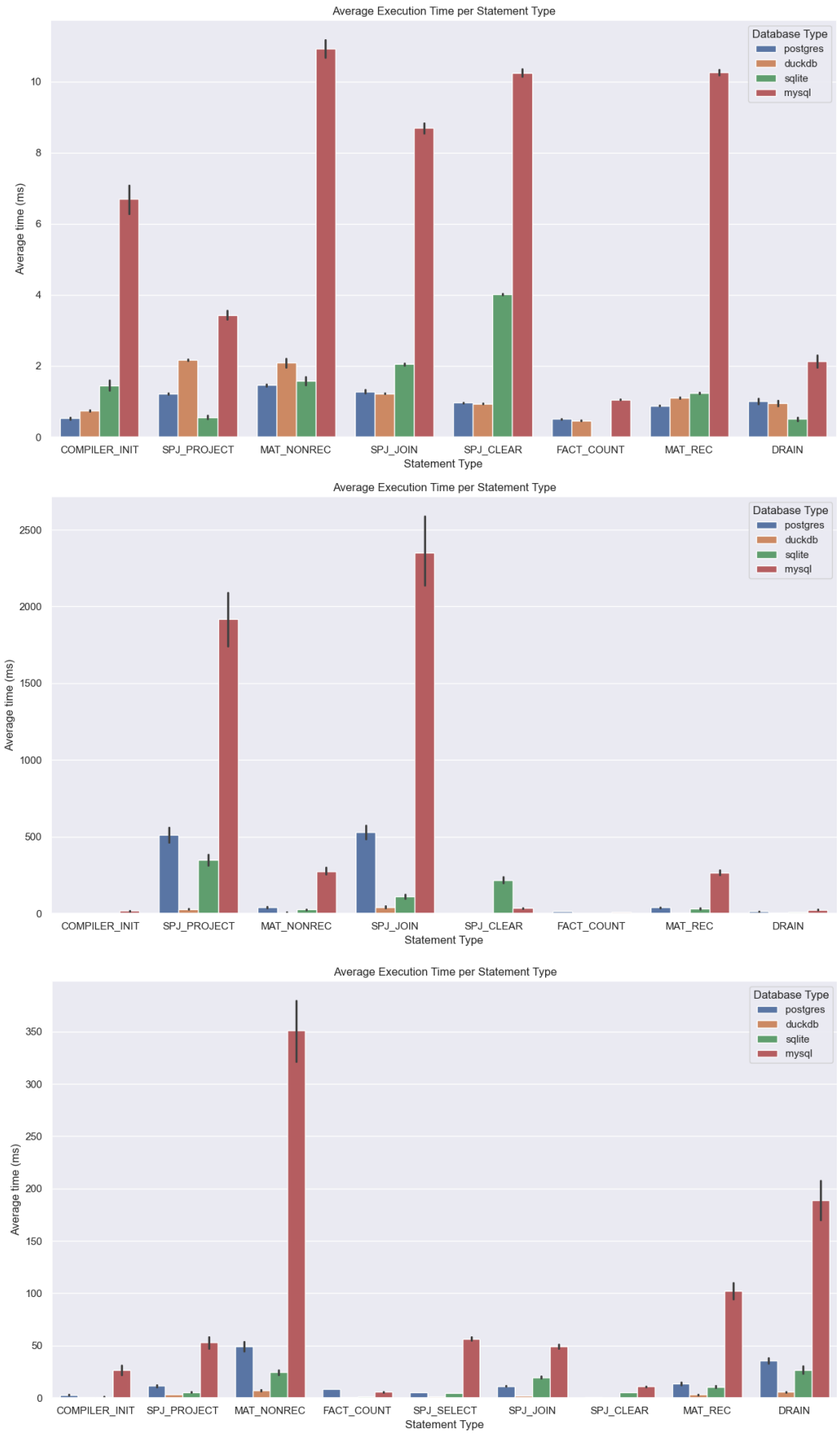

Figure 13. Dense (Top), Sparse (Middle) and RDF (Bottom) Statement Type Performance

5 Conclusion

In this thesis, we have explored the design, implementation, and performance evaluation of a Datalog to SQL interpreter, Terry. The primary motivation behind this endeavour was to enhance the ease of handling complex queries, especially recursive queries, which are naturally expressed in Datalog but often challenging to implement directly in SQL. The thesis was divided into three main components: the theoretical background, technical implementation of Terry, benchmarking and analysis of the results.

The thesis introduced the concepts necessary to understand the theoretical background behind Terry. This involved explaining the fundamentals behind relational algebra and relational databases, SQL, Datalog and how deductive databases operate, as well as the basic idea behind Datalog evaluation. In the second part, we explained how a SQL-to-Datalog interpreter works, showing the algorithms behind Datalog parsing, rule evaluation and program materialization. The third part dealt with profiling the performance and explaining the trends occurring in the performance of the different database engines.

This thesis contributes to the field by providing concrete benchmarks on how different SQL database engines handle the evaluation of Datalog queries. Additionally, this work introduces a novel approach to Datalog query parsing in the form of the SPJ stack that could benefit researchers interested in this field.

Finally, we can answer the research questions we posed in the beginning.

1. Effect of Database Engine Choice on Performance

Overall, the plots in section 4 provide a fairly concise representation of how each database handles graph-based queries, highlighting differences in performance that can inform future developments on the database selection. Namely, we can exclude MySQL from

further consideration due to its notably poor performance in all benchmarks. DuckDB was able to evaluate the sparse dataset, as well as the RDF graph, far more efficiently than its competitors. PostgreSQL did better with the dense dataset, but DuckDB may be a more versatile option when dealing with graph solving problems.

2. Energy Efficiency of Terry's implementation

After seeing the interpreter work in runtime, we can see that the efficiency heavily depends on the database engine that it is attached to. We could approximate the efficiency in terms of energy consumption compared to other languages by looking at our results and comparing them to the energy efficiency of other programming languages [23]. Assuming an average ratio between energy (J) and time (ms) of $\frac{0.040+0.046+0.014}{3} = 0.033$ for Python according to the study, we can apply this ratio to the data we have gathered.

On average, dense graph evaluation took 4208 ms on MySQL, which means that on average, a single transitive closure query will yield an energy amount of $E_{MySQL} = 138.9\ J$. Assuming a user base of $100\ 000$ clients, who will do a hundred queries per day for a year, will yield about 58.7 metric tons of CO_2 equivalent [24].

From the same study we can calculate the average ratio between the energy consumption of Rust and Python, which is a factor of $\frac{49.07:1793.46 + 238.30:12784.09 + 26.15:1061.41}{3} = 0.024$, meaning that Rust is on average 42.5 times more energy efficient than Python. Doing this calculation for the rest of the graphs and database engines gives us an efficiency comparison between them in Table 1.

Table 1. Metric tons of CO_2 equivalent emitted annually by Terry and a hypothetical Rust implementation of Terry

	Python				**Rust**			
	postgres	duckdb	sqlite	mysql	postgres	duckdb	sqlite	mysql
Dense	11.93	17.68	23.41	58.71	0.29	0.42	0.56	1.41
Sparse	183.22	19.83	92.73	829.4	4.4	0.48	2.23	19.91
RDF	37.84	14.3	44.66	212.31	0.91	0.34	1.07	5.1

To illustrate the point, using Rust would save us approximately $89\ 391$ kilograms of coal annually. We can clearly see that an in-memory Rust implementation of Terry would have

a considerable environmental effect, and that future implementations should take this into account.

6 References

[1] G. G. L. T. S. Ceri, “What you always wanted to know about Datalog (and never dared to ask),” *IEEE Transactions on Knowledge and Data Engineering,* vol. 1, no. 1, pp. 146-166, 1989.

[2] e. a. Todd J. Green, “Datalog and Recursive Query Processing,” *Foundations and Trends® in Databases,* vol. 5, no. 2, pp. 105-195, 2013.

[3] e. a. Alexander Shkapsky, “Big Data Analytics with Datalog Queries on Spark,” *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16),* p. 1135–1149, 2016.

[4] S. G. a. M. S. L. J. Seo, “SociaLite: Datalog extensions for efficient social network analysis,” *2013 IEEE 29th International Conference on Data Engineering (ICDE),* pp. 278-289, 2013.

[5] T. S. K. S. K. Pieciukiewicz, “Usable Recursive Queries,” in *Advances in Databases and Information Systems. ADBIS,* 2005.

[6] J. L. Edelmira Pasarella, “A Datalog Framework for Modeling Relationship-based Access Control Policies,” *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies (SACMAT '17 Abstracts),* p. 91–102, 2017.

[7] M. T. Z. Ling Liu, Encyclopedia of Database Systems, Springer Publishing Company, Incorporated, 2009.

[8] C. J. Date, A Guide to the SQL Standard, Standard. Addison-Wesley Longman Publishing Co., Inc., 1989.

[9] M. A. Olson, “Selected and Implementing an Embedded Database System,” *Computer,* vol. 33, no. 9, pp. 27-34, September 2000.

[10] A. Nori, “Mobile and embedded databases,” *SIGMOD '07,* pp. 1175-1177, 2007.

[11] SQLite Consortium, “About SQLite,” 10 10 2023. [Online]. Available: https://www.sqlite.org/about.html. [Accessed 4 April 2024].

[12] “Why DuckDB,” [Online]. Available: https://duckdb.org/why_duckdb. [Accessed 9 May 2024].

[13] The PostgreSQL Global Development Group, “What is PostgreSQL?,” [Online]. Available: https://www.postgresql.org/about/. [Accessed 29 April 2024].

[14] solid IT gmbh, “DB-Engines Ranking,” [Online]. Available: https://db-engines.com/en/ranking. [Accessed 1 May 2024].

[15] S. S. H. B. T. L. a. W. Z. Todd J. Green, “Datalog and Recursive Query Processing,” *Foundations and Trends® in Databases,* vol. 5, no. 2, pp. 105-195, 2013.

[16] J. Z. Z. Z. A. A. P. K. J. P. Zhiwei Fan, “Scaling-Up In-Memory Datalog Processing:,” *Proceedings of the VLDB Endowment,* vol. 12, no. 6, pp. 695-708, 2019.

[17] Python Software Foundation, “General Python FAQ,” [Online]. Available: https://docs.python.org/3/faq/general.html. [Accessed 14 April 2024].

[18] https://www.tiobe.com/tiobe-index/, “TIOBE Index,” 2024. [Online]. Available: https://www.tiobe.com/tiobe-index/. [Accessed 2 May 2024].

[19] SQLAlchemy, “Overview,” [Online]. Available: https://docs.sqlalchemy.org/en/20/intro.html. [Accessed 12 May 2024].

[20] SQLGlot developers, “SQLGlot,” [Online]. Available: https://sqlglot.com/sqlglot.html. [Accessed 10 January 2024].

[21] P. D. Terry, Compilers and Compiler Generators - an introduction with C++, International Thomson Computer Press, 1997.

[22] Lehigh University, “SWAT Projects - the Lehigh University Benchmark (LUBM),” [Online]. Available: https://swat.cse.lehigh.edu/projects/lubm/. [Accessed 7 May 2024].

[23] e. a. Rui Pereira, “Energy efficiency across programming languages: how do energy, time, and memory relate?,” *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering,* pp. 256-267, 2017.

[24] United States Environmental Protection Agency, “Greenhouse Gas Equivalencies Calculator,” [Online]. Available: https://www.epa.gov/energy/greenhouse-gas-equivalencies-calculator. [Accessed 11 May 2024].

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I, Edvard Paas

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Terry, a Datalog to SQL Interpreter”, supervised by Sadok Ben Yahia and Bruno Rucy Carneiro Alves De Lima.
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

12.05.2024

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.