




Article

Matching and Rewriting Rules in Object-Oriented Databases

Giacomo Bergami , Oliver Robert Fox  and Graham Morgan 

School of Computing, Faculty of Science, Agriculture and Engineering, Newcastle University,
Newcastle upon Tyne NE4 5TG, UK; o.fox3@newcastle.ac.uk (O.R.F.); graham.morgan@newcastle.ac.uk (G.M.)

* Correspondence: giacomo.bergami@newcastle.ac.uk

Abstract: Graph query languages such as Cypher are widely adopted to match and retrieve data in a graph representation, due to their ability to retrieve and transform information. Even though the most natural way to match and transform information is through rewriting rules, those are scarcely or partially adopted in graph query languages. Their inability to do so has a major impact on the subsequent way the information is structured, as it might then appear more natural to provide major constraints over the data representation to fix the way the information should be represented. On the other hand, recent works are starting to move towards the opposite direction, as the provision of a truly general semistructured model (GSM) allows to both represent all the available data formats (Network-Based, Relational, and Semistructured) as well as support a holistic query language expressing all major queries in such languages. In this paper, we show that the usage of GSM enables the definition of a general rewriting mechanism which can be expressed in current graph query languages only at the cost of adhering the query to the specificity of the underlying data representation. We formalise the proposed query language in terms declarative graph rewriting mechanisms described as a set of production rules $L \rightarrow R$ while both providing restriction to the characterisation of L , and extending it to support structural graph nesting operations, useful to aggregate similar information around an entry-point of interest. We further achieve our declarative requirements by determining the order in which the data should be rewritten and multiple rules should be applied while ensuring the application of such updates on the GSM database is persisted in subsequent rewriting calls. We discuss how GSM, by fully supporting index-based data representation, allows for a better physical model implementation leveraging the benefits of columnar database storage. Preliminary benchmarks show the scalability of this proposed implementation in comparison with state-of-the-art implementations.



Citation: Bergami, G.; Fox, O.R.; Morgan, G. Matching and Rewriting Rules in Object-Oriented Databases. *Mathematics* **2024**, *12*, 2677. <https://doi.org/10.3390/math12172677>

Academic Editor: Andrea Scozzari

Received: 6 August 2024

Revised: 25 August 2024

Accepted: 25 August 2024

Published: 28 August 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: direct acyclic graphs; generalised semistructured model; graph grammars; graph query languages; algorithms; operator algebras

MSC: 68W40; 68P15; 68Q42; 68Q55; 68R10; 68U35

1. Introduction

Query languages [1] fulfill the aim of retrieving and manipulating data after being adequately processed according to a physical model requiring preliminary data loading and indexing operations. Good declarative languages for manipulating information, such as SQL [2], are agnostic from the underlying physical model while expressing the information need of combining (JOIN), filtering (WHERE) and grouping (e.g., COUNT(*) over GROUP BY-s) data over its logical representation. This language can also be truly declarative as it does not require a user to explicitly convey this in terms of operations to be performed over the data rather than instructing the machine which information should be used and which types of transformations should be applied. Due to the intrinsic nature of graph data representation, graph query languages such as Gremlin [3], Cypher [4], or SPARQL [5] are procedural (i.e., navigational) due to the navigational structure of the data, thus heavily requiring the user to inform the language how to match and transform the data. These

considerations extend to how edge and vertex data should be treated, thus adding further complexity [6,7].

On the other hand, graph grammars [8] provide a high-level declarative way to match specific sub-graphs of interest from vertex-labelled graphs for then rewriting them into another subgraph of interest, thus updating the original graph of interest. These consist in a set of rules $\{L_i \rightarrow R_i\}_{1 \leq i \leq N}$, where each rule $L_i \rightarrow R_i$ consists of two graphs L_i and R_i possibly sharing a common subgraph K : while $K \subseteq L$ specifies a potential removal of either vertices or edges, $K \subseteq R$ specifies their addition. Automating the application of such rules requires explicitly following a graph navigational order for applying each matched subgraph to the structure [9] to guarantee the generation of a unique acceptable graph representation resulting from the querying data. As a by-product of exploiting topological sorts for scheduling visiting and rewriting operations, we then require to precisely identify an *entry match* vertex for each L_i from a rule, thus precisely determining from which position within the graph the rewriting rule should be applied, and in which order.

At the time of writing, the aforementioned graph query languages are not able to express graph grammars natively without resorting to explicitly instructing the query language how to restructure the underlying graph data representation (Lemma 7); furthermore, the replacement of previously matched vertices with new ones invalidates previous matches, thus forcing the user to pipeline different queries until convergence is reached. On the other hand, we would expect any graph query language to directly express the rewriting mechanism by directly expressing the set of rules, without necessarily instructing the querying language in which order such operations shall be performed. Furthermore, when a property-graph model is chosen, and morphisms are expressed through the properties associated with the vertices and edges rather than their associated IDs as per the current Cypher and Neo4j v5.20 specifications, this makes the deletion of vertices and their update through the resulting morphisms quite impractical, as such data model provides no clear way to refer to both vertices and edges via unique identifiers. As a result of these considerations, we then derive that, at the time of writing, both the graph query languages and their underlying representational model are insufficient to adequately express rewriting rules as general as the ones postulated by graph grammars over graphs containing property-value associations, independently from their representation of choice [10].

To overcome the aforementioned limitations, we propose a query language, for the first time directly expressing such rewriting desiderata: we restrict the set of all the possible matching graphs L_i into ego-nets containing one entry-point while incorporating nesting operations (Section 6.1); as updating graph vertices' properties breaks the declarative assumption as the user should specify the order in which the operations are performed, we relax the language declarativity for the sole rewritings. To better support this query language, we completely shift the data model of interest to object-oriented databases, where relationships across objects are expressed through object "containment" relationships, and where both objects and such containments are uniquely identified. By also requiring that any object shall never contain itself at any nesting level [6], we obtain cycle-free containment relationships.

The paper is structured as follows: after providing some preliminary notation used throughout the paper (Section 2), we introduce the relational and the graph data models from current literature, while focusing on both their data representation and associated query languages (Section 3). Their juxtaposition motivates the proposal of our Generalised Semistructured Model (Section 4), for which we outline both the logical and physical data model, where the latter leverages state-of-the-art columnar relational database representations [11]; we also introduce the concept of a view for a generalised semistructured model, as well as introduce some morphism notation used throughout the paper. After introducing the designated operator for instantiating the morphisms by joining matched containments stored in distinct sets of tables (Section 5), we finally propose our query language for matching and rewriting object-oriented databases expressed in the aforementioned GSM model (Section 6). We characterise the formal semantics of such novel graph query lan-

guage in pseudocode notation (Algorithm from Section 6) characterised in terms of both algorithmic and algebraic notation (Sections 6.3 and 6.4) for the matching part, as well as in terms of Structured Operational Semantics (SOS) [12] for describing the rewriting steps (Section 6.5). The remaining sections discuss some of the expressiveness properties of such query language if compared to Cypher (Section 7), discuss its time complexity (Section 8), and benchmark it against Cypher running over Neo4j v5.20, showing the extreme inefficiency of property graph computational model (Section 9), which are fairly restricted due to the impossibility of conveying a novel single query for any possible graph schema (Lemmas in Section 7). Scalability tests show that our solution outperforms Cypher and Neo4j v5.20, providing the query language standard nearer to the recently proposed GQL, by two orders of magnitude (Section 9.1) while providing a computational throughput being 600 times faster than Neo4j by solving more queries in the same comparable amount of time (Section 9.1). Last, we draw our final conclusions where we propose some future works (Section 10). To improve the paper's readability, we move some definitions (Appendix A) and the full set of proofs for the Lemmas and Corollaries (Appendix B) to the Appendix.

These main contributions are then obtained in terms of the following ancillary results:

- Definition of a novel nested relational algebra natural equi-join operator for composing nested morphisms, also supporting left outer joins by parameter tuning (Section 5).
- Definition of an object-oriented database view for updating the databases on the fly without the need for heavy restructuring of the loaded and indexed physical model (Section 4.3).
- As the latter view relies on the definition of a logical model extending GSM (Section 4.1), we show that the physical model is isomorphic to an indexed set of GSM databases expressed within the logical model (Lemma 1).

2. Preliminary Notation

We denote **sets** $S = \{x_1, \dots, x_n\}$ of cardinality $|S| = n$ as standard. The power set $\wp(S)$ of any set S is the set of all subsets of S , including the empty set \emptyset and S itself. Formally, $\wp(S) = \{S' | S' \subseteq S\}$.

We define a *finite function* $f: A \rightarrow B$ via its *graph* $[(x_1, f(x_1)), \dots, (x_n, f(x_n))]$ explicating the association between a value from the domain of f ($\{x_i, \dots, x_i\} = \text{dom}(f)$) and a non-NULL codomain value. Using an abuse of notation, we denote the *restriction* $f|_X$ as the evaluation of f over a domain $X \cap \text{dom}(f)$, i.e., $f|_X = [(x, f(x)) | x \in X \cap \text{dom}(f)]$. We can also denote $f(x) := C$ where C is the definition of the function over x as $x \mapsto C$. With an abuse of notation, we denote $|f|$ as the cardinality of its domain, i.e., $|f| = |\text{dom}(f)|$. We say that two functions f and f' are *equivalent*, $f \doteq f'$, if and only if they both share the same domain and, for each element of their domain, both functions return the same codomain value, i.e., $f \doteq f' \Leftrightarrow \text{dom}(f) = \text{dom}(f') \wedge \forall x \in \text{dom}(f). f(x) = f'(x)$.

A **tuple** or indexed set $t = \langle t_1, \dots, t_n \rangle$ of *length* $|t| = n$ is defined as a finite function in $\mathbb{N} \rightarrow \mathcal{V}$ where t_i has i as a natural-valued independent variable representing the index of the tuple, and the dependent variable corresponds to the element $t(i)$ in the tuple.

A binary relation \mathfrak{R} on a set A is said to be an **equivalence relation** if and only if it is reflexive ($\forall x \in A. x \mathfrak{R} x$), symmetric ($\forall x, y. x \mathfrak{R} y \Leftrightarrow y \mathfrak{R} x$), and transitive ($\forall x, y, z. x \mathfrak{R} y \wedge y \mathfrak{R} z \Rightarrow x \mathfrak{R} z$). Given a set A and an equivalence relationship \mathfrak{R} , an **equivalence class** $[x]_{\mathfrak{R}} \subseteq A$ for $x \in A$ is the set of all the elements in A that are equivalent to x , i.e., $[x]_{\mathfrak{R}} = \{y \in A | x \mathfrak{R} y\}$. Given a set A and an equivalence relationship \mathfrak{R} , the **quotient set** A/\mathfrak{R} is the set of all the equivalence classes of A , i.e., $A/\mathfrak{R} = \{[x]_{\mathfrak{R}} | x \in A\}$. We denote \doteq_X as the equivalence relationship denoting two functions as equivalent if they are equivalent after the same restriction over X , i.e., $f \doteq_X f' \Leftrightarrow f|_X \doteq f'|_X$.

Given a set of all the possible string attributes Σ^* and a set of all the possible values \mathcal{V} , a **record** is defined as a finite function $f: A \rightarrow \mathcal{V}$ with $A \subseteq \Sigma^*$ [13]. Given two records represented as finite functions f and f' , we denote $f \oplus f'$ as the *overriding* of f by f' returning $f'(x)$ for each $x \in \text{dom}(f')$ and $f(x)$ otherwise. Given this, we can also denote

the graph of a function as $\oplus_{x_i \in \text{dom}(f)} [(x_i, f(x_i))] \equiv [(x_i, f(x_i)) | x_i \in \text{dom}(f)]$. We denote as NULL a void value not representing a sensible value in \mathcal{V} : using the usual notation in the database area at the cost of committing an abuse of notation, we say that $f(x)$ returns NULL if and only if f is not defined over x (i.e., $x \notin \text{dom}(f)$). We say that a record ϵ is empty if no attribute is associated with a value in \mathcal{V} (i.e., $\text{dom}(\epsilon) = \emptyset$ and $\forall x \in \mathcal{V}. \epsilon(x) = \text{NULL}$).

Higher-Order Functions

Higher-Order Functions (HOFs) are functions that either take one or more functions as arguments or return a function as a result. We provide the definition of some HOFs used in this paper:

- The **zipping** operator maps n tuples (or records) t^1, \dots, t^n to a record of tuples (or records) r defined as $r(i) = \langle t_i^1, \dots, t_i^n \rangle$ if and only if all n tuples are defined over i :

$$\zeta(t^1, \dots, t^n) = [\langle t_i^1, \dots, t_i^n \rangle | i \leq \min(|t^1|, \dots, |t^n|), \forall 1 \leq j \leq n. i \in \text{dom}(t^j)]$$

- Given a function $f: A \rightarrow B$ and a generic collection C , the **mapping** operator returns a new collection by applying f to each component of C :

$$\mu(f, C) = \begin{cases} \{f(x) | x \in C\} & C \text{ is a set} \\ [f(C(i)) | i \in \text{dom}(C)] & C \text{ is a record} \\ \langle f(C_1), \dots, f(C_n) \rangle & C \text{ is a tuple} \end{cases}$$

- Given a binary predicate p and a collection C , the **filter** function trims C by restricting it to its values satisfying p :

$$F(p, C) = \begin{cases} \{x \in C | p(x)\} & C \text{ is a set} \\ [(x, C(x)) | x \in \text{dom}(C), p(C(x))] & C \text{ is a record} \\ x \mapsto \arg \min_{\substack{|C| \geq i \geq x \\ \text{s.t. } p(C_i)}} C_i & C \text{ is a tuple} \end{cases}$$

- Given a binary function $f: A \times \mathcal{V} \rightarrow A$, an initial value $\alpha \in A$ (accumulator), and a tuple C , the **(left) fold** operator is a tail-recursive function returning either α for an empty tuple, or $f(\dots f(\alpha, t_1), t_n)$ for a tuple $t = \langle t_1, \dots, t_n \rangle$:

$$\Lambda(f, \alpha, C) = \begin{cases} \alpha & |C| = 0 \\ \Lambda(f, f(\alpha, C(m)), C_{|\text{dom}(C)| \setminus \{m\}}) & |C| \neq 0 \wedge m := \min \text{dom}(C) \end{cases}$$

- Given a collection of strings C and a separator s , **collapse** (also referred to as *join* in programming languages such as Javascript or Python) returns a single string where all the strings in C are separated by s . Given “^” the usual string concatenation operator, this can be expressed in terms of Λ as follows:

$$\lambda(C, s) := \Lambda(^, s, C)$$

When s is a space “_”, then we can use $\lambda(C)$ as a shorthand.

- Given a function $f: A \rightarrow B$ and two values $a \in A$ and $b \in B$, the update of f so that it will return b for a and will return any other previous value for $f(x)$ otherwise is defined as follows:

$$\text{PUT}_f(a, b) := f \oplus [(a, b)] \tag{1}$$

- Given a (finite) function f and an input value y , the HOF **optionally obtaining** the value of $f(y)$ if $y \in \text{dom}(f)$ and returning z otherwise is defined as:

$$\text{OPTGET}_f(y, z) := \begin{cases} f(y) & y \in \text{dom}(f) \\ z & \text{oth.} \end{cases}$$

Please observe that we can use this function in combination with PUT to set multiple nested functions:

$$\text{PUT}_f^2(\langle i, j \rangle, v) := \text{PUT}_f(i, \text{PUT}_{\text{OPTGET}_f(i, \emptyset)}(j, v)) \quad (2)$$

3. Related Works

This section outlines different logical models and query languages for both graph data (Sections 3.1.1 and 3.1.2) and the nested relational model (Sections 3.2.1 and 3.2.2), while starting from their mathematical foundations. We then discuss an efficient physical model used for the relational model (Section 3.2.3).

3.1. Graph Data

3.1.1. Logical Model

Direct Acyclic Graphs (DAGs) and Topological Sort

A **digraph** $\gamma = (V, E)$ consists of vertices V and edges E being a subset of V^2 . We say that such a graph is *weighted* if its definition is extended as (V, E, ω) where $\omega: E \rightarrow \mathbb{R}$ is a weight function associating each edge in the graph to a real value. A *closed trail* is a sequence of distinct edges $(s_1, d_1), \dots, (s_n, d_n) \in E$ which edge (s_i, d_i) is such that $s_i = d_{i-1}$ and $d_i = s_{i+1}$ if any, and where $s_1 = d_n$. We say that a digraph is *acyclic* if it contains no closed trails consisting of at least one edge, and we refer to it as a Direct Acyclic Graph (DAG).

If we assume that edges in a graph reflect dependency relationships across their vertices, a natural way to determine the visiting order of the vertices is first to sort its vertices topologically [14], thus inducing an operational scheduling order [11]. This requires the underlying graph to be a DAG. Notwithstanding this, any DAG might come with multiple possible valid topological sorts; among these, the scheduling of tasks operations usually prefers visiting the operations bottom-up in a layered way, thus ensuring the operations are always applied starting from the sub-graphs having less structural-refactoring requirements than the higher ones [11].

We say that a *topological sort* of a DAG (V, E) is a linear ordering of its vertices in a tuple t with $|t| = |V|$ so that for any edge $(u, v) \in E$ we have i and $j > 0$ such that $u = t_i$ and $v = t_{i+j}$. Given this linear ordering of the vertices, we can always define a layering algorithm [15] where vertices sharing no mutual interdependencies are placed in the same layer, and where all the vertices appearing in the shallowest layer will be connected by transitive closure to all the vertices in the deeper layers, while the vertices in the deepest layer will share no interconnection with the other vertices in the graph. To do so, we can use Algorithm 1: we use timestamps for determining the layer ID: we associate all vertices with no outgoing edge to the deepest layer 0 (Line 5) and, for all the remaining vertices, we set the vertex to the layer immediately below to the one of the deepest outgoing vertex (Line 9).

Algorithm 1 Layering DAGs from a vertex topological order in t

```

1: function LAYERFROMTOPOLOGICALSORT( $G = (V, E), t$ )
2:    $\text{time} := \langle -1, \dots, -1 \rangle$   $\triangleright |\text{time}| = |t|$ 
3:    $\text{firstVisit} := \mathbf{true}$ 
4:   for all  $p \in \text{Reverse}(t)$  do
5:      $\text{time}[p] := 0$ 
6:     if  $\text{firstVisit}$  then
7:        $\text{firstVisit} := \mathbf{false}$ 
8:     else if  $\text{IN}(p) \neq \emptyset$  then
9:        $\text{time}[p] := \max\{\text{time}[u] \mid u \in \text{OUT}(p)\} + 1$ 
10:    end if
11:  end for
12:  return  $\text{time}$ 
13: end function

```

When the graph is cyclic, we can use heuristics to approximate the vertex order, such as using the progressive ID associated with the vertices to disambiguate and choose an ordering when required.

Property Graphs

Property graphs [16] represent **multigraphs** (i.e., digraphs allowing for multiple edges among two distinct vertices) expressing both vertices and edges as multi-labelled records. The usefulness of this data model is remarked by its implementation in almost all recent Graph DBMSs, such as **Neo4j** [4].

Definition 1 (Property Graph). *A **property graph** [9] is a tuple $(V, E, L, A, U, \ell, \kappa, \lambda)$, where V and E are sets of distinct integer identifiers ($V \subseteq \mathbb{N}, E \subseteq \mathbb{N}, V \cap E = \emptyset$). L is a set of labels, A is a set of attributes and U is a set of values. $\ell: V \cup E \rightarrow \wp(L)$ maps each vertex or edge to a set of labels; $\kappa: V \cup E \rightarrow A \rightarrow U$ maps each vertex or edge within the graph and each attribute within A , to a value in U ; last, $\lambda: E \rightarrow V \times V$ maps each edge $e \in E$ to a pair of vertices $\lambda(e) = (s, t) \in V \times V$, where s is the source vertex and t is the target.*

Property graphs do not support aggregated values, as values in U cannot contain either vertices or edges, nor U is made to contain a collection of values.

RDF

The RESOURCE DESCRIPTION FRAMEWORK (RDF) [17] distinguishes resources via UNIQUE RESOURCE IDENTIFIERS (URI), be them vertices or edges within a graph; those are linked to their properties or other resources via triples, acting as edges. RDF is commonly used in the semantic web and in the ontology field [18,19]. Thus, modern reasoners such as **Jena** [20] or **Pellet** [21] assume such data structure as the default graph data model.

Definition 2 (RDF (Graph Data) Model). *An **RDF (Graph data) model** [9] is defined as a set of triples (s, p, o) , where s is called “subject”, p is the “predicate” and o is the “object”. Such triple describes an edge with label p linking the source vertex s to the destination vertex o . Such predicate can also be a source vertex [10]. Each vertex is either identified by a unique URI identifier or by a blank vertex b_i . Each predicate is only described by a URI identifier.*

Despite this model using unique resource identifiers for either vertices or edges differently from property graphs, RDF is forced to express attribute-value associations for vertices as additional edges through reification. Thus, property graphs can be entirely expressed as RDF triplestore systems as follows:

Definition 3 (Property Graph over Triplestore). *Given a property graph $G = (V, E, A, U, \ell, \kappa, \lambda')$, each vertex $v_i \in V$ induces a set of triples (v_i, α, β) for each $\alpha \in A$ such that $\kappa(v_i, \alpha) = \beta$ having $\beta \neq \text{NULL}$. Each edge $e_j \in E$ induces a set of triples (s, e_j, d) such that $\lambda'(e_j) = (s, d)$ and another set of triples (e_j, α', β') for each $\alpha' \in A$ such that $\kappa(e_j, \alpha') = \beta'$ having $\beta' \neq \text{NULL}$.*

The inverse transformation is not always possible as RDF properties as property graphs do not allow the representation of edges departing from other edges. RDF also supports the unique identification of distinct databases being loaded within the same physical storage through *named graphs* via a resource identifier. Even though it allows named graphs to appear as triplet subjects, such named graphs can appear as neither objects nor properties, thus not overcoming property graphs' limitations on representing nested data.

Notwithstanding the model's innate ability to represent both vertices and edges via URIs, the inability of the data model to directly associate each URI with a data record while requiring it to express the property-value associations via triplets, requires the expression of property updates via the deletion and subsequent creation of new triplets of the data, which might be quite impractical. Given all the above, we still focus our attention on the Property

Graph model, as it better matches our data representation desiderata by associating to vertices labels, property-value associations, as well as binary relationships without any data duplications.

3.1.2. Query Languages

Despite the recent adoption of a novel graph query language standard, GQL, (<https://www.iso.org/obp/ui/en/#iso:std:76120:en> accessed on 19 April 2024), its definition has still to be implemented yet in existing systems. This motivates us to briefly survey currently available languages. Different from the more common characterisation of graph query languages in terms of their potential of expressing traversal queries [22], a better analysis of such languages involves their ability to generate new data. Our previous work [9] proposed the following characterisation:

Graph Traversal and Pattern Matching: these are mainly navigational languages performing the graph visit through “tractable” algorithms through polynomial time visits with respect to the graph size [18,23,24]. Consequently, such solutions do not necessarily involve running a subgraph isomorphism problem, except when expressly requested by specific semantics [22,25].

(Simple) Graph Grammars: as discussed in the forthcoming paragraph, they can add and remove new vertices and edges that do not necessarily depend on previously matched data, but they are unable to express full data transformation operations.

Graph Algebras: these are mainly designed either to change the structure of property graphs through unary operators or to combine them through n-ary (often binary) ones. These are not to be confused with the path-algebras for expressing graph traversal and pattern-matching constructs, as they allow us to completely transform graphs alongside the data associated with them as well as deal with graph data collections [26–29].

“Proper” Graph Query Languages: We say that a graph query language is “proper” when its expressive power includes all the aforementioned query languages, and possibly expresses the graph algebraic operators while being able to express, to some extent, graph grammar rewriting rules independently from their ability to express them in a fully-declarative way. This is achieved to some extent in commonly available languages, such as SPARQL and Cypher [4].

Graph Grammars

Graph grammars [30] are the theoretical foundations of current graph query languages, as they express the capability of *matching* specific patterns L [31] within the data through reachability queries while applying modifications to the underlying graph database structure (*graph rewriting*) R , thus producing a single graph grammar production rule $L \xrightarrow{f} R$, where there is an implicit morphism between some of the vertices (and edges) matched in L and the ones appearing in R : the vertices (and edges) only appearing in R are considered as newly inserted vertices, while the vertices (and edges) only appearing in L are considered as removed edges; we preserve the remaining matched vertices. Each rule is then considered as a function f , taking a graph database as an input and returning a transformed graph.

The process of matching L is usually expressed in terms of subgraph isomorphism: given two graphs G and L , we determine whether G contains a subgraph G_i that is isomorphic to L , i.e., there is a bijective correspondence $L \xleftrightarrow{\mu_i^L} G_0$ between the vertices and edges of L and G_i . In graph query languages, we consider G as our graph database and return $f(G_i)$ for each matched subgraph G_i . When no rewriting is considered, each possible match G_0 for L is usually represented in a tabular form [31,32], where the column header provides the vertex and edge identifiers (e.g., variables) j from L , each row reflects each matched graph G_i , and each cell corresponding to the column j represents $\mu_i(j)$. Figure 1 shows the process of querying a graph g (Figure 1a) through a pattern L (Figure 1b), for which all the

subgraphs matching in the former could be reported as morphisms listed as rows within a table, which column headers reflect the vertex and edge variables occurring in the graph pattern (Figure 1c).

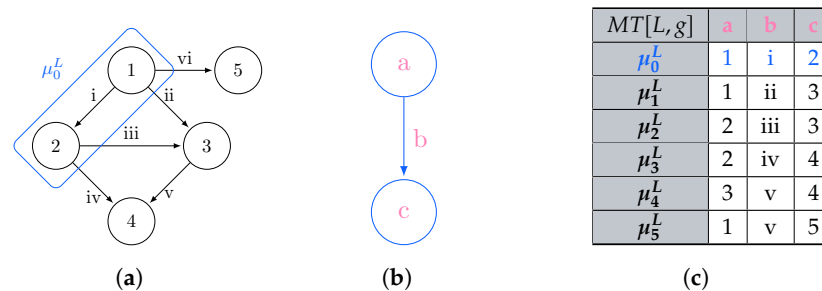


Figure 1. Listing all the subgraphs of g being a solution of the subgraph isomorphism problem of g over L . (a) Graph g to be mathed; (b) Graph pattern L ; (c) Morphism table $MT[L, g]$ where each row describes a morphism μ_i between the graph matching L and the graph g .

Figure 2 illustrates graph grammar rules as defined in GraphLog [33] for both matching and transforming any graph: we can first create the new vertices required in R , while updating or removing x as determined by the vertex or edge $f(\mu_i^{-1}(x))$ occurring in R . Deletions can be performed as the last operations from R . GraphLog still allows running of one single grammar rule at a time, while authors assume to have a graph where vertices are associated with data values and edges are labelled. Then, the rewriting operations derived from R will be applied to every subgraph being matched via a previously identified morphism in $MT[L, g]$ for each graph g of interest. Still, GraphLog considered neither the possibility of simultaneously applying multiple rewriting rules to the same graph nor a formal definition of which order the rules should be applied. The latter is required to ensure that any update on the graph can be performed incrementally while ensuring that any update to a vertex u via th information stored in its neighbours will always rely on the assumption that each neighbour will not be updated in any other subsequent step, thus guaranteeing that the information in u will never become stale.

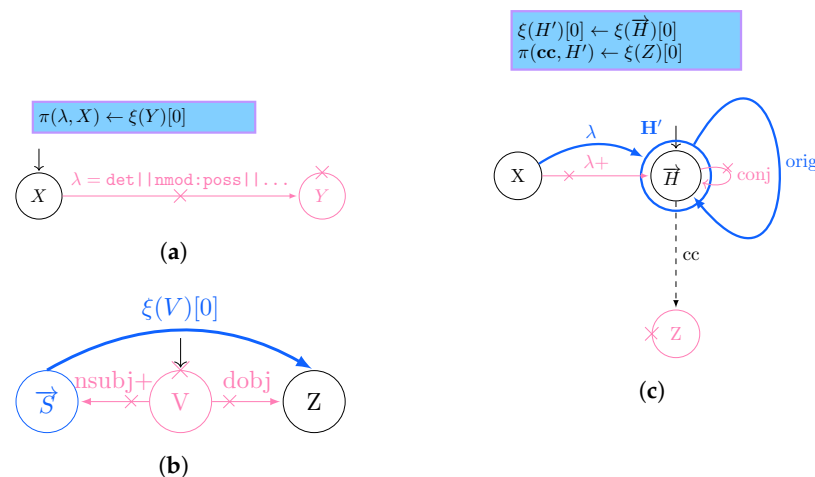


Figure 2. Graph grammar production rules à la GraphLog [33] in this paper’s use case scenario: thick denotes insertions, crosses deletions, and optional matches are dashed. We extended it with multiple optional edge label matches (|), key-value association $\pi(\lambda, X)$ for property λ and vertex X , and multiple vertex values $\xi(X)$. (a) Injecting the articles/possessive pronouns (λ) in Y for an entity X as its own properties, while deleting λ and Y ; (b) Expressing the verb as a binary relationship between subject and direct object; (c) Generating a new entity H' coalescing the ones \vec{H} under the same conjunction Z , while referring to its original constituents via orig.

This paper solves the ordering issue by applying the changes on the vertices according to their inverse topological order, thus updating the vertices sharing the least dependencies with their direct descendants first.

Furthermore, as GraphLog authors are considering a no-property graph model, authors do not consider the possibility of updating multiple property value associations over a vertex, while not providing a formal definition of how aggregation functions over vertex data should be defined. While the first problem can be solved only by properly exploiting a suitable data model, the latter is solved by the combined provision of nested morphism, explicitly nesting the vertices associated with a grouped variable, while exploiting a scripting language for expressing how level data manipulations when strictly required. Given these considerations, we will also discuss graph data models (Section 3.1) and their respective query languages (Section 3.1.2).

Proper Graph Query Languages

Given the above, we will mainly focus our attention on the last type of language. Even though these languages can be closed under either property graphs or RDF, graphs must not be considered as their main output result, since specific keywords like RETURN for Cypher and CONSTRUCT for SPARQL must be used to force the query result to return graphs. Given also the fact that such languages have not been formalised from the graph returning point of view, such languages prove to be quite slow in producing new graph outputs [6,7].

GQL is largely inspired by **Cypher**, for which this standard might be considered its natural extension. Such query language enables the partial definition of graph grammar operations by supporting the following operators restructuring vertices and edges within the graph: SET, for setting new values within vertices and edges, MERGE, for merging a set of attributes within a single vertex or edges, REMOVE for removing labels and properties from vertices and edges and CREATE for the creating of new vertices, edges and paths.

Notwithstanding the former, such language cannot express all the possible data transformation operations, thus requiring an extension via its APOC library (<https://Neo4j.com/developer/Neo4j-apoc/>, accessed on 13 November 2023) for defining User-Defined Functions (UDF) in a controlled way, like concatenating the properties associated with vertices' multiple matches:

```
MATCH (a)-[b:cc]->(c)
WITH Collect(a.name) as names, c
CREATE (x {name: apoc.text.join(names, ' ')})
RETURN x
```

Thus, Cypher can collect values for then generating one single vertex (See also Listing A2) but, due to the structural limitations of the query language (not supporting nested morphisms) and data model (not supporting explicit identifiers for both vertices and edges), it does not support the nesting of entire sub-patterns of the original matching.

A further limitation of Cypher is the inability to create a relationship with a variable as the name, as standard Cypher syntax only allows a string parameter. Using the APOC library we can use `apoc.create.relationship` to pass a variable name from an existing vertex for example. Given our last query creating the vertex `x`, we can continue the aforementioned query:

```
MATCH (a)-[:dobj]->(b)
CREATE (y {name: b.name})
WITH a, x, y
CALL apoc.create.relationship(x, a.name, {}, y) YIELD rel
RETURN x, y, rel
```

As there are no associated formal semantics for the entirety of this language's operators except its fragment related to graph traversal and matching [16], for our proofs regarding this language (e.g., Lemma 7) we are forced to reduce our arguments to common-sense

reasoning an experience-driven observation from the usage of Cypher over Neo4j similarly to the considerations in the former paragraph. In fact, such algebra does not involve the creation of new graphs: this is also reflected by its query evaluation plan, which preferred evaluation is a morphism table rather than expressing the rewriting in terms of updated and resulting property graph. As a result, the process of creating or deleting vertices and edges is not optimised.

Overall, Cypher suffers from the limitations posed by the property graph data model which, by having no direct way to refer to the matched vertices or edges by reference, forces the querying user to always refer to the properties associated with them; as a consequence, the resulting morphism tables are carrying out redundant information that cannot reap the efficient data model posed by columnar databases, where entire records can be referenced by their ID. This is evident for **DELETE** statements, voiding objects represented within the morphisms. This limitation of the property graph model, jointly with the need for representing acyclic graphs, motivates us to use the Generalised Semistructured Model (GSM) as an underlying data model for representing graphs, thus allowing us to refer to the vertices and edges by their ID [34]. Consequently, our implementation represents morphisms for acyclic property graphs as per Figure 1c.

Figure 3a provides a possible property graph instantiation of the digraph originally presented in Figure 1a. Notwithstanding the former definition, Neo4j’s implementation of the Property Graph model substantially differs from the aforementioned mathematical definition, as it does not allow the definition of an explicit resource identifier for both vertices and edges. After expressing the matching query in Figure 1b in Cypher as **MATCH (a)-[b]->(c)RETURN ***, the resulting morphism table from Figure 3b does not explicitly reference the IDs referring to the specific vertices and edges, thus making it quite impractical to update the values associated with the vertices while continuing to restructure the graph, as this will require to re-match the previously updated data to retain it in the match. Although this issue might be partially solved by exploiting explicit incremental views over the property graph model [32], this solution had no application in Neo4j v5.20, thus making it impossible to fully test its feasibility within the available system. Furthermore, the elimination of an object previously matched within a morphism will update the table by providing an empty object rather than providing a NULL match. This will motivate us to investigate other ID-based graph data models.

Neo4j lists some additional existing limitations beyond APOC and the expressibility of graph grammars in a declarative way with Cypher within their documentation (<https://Neo4j.com/docs/operations-manual/current/authentication-authorization/limitations/>, accessed on 13 November 2023), mainly related to the interplay between data access security requirements and query computations.

At the time of writing, the most studied graph query language both in terms of semantics and expressive power is **SPARQL**, which allows a specific class of queries that can be sensibly optimised [5,31]. The algebraic language used to formally represent SPARQL performs queries’ incremental evaluations [35], and hence allows for boosting the querying process while data undergoes updates (both incremental and decremental).

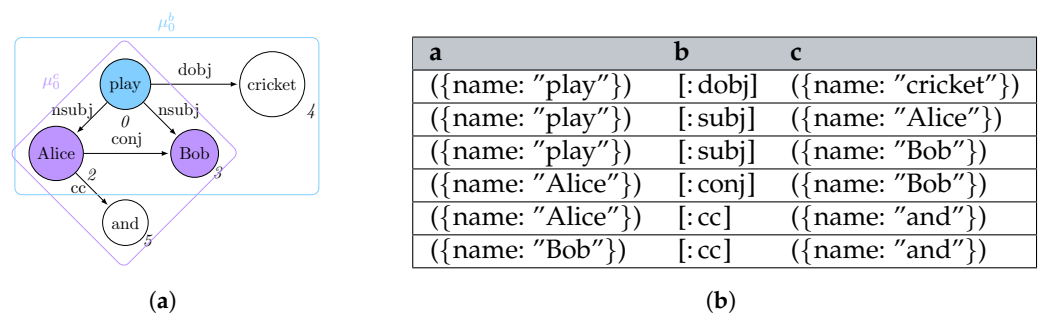


Figure 3. Framing Figure 1 in the context of Neo4j’s implementation of the Property Graph model. (a) Dependency graph for “Alice and Bob play cricket”; (b) Neo4j’s property graph morphism table.

While the clauses represented within the WHERE statement are mapped to an optimisable intermediate algebra [31,36], thus including the execution of “optional joins” paths [37] for the optional matching of paths, such considerations do not apply for the constraints related to the graph update or return, such as CONSTRUCT, INSERT, and DELETE. While CONSTRUCT is required for returning a graph view as a final outcome, INSERT and DELETE create and remove RDF triplets by chaining them with matching operations. These operations also come with sensible limitations: while the first does not allow the return of updated graphs that can be subsequently queried by the same matching algorithm, the two latter statements merely update the underlying data structure and require the re-computation of the overall matching query to retain the updated results. We will partially address these limitations in our query language and data model by associating ID properties directly via an object-oriented representation while keeping track of the updated information on an intermediate view, which is always accessible within the query evaluation phase.

Last but not least, the usage of so-called *named graphs* allows for the selection of over two distinct RDF graphs, which substantially differs from the queries expressible on Cypher, where those can be only computed by one graph database at a time. Notwithstanding the former, the latest graph query language standard is very different from SPARQL, hence, for the rest of the paper, we are going to draw our attention to Cypher.

3.2. Nested Relational Model

3.2.1. Logical Model

A nested relational model describes data represented in tabular format, where each table, composed of multiple records, comes with a schema.

Given a set of attributes Σ^* and a set of *datatypes* \mathcal{T} , a **schema** $S \in \mathcal{T}$ is a finite function mapping each string attribute in Σ^* to its associated data type ($S: \Sigma^* \rightarrow \mathcal{T}$). A schema is said to be not *nested* if it maps attributes to all basic data types, and *nested* otherwise. This syncretises the traditional function-based notation for schemas within the traditional relational model [13] with the tree-based characterisation of nested schemas

In this paper, we restrict the *basic datatypes* in $\mathcal{B} \subseteq \mathcal{T}$ to the following ones: vertex-ID ni , containment-ID ci , and a label or string str . Each of these types is associated with a set of possible values through a $\bar{\mathcal{B}}$ function: vertex- and containment-ID are associated with natural numbers ($\bar{\mathcal{B}}(ni) = \bar{\mathcal{B}}(ci) = \mathbb{N}$), while the string type is associated with the set of all the possible strings ($\bar{\mathcal{B}}(str) = \Sigma^*$).

A **record** Γ , associated with a schema $\mathcal{S}(\Gamma) = S$, is also a finite function mapping each attribute in $\text{dom}(S)$ to a possible value, either a natural number, a string, or a list of records (**tables**) as specified by the associated schema S ($\forall x \in \text{dom}(\Gamma). \Gamma(x) \in \bar{\mathcal{B}}(S(x))$). We define a **table** T with schema $\mathcal{S}(T) = S$ as a list of records all having schema S , i.e., $\forall \Gamma \in T. \mathcal{S}(T) = \mathcal{S}(\Gamma)$.

3.2.2. Query Languages

Relational algebra [13] is the de facto standard to decompose relational queries expressed in SQL to its most fundamental operational constituents while providing well-founded semantics for SQL. This algebra was later extended [38,39] to consider nested relationships. Relational algebra was also adapted to represent the combination of single edge/triple traversals in SPARQL, so as to express the traversal semantics of both required and optional patterns [5].

We now detail a set of operators of interest that will be used across the paper.

We relax the union operation from standard relational algebra by exploiting the notion of **outer union** [40]: given two tables t and s , respectively, with schema S and U , their outer union is a table $t \sqcup s$ with schema $S \oplus U$ and containing each record from each table where shared attributes are associated with the same types ($\forall x \in \text{dom}(S) \cap \text{dom}(U). S(x) = U(x)$):

$$t \sqcup s = \{\Gamma \mid \Gamma \in t \vee \Gamma \in s\} \tag{3}$$

A **restriction** [13] or **projection** [41] operation $\pi_L(t)$ over a relational table t with schema S returns a new table $\pi_L(t)$ with schema in $S|_L$ where both its schema and its records have a domain restricted to the attributes in L :

$$\pi_L(t) := \langle \Gamma|_L \mid \Gamma \in t \rangle$$

A **renaming** [41] operation $R_{L \rightarrow R}(t)$ over a relational table t with schema S and $L \subseteq \text{dom}(S)$ replaces all the occurrences of attributes in L with ones in R , thus returning a new table with schema $S|_{\text{dom}(S) \setminus L} \oplus [(r_i, S(l_i)) \mid \langle l_i, r_i \rangle \in \zeta(L, R)]$:

$$R_{L \rightarrow R}(t) = \left\langle \Gamma|_{\text{dom}(\Gamma) \setminus L} \oplus [(r_i, \Gamma(l_i)) \mid \langle l_i, r_i \rangle \in \zeta(L, R), l_i \in \text{dom}(\Gamma)] \mid \Gamma \in t \right\rangle \quad (4)$$

A **nesting** [38] operation $\nu_{B \rightarrow A}$ over a table t with schema S returns a new table $\nu_{B \rightarrow A}(t)$ with schema $S|_{\text{dom}(S) \setminus B} \oplus [(A, S|_B)]$, where all the attributes referring to B are nested within the attribute A , which is associated with the type $S|_B$ resulting from the nested attributes. Operatively, it coalesces all the tuples in t sharing the same values not in B as a single equivalence class c : we then restrict one representative of this class to the attributes not in B for then extending it by associating to a novel attribute A the projection of c over the attributes in B , i.e., $\pi_B(c)$:

$$\nu_{B \rightarrow A}(t) := \langle (\min c)|_{\text{dom}(S) \setminus B} \oplus [(A, \pi_B(c))] \mid c \in t / \cdot|_{\text{dom}(S) \setminus B} \doteq \cdot|_{\text{dom}(S) \setminus B} \rangle \quad (5)$$

A **(natural) join** [42] between two non-nested tables t and s with schemas S and U , respectively, if $\text{dom}(S) \cap \text{dom}(U)$ then it combines records from both tables that share the same attributes, and otherwise extends each record from the left table with a record coming from the right one (cross product):

$$t \bowtie s = \langle \Gamma_i \oplus \Gamma_j \mid \Gamma_i \in t, \Gamma_j \in s, (\Gamma_i \oplus \Gamma_j)|_S = \Gamma_i, (\Gamma_i \oplus \Gamma_j)|_U = \Gamma_j \rangle$$

Given a sequence of attributes $\vec{L} = \langle L_1 \dots L_n \rangle$, this operation can be extended to join the relationship coming from the right at any desired depth level by specifying a suitable path to traverse the nested schema from the left relationship [38]:

$$t \vec{\bowtie} s = \begin{cases} t \bowtie s & \vec{L} = \emptyset \\ \langle \tilde{t}|_{S \setminus \{L_1\}} \oplus (\tilde{t}(L_1) \langle L_2, \dots, L_n \rangle \bowtie s) \mid \tilde{t} \in t \rangle & \vec{L} = L_1, L_2, \dots, L_n \end{cases} \quad (6)$$

This paper will automate the determination of \vec{L} given the schemas of the two tables (Section 5). Although it can be shown that the nested relational model can be easily represented in terms of the traditional not-nested relational model [43], this paper uses the nested relational model for compactly representing graph nesting operations over morphisms. Last, the **left (outer) join** $t \bowtie\bowtie s$ extends the results from $t \bowtie s$ by also adding all the records from t having no matching tuples in s :

$$t \bowtie\bowtie s = (t \bowtie s) \cup (t - \pi_S(t \bowtie s)) \quad (7)$$

As per SPARQL semantics, the left outer join represents patterns that might optionally appear.

3.2.3. Columnar Physical Model

Columnar physical models offer a fast and efficient way to store and retrieve data where each table \mathfrak{R} with a schema having a domain $\{\text{id}, A_1, \dots, A_n\}$ is decomposed into distinct binary relations \mathfrak{R}_{A_i} with a schema with domain $\{\text{id}, A_i\}$ for each attribute A_i in $\text{dom}(\mathfrak{R})$, thus requiring to only refer to one record by its ID while representing data boolean conditions through algebraic operations. As this decomposition guarantees that

the full-outer natural join $\bowtie_{1 \leq i \leq n} \mathfrak{R}_{A_i}$ of the decomposed tables is equivalent to the initial relation \mathfrak{R} , we can avoid listing NULL values in each \mathfrak{R}_{A_i} , thus limiting our space allocation to the values effectively present in our original table \mathfrak{R} . Another reason for adopting a logical model compatible with this columnar physical model is to keep provenance information [44] while querying and manipulating the data while carrying out data transformations. By exploiting the features of the physical representation, we are no longer required to use the logical model for representing both data and provenance information as per previous attempts for RDF graph data [45], as the columnar physical model allows natively supporting *id* information for both objects (i.e., vertices) and containments (i.e., edges), thus further extending the RDF model by extensively providing unique ID similarly to what was originally postulated by the EPGM data model for allowing efficient distributed computation [46]. These considerations remark on the generality of our proposed model relying upon this representation.

We now discuss a specific instantiation of this columnar relational model for representing temporal logs: KnoBAB [11]. Although this representation might sound distant from the aim of supporting an Object-Oriented database, Section 4.2 will outline how this might be achieved. KnoBAB stores each temporal log into three distinct types of tables: (i) a CountingTable storing the number of occurrences n of a specific activity label $a \in \Sigma$ in a trace $\sigma^i \in \mathcal{L}$ as a record $\langle a, i, n \rangle$. Such a table, created in almost linear time while scanning the log, comes at no significant cost at data loading. (ii) An ActivityTable preserving the traces' temporal information through records $\langle id, a, i, j, p, x \rangle$ asserting that the j -th event σ_j^i of the i -th log trace σ^i comes with an activity label a and it is stored as the id -th record of such a table. p (and x) points to the records containing the immediately preceding (and following) event of the trace, thus allowing linear scans of the traces. This is of the uttermost importance as the records are sorted by activity label, trace ID, and event ID for enhancing query run times. (iii) The model also instantiates as many AttributeTable k as the keys $\kappa \in K$ in the data payload associated with temporal events, where each record $\langle a, v, id \rangle$ remarks that the event occurring as the id -th element within the ActivityTable $_{\mathcal{L}}$ table with activity label a associates a key κ to a non-NULL value v . This data model also come with primary and secondary indices further enhancing the access to the underlying data model; further details are provided in [11].

At the time of writing, no property graph database exploits such a model for efficiently querying data, in particular, Neo4j v5.20 stores property graphs using a simple data model representation where the whole vertices, relationships, or properties are stored in distinct tables (<https://Neo4j.com/developer/kb/understanding-data-on-disk/>, accessed on 24 August 2024). This substantially differs from the assumptions of the columnar physical model, prescribing the necessity for decomposing any data representation in multiple different tables, thus making the overall search more efficient by assuming that each data record can be associated with a unique ID. To implement a model under these assumptions, we then require both our logical (Section 4.1) and physical (Section 4.2) model to support explicit identifiers for both objects (acting as graph vertices) and containment relationships (acting as graph edges). Although Neo4j v5.20 guarantees to represent records in fixed-size to fasten up the data retrieval process, this is insufficient to ensure fast access to edges or vertices associated with a specific label. On the other hand, our solution tries to address this limitation by explicitly indexing objects and containments by label information.

Concerning triple stores for RDF data. The only database effectively using column-based storage is Virtuoso v1.1 (https://virtuoso.openlinksw.com/whitepapers/Virtuoso_a_Hybrid_RDBMS_Graph_Column_Store.html, accessed on 24 August 2024): this solution is mainly exploited for fast retrieving the data based on the subject, predicate, and object values. As the underlying physical model does not differentiate between vertices and edges due to the specificity of the logical model requiring that both relationships among vertices (URIs) and properties associated with those must be represented via triples, it is not possible to further optimise the data access by reducing the overall size of the data to be navigated and searched. On the other hand, our proposed physical model (Section 4.2)

will store this information into separate tables: the ActivityTable, registering all the objects (and therefore, vertices) being loaded, an AttributeTable^k storing key-value properties associated with the objects, and a PhiTable^k for representing the containment relationships (and therefore, edges).

Last, both Virtuoso v1.1 and Neo4j v5.20 do not exploit query caching mechanisms as the ones proposed in [47] for also enhancing the execution of multiple relational queries. This approach, on the other hand, was recently implemented in KnoBAB, thus reinforcing our previous claims for extending such previous implementation to adapt it to a novel logical model. In particular, the Algorithm presented in Section 6.3 partially implements this mechanism for caching intermediate traversal queries that might be shared across patterns to be matched, thus avoiding visiting the same data multiple times for different pattern matching.

4. Generalised Semistructured Model v2.0

We continue the discussion of this paper's methodology by discussing the logical model (Section 4.1) as a further extension of the Generalised Semistructured Model [34] to explicitly support property-value associations for vertices via π . Although we propose no property-value extension for containments, we argue that this is minor and does not substantially change the theoretical and implementation results discussed in this paper for Generalised Graph Grammar. We also introduce a novel columnar-oriented physical model (Section 4.2 on the next page) being a direct application of our previous work on temporal databases [11] already supporting collections of databases (log of traces): this will be revised for loading and indexing collection of GSM databases defining our overall physical storage. As the external data to be loaded into the physical model directly represents the logical model, we show that these two representations are isomorphic (Lemma 1) by defining explicitly the loading and serialisation operations (Algorithm from Section 4.2).

As directly updating the physical model with the changes specified in the rewriting steps of the graph grammar rules might require massive restructuring costs, we instead keep track of such changes in a separate non-indexed representation acting as an incremental view to the entire database. We then refer to this additional structure as a GSM view $\Delta(g)$ for each GSM database g loaded in the physical model (Section 4.3). We then characterise the semantics of $\Delta(g)$ by defining the updating function for g as a materialisation function considering the incremental changes recorded in $\Delta(g)$ (Section 4.3.2 on page 19).

Last, as we consider the execution of the rewriting steps for each graph as a transformation of the vertices and edges as referenced within each morphism, modulo the updates tracked in $\Delta(g)$, it becomes necessary to introduce some preliminary notation for resolving vertex and edge variables from such morphism (Section 4.4).

4.1. Logical Model

The logical model for GSM describes a single object-oriented database g as a tuple $\langle O, \ell, \xi, \epsilon, \pi, \phi, t_\phi \rangle$, where $O \subseteq \mathbb{N}$ is a collection of objects (references). Each object is associated with a tuple of possible types $\ell(o)$ and to a tuple of string-representations $\xi(o)$ providing its human-readable descriptions. As an object might be the result of an automated entity-relationship extraction process, each object is associated with a list of confidence values $\epsilon: O \rightarrow \wp(\mathbb{R})$ describing the trustworthiness of the provided information. Differently from the previous definition [34], we extend the previous model to also associate each object to an explicit property-value association for each object $o \in O$ through a finite function $\pi: O \times \Sigma^* \rightarrow \mathcal{V}$.

Differently from our previous definition of our GSM model, we express object relationships through uniquely identified vertex containments similar to edges as in Figure 1a, to explicitly reference such containments in morphism tables similarly to Figure 1c. We associate each object o with a containment attribute κ referring to multiple containment IDs via $\phi(o, \kappa) \in \wp(\mathbb{N})$. An explicit index t_ϕ maps each of these IDs to a concrete containment $\langle o_j, w \rangle$ denoting that o_j is contained in o through the containment relationship κ with a

confidence score of w . This separation between indices and values is necessary to allow the removal of containment values when computing queries efficiently. This also guarantees the correct correspondence between each value ι in the domain of t_ϕ to the label associated with the ι -th containment requiring each ι will be associated with one solve containment relationship (e.g., $\arg \min_{\kappa \in \Sigma^*} \exists j \in g. \iota \in \phi(j, \kappa)$).

We avoid objects containing themselves at any nesting level by imposing a *recursion constraint* [6] to be checked at indexing time: this allows both the definition of sound structural aggregations, which can be then conveniently used to represent multi-dimensional data-warehouses [34]. Thus, we freely assume that no object shall store the same containment reference across containment attributes [6]: $\forall o, \kappa. \forall o', \kappa'. ((o = o' \wedge \kappa \neq \kappa') \vee o \neq o') \Rightarrow \phi(o, \kappa) \cap \phi(o', \kappa') = \emptyset$. This property can also be adapted to conveniently represent Direct Acyclic Graphs, by representing each vertex $v \in V$ of a weighted property-graph $G = (V, E)$ as a GSM object, and each edge $u \rightarrow v \in E$ with weight w and label κ reflects a containment $\langle v, w \rangle \in t_\phi(\phi(i, \kappa))$. Given this isomorphism $g \simeq G_g$, we can also apply a layered and reverse topological ordering of the vertices of g and denote it as $O_{\text{rtopo}}(g)$.

The Python code provided in Appendix A.5 showcases the possibility of instantiating Python objects within the GSM model via the implementation of an adequate transformation function, mapping all native types as π key-value properties of a GSM object, while associating the others to ϕ and t_ϕ containment relationships (Line 222). Containments also enable the representation of other object-oriented data structures, such as dictionaries/maps (Line 144) and linear data structures (Line 164), thus enabling a direct representation of JSON and nested relational data (Line 53). As per our previous work [9], GSM also supports the representation of XML (Line 117) and property graph data (Line 65). This also achieves the representation aim of EPGM by natively supporting unique identifiers for both vertices and edges, to better operate on those by directly referring to their IDs without the need to necessarily carry out all of its associated payload information [46]. As such, this model leverages all the pros and cons of the previous graph data models by framing them within an object-oriented semistructured model.

4.2. Physical Model

We now describe how the former model can be represented in primary memory for fastening up the matching and rewriting mechanism.

First, we would like to support the loading of multiple GSM databases while being able to operate over them simultaneously similar to the named graphs in the RDF model. Differently from Neo4j v5.20 and similarly to RDF's named graphs, we ensure a unique and progressive ID across different databases.

We directly exploit our ActivityTable for listing all the objects appearing within each loaded GSM: as such, the i -th record $\langle id, a, g, i, p, x \rangle$ will refer to the i -th object in the g -th GSM database, while a will refer to the first label of $\ell_j(i)$, that is $\ell_j(i)[1]$.

We extend KnoBAB's ActivityTable to represent the ϕ containment relationship; the result is a PhiTable ^{κ} for each containment attribute κ : each record $\langle \ell_0, g, o_{src}, w, o_{dst}, \iota \rangle$ refers to a specific GSM database g through which object o_i associated with the first label $\ell_0 = \ell_g(o_i)[0]$ contains o_j with an uncertainty score of w and is associated with an index ι : this expresses the idea that $\iota \in \phi(o_{src}, \kappa)$ with $t_\phi(\iota) = \langle o_{dst}, w \rangle$. At the indexing phase, the table is sorted by lexicographical order over the record's constituents. We extend this table with two indices: a primary index P_1^1 mapping each first occurring ℓ value to the first and the last object within the collection, and a secondary index P_1^2 mapping such each database ID g and object ID o_i to a collection of containment records expressing $\mu(t_\phi, \phi(o_i, \kappa))$ for each o_i and κ such that $\phi(o_i, \kappa) \neq \emptyset$.

We retain the AttributeTable ^{κ} for expressing the properties associated with the GSM vertices, for which we keep the same interpretation from Section 3.2.3 on page 12: thus, each record $\langle a, v, id \rangle$ refers to $\phi(o_i, \kappa) = v$ where o_i appears as the id -th record within the ActivityTable _{\mathcal{L}} , now used to list all the objects occurring across GSM models.

Last, ℓ and ξ properties are stored in a one-column table and a secondary index mapping each database ID and object ID to a list of records referring to the strings associated with the object ID. The same approach is also used to store the confidence values ϵ associated with each GSM object.

Algorithm 2 shows the algorithms used for loading all of the GSM databases g_i to be loaded of interest to a single columnar database representation db , as well as providing the algorithm used to serialize back the stored data into a GSM model for data visualisation and materialisation purposes. Given this, we can easily prove the isomorphism between the two shared data structures, thus also providing proof of the correctness of the two transformations which, as a result, explains the formal characterisation of such an algorithm.

Algorithm 2 Loading a Logical Model into the Physical Model and serialising it back

```

1: function LOADINGANDINDEXING( $G = \{g_1, \dots, g_n\}$ )
2:   for all  $g_i = \langle O_i, \ell_i, \xi_i, \epsilon_i, \pi_i, \phi_i, t_{i,\phi} \rangle \in G$  do
3:     for all  $j \in O_i$  do
4:        $L_i(j) := \ell_i(j); X_i(j) := \xi_i(j); C_i(j) := \epsilon_i(j)$ 
5:       ActivityTable.add( $\langle \ell(j)[0], i, j, \text{NULL}, \text{NULL} \rangle$ )
6:     end for
7:   end for
8:   INDEX(ActivityTable) ▷ Also sorting the table, [11]
9:   for all  $\kappa \in \Sigma^*, j \in O_i$  s.t. ActivityTable[ $r = \langle \ell_i(j)[0], i, j, p, x \rangle$ ] do
10:    for all  $\iota \in \phi_i(j, \kappa)$  s.t.  $t_{i,\phi}(\iota) = \langle t, w \rangle$  do
11:      PhiTable $^\kappa$ .add( $\langle \ell(j)[0], i, j, w, t, \iota \rangle$ )
12:    end for
13:    if  $\pi_i(j, \kappa) \neq \text{NULL}$  then, AttributeTable $^\kappa$ .add( $\langle \ell_i(j)[0], \pi(j, \kappa), r \rangle$ )
14:    end for
15:   INDEX(ActivityTable $^\kappa$ , PhiTable $^\kappa \mid \kappa \in \Sigma^*$ ) ▷ As in [11]
16:   return  $db := \langle L, X, C, \text{ActivityTable}, [(\kappa, \text{AttributeTable}^\kappa) \mid \kappa \in \Sigma^*], [(\kappa, \text{PhiTable}^\kappa) \mid \kappa \in \Sigma^*] \rangle$ 
17: end function

18: function SERIALISATION( $db$ )
19:    $n := \max_{r \in \text{AttributeTable}} r(1)$  Determining the maximum number of GSM databases
20:   for all  $i := 0$  to  $n$  do
21:      $O_i := \{j \mid \exists l, p, x. \langle l, i, j, p, x \rangle \in \text{ActivityTable}\}$ 
22:      $\ell_i := [(i, L_i(j)) \mid j \in O_i]; \xi_i := [(i, X_i(j)) \mid j \in O_i]; \epsilon_i := [(i, C_i(j)) \mid j \in O_i]$ 
23:      $\pi_i := [((j, \kappa), v) \mid \langle l, v, r \rangle \in \text{AttributeTable}^\kappa, \exists p, x. \langle l, i, j, p, x \rangle \in \text{ActivityTable}, \kappa \in \Sigma^*]$ 
24:      $\phi_i := [((s, \kappa), \{l' \mid \exists l', w', d', \langle l', i, j, w', d', l' \rangle \in \text{PhiTable}^\kappa\}) \mid \langle l, i, s, w, d, \iota \rangle \in \text{PhiTable}^\kappa, \kappa \in \Sigma^*]$ 
25:      $t_{i,\phi} := (\iota, \{d, w\}) \mid \langle l, i, s, w, d, \iota \rangle \in \text{PhiTable}^\kappa, \kappa \in \Sigma^*$ 
26:     yield  $\langle O_i, \ell_i, \xi_i, \epsilon_i, \pi_i, \phi_i, t_{i,\phi} \rangle$ 
27:   end for
28: end function

```

Lemma 1. A collection of logical model GSM databases $\langle g_i \rangle_{i \leq n}$ is isomorphic to the ones loaded and indexed physical model.

The proof is given in Appendix B.1. We refer to Section 8 for proofs related to the time complexity for loading, indexing, and serialising operations.

4.3. GSM View $\Delta(g)$

To avoid massive restructuring costs while updating the information indexed in the physical model, we use a direct extension of the logical model to keep track of which objects were newly generated, removed, or updated during the application of the rule rewriting mechanisms. At query time, we instantiate a *view* $\Delta(g_i)$ for each g_i being loaded within the physical model (Section 6.5). We want our view to support the following operations:

(i) creation of new objects, (ii) update of the type/labelling information ℓ , (iii) updating the human-readable value characterisation ζ , (iv) update of the containment values, (v) removal of specific objects, and (vi) substitution of previously matched vertices with newly-created or other previously matched ones. While we are not explicitly supporting the removal of specific properties or values, these can be easily simulated by setting specific fields to empty strings or values. A view for g_i is defined as follows:

$$\Delta(g_i) = \langle g_i^\Delta, \Gamma_i, \Gamma_i^v, O_i^+, O_i^-, E_i^-, \rho_i \rangle \tag{8}$$

where $g_i^\Delta = \langle O_i^\Delta, \ell_i^\Delta, \zeta_i^\Delta, \epsilon_i^\Delta, \pi_i^\Delta, \phi_i^\Delta, t_{\phi_i}^\Delta \rangle$ is a GSM database holding all the objects being newly inserted alongside with the properties as well as the updated properties for the objects within the graph g (i–iv), Γ refers to the nested morphism being considered while evaluating the query, Γ^v denotes the extension of such morphism with the newly inserted objects through variable declaration, which resulting objects are then collected in O^+ (i). O^- (and E^-) tracks all the removed objects (and specific containment objects, resp.) through their ID (v). Last, ρ is a replacement map $\bigoplus_i[(o_i, o_{\alpha(i)})]$ to be used when evaluating the transformations over morphisms occurring at a higher topological sort layer, stating to refer to an object $o_{\alpha(i)}$ when o_i occurs (vi). Γ^v and O^+ are used to retain the updated information locally to each evaluated morphism, while the rest are shared across the evaluation of each distinct morphism.

We equip $\Delta(g)$ with update operations reflecting the insertion, deletion, update, and replacement operations as per rewriting semantics associated with each graph grammar production rule. Such operations are the following:

START: re-initialises the view to evaluating a new morphism Γ' by discarding any information being local to each specific morphism:

$$\text{START}_{\Delta(g)}(\Gamma') = \langle g^\Delta, \Gamma', \emptyset, \emptyset, O^-, E^-, \rho \rangle \tag{9}$$

DELCONT: We remove the i -th containment relationship from the database:

$$\text{DELCONT}_{\Delta(g)}(i) := \langle g^\Delta, \Gamma, \Gamma^v, O^+, O^-, E^- \cup \{i\}, \rho \rangle$$

NEWOBJ: Generates a new empty object associated with a variable j and with a new unique object ID $|g| + |g^\Delta| + 1$:

$$\begin{aligned} \text{NEWOBJ}_{\Delta(g)}(j) := & \text{let fresh} := |g| + |g^\Delta| + 1 \text{ in} \\ & \langle g^\Delta, \Gamma, \text{PUT}_{\Gamma^v}(j, \Gamma^v(j) \cup \{\text{fresh}\}), O^+ \cup \{\text{fresh}\}, O^-, E^-, \rho \rangle \end{aligned}$$

REPLOBJ: replaces o_i with o_j if and only if o_j was not removed ($o_j \notin O^-$):

$$\text{REPLOBJ}_{\Delta(g)}((o_i, o_j)) := \begin{cases} \Delta(g) & o_j \in O^- \\ \langle g^\Delta, \Gamma, \Gamma^v, O^+, O^-, E^-, \rho \circ [(o_i, o_j)] \rangle & \text{oth.} \end{cases}$$

DELOBJ: We remove an object o_i only if this was already in g or if this was inserted in a previous evaluation of a morphism and, within the evaluation of the current morphism, we remove the original object o_i being replaced by $\tilde{o} = \rho(o_i)$:

$$\begin{aligned} \text{DELOBJ}_{\Delta(g)}(o_i) := & \text{let } \tilde{o} := \text{OPTGET}_\rho(o_i) \text{ in} \\ & \begin{cases} \langle g^\Delta, \Gamma', \emptyset, O^+, O^- \cup \{o_i\}, E^-, \rho \rangle & \tilde{o} \in O^+ \\ \langle g^\Delta, \Gamma', \emptyset, O^+, O^- \cup \{\tilde{o}\}, E^-, \rho \rangle & \text{oth.} \end{cases} \end{aligned}$$

UPDATE: updates one of the object property functions by specifying which of those should be updated. In other words, this is the extension of PUT^2 (Equation (2)) as a higher function for updating the view alongside one of these components:

$$\text{UPDATE}_{\Delta(g_i)}^f(\langle i, j \rangle, u) := \begin{cases} \langle O_i^\Delta, \text{PUT}_{\ell_i^\Delta}^2(\langle i, j \rangle, u), \zeta_i^\Delta, \epsilon_i^\Delta, \pi_i^\Delta, \phi_i^\Delta, t_{\phi_i}^\Delta \rangle & f \equiv \ell \\ \langle O_i^\Delta, \ell_i^\Delta, \text{PUT}_{\zeta_i^\Delta}^2(\langle i, j \rangle, u), \epsilon_i^\Delta, \pi_i^\Delta, \phi_i^\Delta, t_{\phi_i}^\Delta \rangle & f \equiv \zeta \\ \langle O_i^\Delta, \ell_i^\Delta, \zeta_i^\Delta, \epsilon_i^\Delta, \text{PUT}_{\pi_i^\Delta}(\langle i, j \rangle, u), \phi_i^\Delta, t_{\phi_i}^\Delta \rangle & f \equiv \pi \\ \text{let } n := |\text{dom}(t_{\phi_i}^\Delta)| \text{ in} & f \equiv \phi \\ \text{let } \tilde{\phi} := \text{PUT}_{\phi_i^\Delta}(\langle i, j \rangle, \{n, \dots, n + |u|\}) \text{ in} \\ \text{let } \tilde{t} := t_{\phi_i}^\Delta \oplus \bigoplus_{0 \leq j < |u|} [(j + n, u(j))] \text{ in} \\ \langle O_i^\Delta, \ell_i^\Delta, \zeta_i^\Delta, \epsilon_i^\Delta, \pi_i^\Delta, \tilde{\phi}, \tilde{t} \rangle \end{cases}$$

Concerning the time complexity, we can easily see that all operations take $O(1)$ time to compute by assuming efficient hash-based sets and dictionaries. Concerning UPDATE^ϕ , this operation takes $O(1)$ time, as we are merely inserting one pair at a time.

4.3.1. Object Replacement and Resolution

As our query language will have variables to be resolved via matched morphisms and view updates (Appendix A.1), we focus on specific variable resolution operations. Replacement operations should be interpreted as a reflexive and transitive closure over the step-wise replacement operations performed while running the rewriting query (Section 6.5 on page 33).

Definition 4 (Active Replacement). *The **active replacement** function resolves any object ID x into its final replacement vertex following the chain of subsequent unique substitutions of each single vertex in ρ , or otherwise returns x :*

$$\rho_{\Delta(g)}^*(x) := \begin{cases} \rho^{n'}(x) & n' = \arg \max_{n \in \mathbb{N}} .x \in \text{dom}(\rho^n) \wedge \rho^n(x) \neq \rho^{n+1}(x) \\ x & \text{oth.} \end{cases}$$

During an evaluation of a morphism to be rewritten, such replacements and changes should be effective from the next morphism while we would like to preserve the original information while evaluating the current morphism.

Definition 5 (Local Replacement). *The **local replacement** function blocks any notion of replacement while evaluating the original data matched by the current morphism while activating the changes from the evaluation of any subsequent morphism where such newly-created vertices from the current morphism will not be considered:*

$$\downarrow \rho_{\Delta(g)}(x) := \begin{cases} \rho^*(x) & \rho^*(x) \notin O^+ \\ x & \text{oth.} \end{cases}$$

We consider objects as removed if they have no effective replacements to be enacted in any subsequent morphism evaluation: $x \in \text{dom}(\rho) \wedge x \in O^-$. Thus, we also need to resolve objects' properties (such as ℓ , ζ , π , and ϕ) by considering the changes registered in $\Delta(g_i)$. We want to define a HOF property extraction that is independent of the specific function of choice. By exploiting the notion of local replacement (Definition 5), we obtain the following definition:

Definition 6 (Property Resolution). *Given any property access function ℓ, ζ, ϕ, π , a GSM database g_i and a corresponding view $\Delta(g_i)$ we define the following **property resolution** high-order function:*

$$\downarrow \rho_{\Delta(g)}^f(o) = \begin{cases} \emptyset & \downarrow \rho_{\Delta(g)}(o) \in O_i^- \\ f_{\Delta(g_i)}(\downarrow \rho_{\Delta(g)}(o)) & \downarrow \rho_{\Delta(g)}(o) \in O_i^{\Delta} \wedge f_{\Delta(g_i)}(\downarrow \rho_{\Delta(g)}(o)) \neq \emptyset \\ f_{g_i}(\downarrow \rho_{g_i}(o)) & \text{oth.} \end{cases}$$

where we ignore any value associated with a removed vertex in O_i^- (first case), we consider any value stored in $\Delta(g_i)$ as overriding any other value originally in the loaded graph (second case), while returning the original value if the object underwent no updates (last case).

4.3.2. View Materialisation

Last, we define a materialisation function as a function updating a GSM database g_i with the updates stored in the incremental view $\Delta(g_i)$. We consider all the objects being inserted (implicitly associated with a 1.0 ϵ score) and removed, as well as extending all the properties as per the view, thus removing containment relationships originating from or arriving at GSM objects.

$$\begin{aligned} \text{MATERIALISE}'(g_i, \Delta(g_i)) = & \langle O_i \cup O_i^+ \setminus O_i^-, \\ & \ell \oplus \ell_i^{\Delta}, \\ & \xi \oplus \xi_i^{\Delta}, \\ & \epsilon \oplus \left(\bigoplus_{o \in O_i^{\Delta} \setminus O_i^-} [(o, 1.0)] \right), \\ & \pi \oplus \pi_i^{\Delta}, \\ & \bigoplus_{\langle p, k \rangle \in \text{dom}(\phi)} [(\langle p, k \rangle, F(y \mapsto y \notin E_i^-, \phi(p, k)))] \oplus \phi_i^{\Delta}, \\ & (t_{\phi} \oplus t_{\phi, i}^{\Delta}) \rangle \end{aligned}$$

As a rewriting mechanism might add edges violating the recursion constraint, we prune the containments loading to its violation by adopting the following heuristic: after approximating the topological sort by prioritising the object ID, we remove all the containments generating a cycle where the contained object has an ID with a lower value than its container ID. From this definition, we then derive the definition of the update of all the GSM databases loaded in the physical model G with their corresponding updates in Δ via the following expression:

$$\text{MATERIALISE}(G, \Delta) = \mu(\text{MATERIALISE}', \zeta(G, \Delta)) \tag{10}$$

4.4. Morphism Notation

We consider nested relationships mapping attributes to either basic data types or to nested schemas, as our query language will syntactically avoid the possibility of arbitrary nesting depths. Given this, any attribute A_i can nest at a maximum level 1 of depth. This will then motivate a similar requirement for the envisioned operator for composing matched containments (collected in relational tables) into nested morphisms (Section 5).

As our query language requires resolving variables by associating each variable A_i to the values stored in a specific morphism Γ , we need a dedicated function enabling this. We can define a value extraction function for each morphism Γ and attribute $A_i \in \text{dom}(\Gamma)$, returning directly the value associated with A_i in Γ if A_i directly appears in the schema

of Γ ($\text{dom}(\Gamma)$), and otherwise returns the list of all the possible values associated with it within a nested relationship A_j having A_i in its domain:

$$\text{IDX}_{\Gamma}(A_i) := \mathbf{let} S := \mathcal{S}(\Gamma) \mathbf{in} \begin{cases} \langle \Gamma(A_i) \rangle & S(A_i) \in \mathcal{B} \\ \langle \gamma_i(A_i) | \gamma_i \in \Gamma(A_j) \rangle & \exists! A_j. A_i \in \text{dom}(S(A_j)) \\ \emptyset & \text{oth.} \end{cases} \quad (11)$$

When resolving a variable, we need to determine whether this refers to a containment or to an object, thus selecting to remove the most appropriate type of constituent indicated within a morphism. So, we can define a function similar to the former for extracting the basic datatypes associated with a given attribute:

$$\text{TIDX}_{\Gamma}(A_i) := \mathbf{let} S := \mathcal{S}(\Gamma) \mathbf{in} \begin{cases} S(A_i) & S(A_i) \in \mathcal{B} \\ (S(A_j))(A_i) & \exists! A_j. A_i \in \text{dom}(S(A_j)) \wedge S(A_j)(A_i) \in \mathcal{B} \\ \emptyset & \text{oth.} \end{cases} \quad (12)$$

We also need a function determining the occurrence of an attribute x nested in one of the attributes of S . This will be used for both automating the discovery of the path \vec{L} for joining nested tables from our recently designed operator (Section 5) or for determining whether two variables belong to the same nested cell of the morphism while updating the GSM view. This boils down to defining a function returning A_j if A_i is an attribute of a table nested in A_j , and NULL otherwise.

$$\text{IDNEST}_S(A_i) := \arg \min_{\substack{A_j \in \text{dom}(S) \\ \text{s.t. } S(A_j) \neq \mathcal{B}}} A_i \in \text{dom}(S(A_j)) \quad (13)$$

Last, we need a function for returning all the object and containment IDs under the circumstance that these contribute to the satisfaction of a boolean expression. We then define such a function returning such IDs at any level of depth of a nested morphism:

$$\text{SE}(\Gamma) = \mathbf{let} S = \mathcal{S}(\Gamma) \mathbf{in} \{x \in \text{dom}(\Gamma) | S(x) \in \mathcal{B}\} \cup \bigcup_{x \in \text{dom}(S), S(x) \neq \mathcal{B}} \text{SE}(\Gamma(x)) \quad (14)$$

5. Nested Natural Equi-Join

Although previous literature defines nested natural join, no known algorithmic implementation is available. As our query language will return nested morphisms by gradually composing intermediate tables through natural or left joins is, therefore, important to provide an implementation for such an operator. This will be required to combine tables derived from the containment matching (Section 6.3) into nested morphisms, where it is required to join via attributes appearing within nested tables (Section 6.4). Our lemmas show the necessity of this operator by demonstrating the impossibility of expressing it via Equation (6) directly, while capturing the desired features for generating nested morphisms.

We propose for the first time Algorithm 3 for computing the nested (left outer) equi-join with a path \vec{L} of depth at most 1. The only parameter provided to the algorithm is whether we want a left outer equi-join or a natural one otherwise (`isLeft`) and, given that the determination of the nesting path will depend on the schema of both the left and right operand, we automate (Line 9) the determination of the $\vec{L} = \langle N \rangle$ path along which compute the nested join for which, we freely assume that we navigate on the nested schema of the left operand similarly to Equation (6): this assumption comes from our practical use case scenario that we are gradually composing the morphisms provided as a left operand argument with the containment relationships provided as the right operand. Furthermore, to apply the definition from Equation (6) while automating the discovery of the path to navigate to nest the relationship, we require that each attribute appearing from the right table schema might appear as nested in one single attribute from the left table or, otherwise,

we cannot automatically determine which left attribute to choose to nest the graph visit (Line 8). Otherwise, we determine a unique attribute from the left table alongside which apply the path descent (Line 9).

Algorithm 3 Nested Natural Equi-Join

```

1: function NESTEDNATURALEQUIJOINisLeft(L, R)
2:    $S_L := S(L); S_R := S(R)$ 
3:    $IR := (\text{dom}(S_L) \setminus \{x \in \text{dom}(S_L) \mid S_L(x) \notin \mathcal{B}\}) \cap \text{dom}(S_R)$ 
4:   if  $IR = \emptyset$  then return  $L \times R$  ▷ Cross Product
5:   if  $\bigcup \{\text{dom}(A_i) \mid A_i \in \text{dom}(S_L) \wedge S_L(A_i) \notin \mathcal{B}\} \cap \text{dom}(S_R) = \emptyset$  then
6:     if isLeft then return  $L \bowtie R$  else return  $L \bowtie R$ 
7:   end if

8:   assert  $\left| \bigcup_{x \in \text{dom}(S_R)} \text{IDNEST}_{S_L}(x) \right| = 1$  ▷ Equation (13)
9:    $N := \min \bigcup_{x \in \text{dom}(S_R)} \text{IDNEST}_{S_L}(x)$ 
10:   $LM := \bigoplus_{c \in L/\dot{=}_{IR}} [(c(IR), \pi_{S_L \setminus IR}(c))]$ 
11:   $RM := \bigoplus_{c \in R/\dot{=}_{IR}} [(c(IR), \pi_{S_R \setminus IR}(c))]$ 
12:  for all  $k \in \text{dom}(LM) \cup \text{dom}(RM)$  do
13:    if  $k \notin \text{dom}(RM)$  and isLeft then
14:      for  $y \in LM(k)$  yield  $k \oplus y$ 
15:    else if  $k \in \text{dom}(LM)$  then
16:      for all  $y \in LM(k), z \in RM(k)$  do
17:         $y' := \text{copyof } y$ 
18:         $y'(N) := \text{if } \textit{isLeft} \text{ then return } y'(N) \bowtie z \text{ else return } y'(N) \bowtie z$ 
19:        yield  $k \oplus y'$ 
20:      end for
21:    end if
22:  end for
23: end function

```

The algorithm also takes into account whether no nesting path $\vec{L} = \langle N \rangle$ is derivable, thus resorting to traditional relational algebra operations: if there are no shared attributes, we boil down to the execution of a cross product (Line 4) and, if none of the attributes from the right table appear within a nested attribute from the left table, then we boil down to a classical left-outer or natural equijoin depending on the *isLeft* parameter (Line 6).

Otherwise, we know that some attributes from the right table appear as nested within the same attribute N of the left table and that the two tables share the same non-nested attributes. Then, we initialize the join across the two tables by first identifying the nested attribute N from the left (Line 9). Given IR , the attributes appearing as nonnested attributes from the left table also appear in the right one, we partition the tables by $\dot{=}_{IR}$, thus identifying the records having the same values for the same attributes in IR (Lines 10–11). Then, we start producing the results for the nested join by iterating over the values k appearing in either of the two tables (Line 12): if k appears only over the left table and we want to compute a left nested join (Line 13), we reconstruct the original rows appearing from such table and return them (Line 14). Last, we only consider values k for IR appearing on both tables and, for each row y from the left table having values in k , we compute the left (or natural equi-)join of $y(N)$ with each row z from the right table and combine the results with k (Line 18).

Properties

We prove that $\vec{L} \bowtie$ cannot trivially boil down to \bowtie unless $\vec{L} = \emptyset$; otherwise, if A_i is in \vec{L} not appearing as an attribute for the to-be-joined table schemas, we will be left out with the left table and not a classic un-nested natural join. Proofs are postponed to Appendix B.2.

Lemma 2. Given $\mathcal{S}(t) = S$ and $\vec{L} = \langle A_1, A_2, \dots, A_n \rangle$, if $A_1 \notin \text{dom}(S)$, then $t^{\vec{L}}_{\bowtie S} = t$

As this is not a desired feature for an operator whose application should be automated, this justifies the need for a novel nested algebra operator for composing nested morphisms, which should shift to left joins [37] for composing optional patterns provided within the right operand (`isLeft`), while also backtracking to natural joins or cross products if no nested attribute is shared across matched containments. The following lemma discards the possibility of the aforementioned limitation to occur from our operator, by instead capturing the notion of cross-products when tables are not sharing non-nested attributes.

Lemma 3. Given tables L and R , respectively, with schema S and U with non-shared attributes ($\text{dom}(S) \cap \text{dom}(U) = \emptyset$), either $\text{NESTEDNATURALEQUIJOIN}_{\text{False}}(L, R)$ or $\text{NESTEDNATURALEQUIJOIN}_{\text{True}}(L, R)$ compute $L \times R$.

We also demonstrate that the proposed operator falls back to the natural join when no attribute nested in the left operand appears in the right one, while also capturing the notion of left join by changing the `isLeft`

Lemma 4. Given tables L and R , respectively, with schema S and U where no nested attribute appearing in the left table appears in the schema of the second, then $\text{NESTEDNATURALEQUIJOIN}_{\text{False}}(L, R) = L \bowtie R$ and $\text{NESTEDNATURALEQUIJOIN}_{\text{True}}(L, R) = L \bowtie R$.

The next lemma observes that our proposed operator not only nests the computation of the join operator within a table, but also implements an equijoin doing a value match across the table fields that are shared within the shallowest level. This is a *desideratum* to guarantee the composition of nested morphisms within the same GSM database ID, thus requiring sharing at least the same `dbid` field (Section 6.3). Still, these operations cannot be expressed through the nested join operator available from the current literature (Equation (6)).

Lemma 5. Given tables L and R , respectively, with schema S and U , that is $\mathcal{S}(L) = S$ and $\mathcal{S}(R) = U$, where the left table has a column N ($N \in \text{dom}(S)$) being nested ($S(N) \notin \mathcal{B}$) and also appearing in the right table ($N \in \text{dom}(U)$), $\text{NESTEDNATURALEQUIJOIN}_{\text{False}}(L, R)$ cannot be expressed in terms of $L^{\langle N \rangle} \bowtie R$ for $N \in \text{dom}(S) \cap \text{dom}(U)$, $N \in \text{dom}(S(N))$, and $\text{dom}(S(N)) \cap \text{dom}(U) \neq \emptyset$.

6. Generalised Graph Grammar

After a preliminary and example-driven representation of the proposed query language (Section 6.1), we characterise the semantics of the proposed query language in terms of procedural semantics being subsumed in Algorithm 4. This is defined by the following phases: after determining the order of application of the matching and rewriting rules (Line 2), we match and cache the traversal of each containment relationship to reduce the number of accesses to the physical model (Line 3), from which we then proceed to the instantiation of the morphisms, to produce the $MT[\cdot, \cdot]$ table (Line 4). This fulfills the matching phase. Finally, by visiting the objects from each GSM database in reverse topological order, we then access each morphism stored in $MT[\cdot, \cdot]$ for then applying (Line 5) the rewriting rules according to the sorting in Line 2. As this last phase produces the views for each g_i GSM database we then materialise this view and store the resulting logical model on disk (Line 6). Each of the forthcoming sections discusses each of the aforementioned phases.

Algorithm 4 Generalised Graph Grammar (gg) evaluation

```

1: function GENERALISEDGRAPHGRAMMARS(gg,  $G = \{g_1, \dots, g_n\}$ )  $\triangleright G \simeq db$  (Lemma 1)
2:   SORTRULES(gg)  $\triangleright$  Algorithm 5
3:   CACHEINTERMEDIATERESULTS(gg, db)  $\triangleright$  Algorithm 6
4:   INSTANTIATEMORPHISMS(gg, db)  $\triangleright$  Algorithm 7
5:    $\Delta :=$ GENERATEGRAPHVIEWS(gg, G)  $\triangleright$  Algorithm 8
6:   return MATERIALISE(G,  $\Delta$ )  $\triangleright$  Equation (10)
7: end function

```

6.1. Syntax and Informal Semantics

We now discuss our novel's proposed matching and rewriting language by taking inspiration from graph grammar. To achieve language declarativeness, we do not force the user to specify the order of application of the rules as in graph rewriting.

Figure 4 provides the Backus-Naur Form (BNF) [48] for the proposed query language for matching and rewriting object-oriented databases by extending the original definition of graph grammars. Each query (gg) is a list of semi-colon-separated rules (rule), where each of those describes a matching and rewriting rule, $L_i \rightarrow_{\Theta} R_i$. For each uniquely identified rule (pt), we identify a match L_i (obj cont+ joining*) and an optional (?) rewrite R_i (op* obj). Those are separated by an optional condition predicate Θ (Appendix A.2 on page 43), providing the conditions upon which the rewriting needs to be applied to the database view, and \leftrightarrow .

L_i is characterised by one single entry-point similarly to GraphQL [49] as well as other navigational approaches to visiting graph-based data [50], thus defining the main GSM object through which we relativise the graphs' structural changes or update around its neighbouring vertices, as defined by its ego-network cont of objects being either contained ($-- \text{clabel} \rightarrow \text{obj}$) or containing ($\leftarrow \text{clabel} -- \text{obj}$) the entry-point obj. While objects should be always referred to through variables, containment relationships might be optionally referred to through a variable. Edge traversal beyond the ego-net is expressed through additional edges (joining). We require that at least one of the edges should be a mandatory one. Differently from Cypher, we can match containments by providing more possible alternatives for the containment label rather than just considering one single alternative: this acts as the SPARQL's union operator across differently matched edges, each for a distinct edge label. Please observe that this boils down to a potentially polynomial sub-problem of the usual subgraph isomorphism problem, being still in NP despite the results in Section 8 proposed for the present query language.

Up to this point, all these features are shared with graph query languages. We now start discussing features extending those: we generate nested embeddings by structurally grouping entry-point matches sharing the same containing vertex: this is achieved by specifying the need to group an object (\gg) along one of its containment vertices via a containment relationship remarked with \forall . Last, we use return statements in the rewritings to specify which entry-point objects should be replaced by any other matched or newly created objects.

Example 1. Listing 1 expresses the graph grammar rules from Figure 2 in our proposed language with minor extensions: we achieve declarativeness when associating multiple string values coming from nested vertices (and therefore, associated with a single variable) to one single vertex, as strings will be normally space-joined (Line 12) with a syntax equivalent to setting such properties where no nestings are in a morphism (Line 2). A return statement at Line 22 guarantees that, while considering matching a GSM database from Figure 3a, objects 2 and 3 for Alice and Bob in X will be replaced by the newly instantiated "Alice Bob" object h when considering the subsequent creation of the edge "plays" (Line 31). This remarks the need for visiting the GSM database in topological order to minimise the rewriting costs when the updates are applied. This is also guaranteed by the matching assumption only considering objects within

the entry-point's ego-net, as we ensure to pass on the information by layers via return statements. Figure 5a shows the result of this rewrite.

String	str	∈	Σ^*
Integer	int	∈	\mathbb{N}
Variables	vars	::=	str
Labels	labels	::=	(str)* str
Query	gg	::=	(rule ;)* rule
Rule	rule	::=	(pt:str) = (entrypoint:obj) cont+ joining* (WHERE Θ)? (\hookrightarrow op* (res:obj))?
Object	obj	::=	(>>? vars)
Contain	cont	::=	-- clabel -> obj <- clabel -- obj -- clabel hook
Join-Contain	joining	::=	obj -- clabel -> obj obj <- clabel -- obj
CLabel	clabel	::=	[$\forall?$?? (var: str :)? labels]
Pred	Θ	::=	arg ₁ = arg ₂ arg ₁ ≠ arg ₂ arg ₁ ≤ arg ₂ arg ₁ ≥ arg ₂ $\Theta_1 \vee \Theta_2$ $\Theta_1 \wedge \Theta_2$ TEST <i>script</i> str ₁ unmatched str ₂ .str ₃ str ₁ matched str ₂ .str ₃ FILL Θ
Rewrite	op	::=	del str new str set expr ₁ as expr ₂
Val	expr	::=	str SCRIPT <i>script</i> label expr src expr dst expr ξ init @ expr ℓ init @ expr π expr ₁ @ expr ₂ ϕ expr ₁ , expr ₂ if Θ over str then expr ₂ else expr ₃
PredArg	arg	::=	str expr

Figure 4. Proposed language for Graph Grammars over the GSM expressed in ANTLR4-flavoured BNF notation [51]. Terminal symbols are expressed in green. *script* refers to a double-quoted string representing a program in the Script v2.0 language [34]. Similarly to Regular Expression syntax [48], ? refer to optional sub-expressions, + (or *) indicate one (or zero) or more occurrences of a given sub-expression.

When grouping entry-points, we require those to be grouped over one same containing object, to unambiguously refer the nested entry-points to one single morphism. This allows the query language to coalesce morphisms.

Listing 1. Expressing the graph grammar rules represented visually in Figure 2 in our proposed language (file: paper_simple.txt).

```

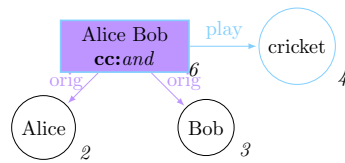
1 p1 = (X)--[l:det||nmod_poss||amod||mark||case||punct||advmod||
    ↪ advcl||dep]->(Y)
2 ↪ set (π (label l) @ X) as (ξ 0 @ Y)
3 del Y
4 (X);
5
6 p2 = (>> H)<-[∀l:]--(X)
7 --[conj] hook
8 --[? case]->(K)
9 --[? c : cc]->(Z)
10 ↪ new h
11 set (φ orig , h) as H element (>>)
12 set (π conj @ h) as (ξ 0 @ Z)
13 set (ξ 0 @ h) as (ξ 0 @ H)
14 set (φ (if ((label l) = nmod) over l then
15 (ξ 0 @ K)
16 else
17 (label l)
18 ), X) as h
19 del Z
20 del K
21 del l
22 (h);
23
24 p3 = (V)--[∀n:nsubj]->>(>>S)
25 --[? mark]->(M)
26 --[? aux]->(A)
27 --[? neg]->(N)
28 --[∀l:dobj||ccomp||nmod]->>(>>Z)
29 (Z)--[? case]->(T)
30 ↪
31 set(φ (SCRIPT "(^(^(^(^_(_[(ξ {\\"A\\"} } ) 0]) \\" \") ([ (ξ {\\"N
    ↪ \"} } ) 0])) \\" \") ([ (ξ {\\"V\\"} } ) 0])) \\" \") ([ (ξ {\\"T\\"
    ↪ } ) 0])") , S) as Z
32 set(π kernel @ S) as (ξ 0 @ S)
33 del V
34 del T
35 del M
36 del A
37 del N
38 (S)

```

Example 2. The usefulness of a nested morphism representation can be promptly shown with the example in Figure 5 while focusing on the morphism tables referring to the matching of the subject-verb.object structure of a sentence (Figure 5b). Each morphism can contain two distinct nested relationships, one referring to the subject (S) and one to the object (Z). The possibility of representing such values in a nested morphism allows us to better group vertices to be considered while referring to those with the same variable while keeping unique entry-point instances.

Example 3. With reference to the morphism resulting from matching (con-)/dis-junctions with a sentence (Figure 5c), entry-point grouping allows the creation of one single vertex matching as

a single subject for the sentence, thus ensuring the creation of one final single vertex per group of matched entry-points.



(a)

graph	V	n_label	S		l_label	Z	
∅	∅	nsubj	n	S	dobj	I	Z
			↑2	2		↑1	4
			↑3	3			

(b)

graph	X	*					
∅	∅	H	l_label	l	c_label	c	Z
		2	nsubj	↑2	cc	↑4	5

(c)

Figure 5. Applying the rewriting rules expressed in Figure 2 to the graph originally presented in Figure 3a: different colours refer to different matching rules. Filled vertices in the left (and right) graph refer to the distinct vertex entry-points (and newly generated components), while uparrows ↑ are used to differentiate containment IDs from the ones for the objects. (a) Generating a binary relationship between the subject as a single entity and the direct object. (b) Morphisms $M[p_3, g_0]$. (c) Morphisms $M[p_2, g_0]$, where * refers to sub-matches nested over the entry point (See Algorithm from Section 6.4).

We also show how the language allows us to break the declarativeness assumption when we want to specifically compose values according to a specific value composition function:

Example 4. The user is free to break this declarative assumption by directly specifying the order of combination when it is required to combine the values from different variables. This can be observed in a longer query considering more morphosyntactic language features, which is provided online (https://github.com/datagram-db/datagram-db/blob/v2.0/data/test/einstein/einstein_query.txt, accessed on 18 July 2024). This can be used to fully rewrite the database as per Figure 6. As the creation of the will-not-have containment in this will require combining values from vertices 3, 8, and 9 and, respectively, associated with variables V, A, and N, we can use a scripting language as a direct extension of Script v2.0 [34] for determining the order of strings’ composition. Please observe that this formulation, contrary to Neo4j’s APOC in v5.20, also supports optional object matches, where the values associated with non-existing NULL objects are resolved into empty strings (see Proof for Lemma 7 in Appendix B.3).

By explicitly expressing a containment relationship across the nested entry-point X via so-called hook containments defining equivalence classes, we split the nested morphism Γ into as many new morphisms as the equivalence classes in $\frac{Id_{X^r}(X)}{\mathbb{R}}$ (see Section 6.3).

Example 5. We appreciate the usefulness of such morphism splitting while looking at a more convoluted example, as the one considering the rewriting in the sentence depicted in Figure 6a: vertices Matt and Tray and play and have from conjunctions but at different branches of the

sentence structure. Furthermore, all four constituent vertices have the same containing vertex believe for which, if no hook relationship was considered, they would have been added within the same morphism as per the previous example.

Still, given that those vertices are associated with different conj as they appear in different coordinating conjunctions, we can use this as a hook relationship to distinguish those, for then obtaining two separate morphisms as illustrated by the first two rows in Figure 7b. Thus, hooks help in splitting nested entry-points structurally by identifying similar elements through structural information.

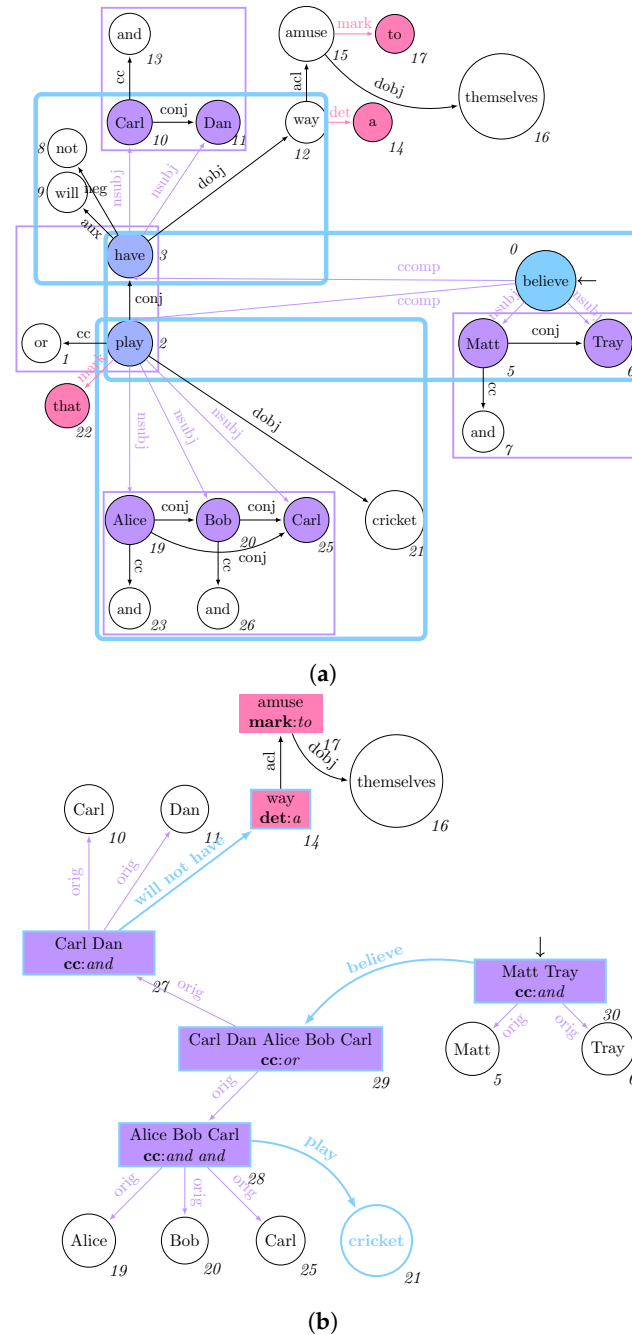


Figure 6. Applying the rewriting rules expressed in Figure 2: different colours refer to different graph grammar rules (*b* and *c*), filled vertices in the left (and right) graph refer to the distinct vertex entry-points (and newly generated components). (a) Dependency graph for “Matt and Tray believe that either Alice and Bob and Carl play cricket or Carl and Dan will not have a way to amuse themselves”. While object IDs are presented as numbers, containment IDs are omitted. (b) Generating a binary relationship between the subject as a single entity and the direct object.

graph	X	l_label	l	Y
1	2	mark	↑12	22
1	12	det	↑1	14
1	15	mark	↑13	17

(a)

graph	X	*					
1	0	H	l_label	l	c_label	c	Z
		2	ccomp	↑29	cc	↑22	1
		3	ccomp	↑30	NULL	NULL	NULL
1	0	H	l_label	l	c_label	c	Z
		5	nsubj	↑14	cc	↑23	7
		6	nsubj	↑15	NULL	NULL	NULL
1	2	H	l_label	l	c_label	c	Z
		19	nsubj	↑16	cc	↑25	23
		20	nsubj	↑17	cc	↑26	26
		25	nsubj	↑18	NULL	NULL	NULL
1	3	H	l_label	l	c_label	c	Z
		10	nsubj	↑19	cc	↑24	13
		11	nsubj	↑20	NULL	NULL	NULL

(b)

graph	V	n_label	S		l_label	Z		A	N	M
1	0	nsubj	n	S	ccomp	l	Z	NULL	NULL	NULL
			↑14	5		↑29	2			
			↑15	6		↑30	3			
1	2	nsubj	n	S	dobj	l	Z	NULL	NULL	22
			↑16	19		↑3	21			
			↑17	20						
			↑18	25						
1	3	nsubj	n	S	dobj	l	Z	9	8	NULL
			↑19	10		↑4	12			
			↑20	11						

(c)

Figure 7. Resulting morphisms from the application of the graph grammar rules from Listing 1 over the GSM database in Figure 6a, from which the resulting rewritten database Figure 6b is then obtained. (a) Morphisms $M[p_1, g_1]$. (b) Morphisms $M[p_2, g_1]$, where * refers to sub-matches nested over the entry point (See Algorithm from Section 6.4). (c) Morphisms $M[p_3, g_1]$.

Last, these examples provide an intuitive motivation for why the matching within our query language can express distances of at most one containment relationship from the entry-point match. We want to guarantee that, given two objects matching the query entry-points, located at different distances from the vertex appearing last within the reverse topological order, these are still reachable within the distance of crossing a single

containment. Similarly to semistructured data literature, we refer to one as its (direct) ancestor and to the other as its (direct) descendant. If the entry point considering in its match the aforementioned direct descendant object is replaced with another object, be it recently created during the application of the rewriting rules or already existing within the database, we want this information to be passed directly during the execution of the rewriting associated with the match of the object of the direct descendant. For this, we need both an explicit return mechanism, which allows the possibility of explicitly telling the objects appearing at the higher layers induced by the topological order that the previous object has been replaced, and to keep the match size compact, so that we can guarantee that any entry-point value updated at a lower level is retrieved immediately.

6.2. Determining the Order of Application of the Rules

We determine the application order of our language rules for each entry-point vertex of interest (Algorithm 5). This boils down to solving a scheduling problem, which requires first determining the interdependencies across the graph grammar rules. All the matching constructs L_i for each rule $L_i \rightarrow_{\Theta} R_i$ in our query gg have variables that might be shared across morphisms.

Algorithm 5 Sorting the Graph Grammar rules by application order

```

1: procedure SORTRULES( $gg$ )
2:    $V := \{p_i \mid p_i = L_i \rightarrow_{\Theta} R_i \in gg\}$ 
3:    $E := \left\{ p_i \rightarrow p_j \mid p_i.res \neq NULL \wedge p_i.res = p_j.entrypoint \vee \right.$ 
       $\left. p_i.entrypoint = p_j.entrypoint, p_i \in V, p_j \in V \right\}$ 
4:    $G := (V, E)$ 
5:    $time := LAYERFROMTOPOLOGICALSORT(G, APPROXV_{TOPO}(G))$  ▷ Algorithm 1
6:   sort each  $x$  in  $gg$  by  $time[x]$  in ascending order
7: end procedure

```

As per Example 1, each rewriting R_i might replace the entry-points with a single new object, or we preserve the previously matched ones otherwise. These are then input to any later morphism being considered while applying the rewritings. For this, we might consider the variables across patterns as hints to the query language on how the updated or matched objects are going to influence their updates, thus declaring their interdependencies. By reducing this to a dependency ordering, we consider the dependency graphs for the matching and rewriting rules, (Line 4), where each vertex represents a rule (Line 2). Each edge connecting two vertices (or patterns) represents a connection between the entry-point or returned variable from the source pattern and any other non-entrypoint variable occurring in the target pattern (Line 3). As the resulting graph might contain loops as some patterns might exhibit mutual dependencies, we are then forced to run an approximated topological sorting algorithm (Line 5) to determine an approximated scheduling order across such tasks through a DFS visit of the graph [52]: we start traversing the graph from the first rule appearing in the graph grammar while avoiding visiting edges leading to already-visited vertices; if we visited all the vertices reachable from such initial vertex while still having unvisited ones, we recommence the same visit starting from the earliest vertex that was not already visited. We add each visited vertex inside a stack after each of its children has been fully visited. By doing so, we prioritise the rules' declaration order which then acts as a heuristic for guiding the algorithm to decide upon a specific visiting order.

6.3. Containment Matching

With Algorithm 6, we define the steps realising containment matching for each L_i from a rule p_i to later on generate a morphism table $MT[L_i, g_j]$ per GSM database g_j , as discussed in Section 6.4. The algorithm works as follows: (i) after caching the PhiTable^K referenced by the containments in the matching patterns to minimize the tables' access in a

uniform schema specification, (ii) we specialise such tables to the specific schema induced by the variables' names and nesting occurring in the matching L_i . Last, (iii) we collect the matching containments by separating them between required or optional ones.

Algorithm 6 Intermediate Edge Result Caching

```

1: procedure UNIQUECACHEID( $c = (q, isOut), L$ )
2:   global queryCache, queryMap, emptySet
3:   if  $L = \emptyset$  then emptySet=emptySet $\cup\{c\}$  else
4:     for all  $l \in L$  do
5:       queryCache = queryCache $\cup\{l\}$ 
6:     end for
7:   end procedure
8: function FROMDB( $\kappa, db$ )
9:    $t := \langle \rangle$ ;  $\mathcal{S}(t) := [(dbid, \mathbb{N}), (src, ni), (edge, ci), (edgeLabel, \Sigma^*), (dst, ni)]$ 
10:  if  $\text{PhiTable}^\kappa \in db$  then
11:     $t := \langle [(dbid, i), (src, j), (edge, r), (edgeLabel, \kappa), (dst, d)] \mid$ 
12:       $r < |\text{PhiTable}^\kappa|, \text{PhiTable}^\kappa(r) = \langle l, i, j, w, d, i \rangle \rangle$ 
13:  end if
14:  return  $t$ 
15: end function
16: function TOTABLE( $t, x, y, l_x, nestCont$ )
17:  if  $l_x = \text{NULL}$  then  $t := R_{src, dst \rightarrow x, y}(\pi_{dbid, src, dst}(t))$ 
18:  else  $t := R_{src, edge, edgeLabel, dst \rightarrow x, l_x, l_x + \text{"_label"}, y}(t)$ 
19:  if  $nestCont$  then  $t := \nu_{l_x, y \rightarrow y}(t)$ 
20:  return  $t$ 
21: end function
22: procedure CACHEINTERMEDIATERESULTS( $gg, db$ )
23:  global queryCache, queryMap, emptySet
24:  for all  $p_i = L \rightarrow_{\Theta} R \in gg$  s.t.  $L \equiv \langle ep, in, out, join, hook \rangle$  do
25:    for all  $q \in in$  s.t.  $q = \langle u, l, l_x, all, opt \rangle$  do UNIQUECACHEID( $(q, \text{false}), l$ )
26:    for all  $q \in out$  s.t.  $q = \langle u, l, l_x, all, opt \rangle$  do UNIQUECACHEID( $(q, \text{true}), l$ )
27:    for all  $q \in join$  s.t.  $q = \langle u, l, l_x, all, opt, v \rangle$  do UNIQUECACHEID( $(q, \text{true}), l$ )
28:    queryCache = queryCache  $\cup$  hook
29:  end for
30:  if emptySet  $\neq \emptyset$  then queryCache=  $\{\kappa \mid \text{PhiTable}^\kappa \in db\}$ 
31:  cache :=  $\bigoplus_{x \in \text{queryCache}} [(x, \text{FROMDB}(x, db))]$ 
32:  for all  $p_i = L \rightarrow_{\Theta} R \in gg$  s.t.  $L \equiv \langle \langle ep, agg_{ep} \rangle, in, out, join, hook \rangle$  do
33:    hook $_i := \pi_{dbid, src, dst}(\bigsqcup_{j \in \text{hook}} \text{TOTABLE}(\text{cache}(i), src, dst, \text{NULL}, \text{false}))$ 
34:    for all  $q \in out, i \in \text{queryCache}(q, \text{true})$  s.t.  $q = \langle \langle u, agg_u \rangle, l, l_x, all, opt \rangle$  do
35:       $t := \text{TOTABLE}(\text{cache}(i), ep, u, l_x, all \text{ and } agg_u)$ 
36:      if  $opt$  then  $opt_i := opt_i \cup \{t\}$  else  $req_i := req_i \cup \{t\}$ 
37:    end for
38:    for all  $q \in in, i \in \text{queryCache}(q, \text{false})$  s.t.  $q = \langle \langle u, agg_u \rangle, l, l_x, all, opt \rangle$  do
39:       $t := \text{TOTABLE}(\text{cache}(i), u, ep, l_x, \text{false})$ 
40:      if  $opt$  then  $opt_i := opt_i \cup \{t\}$  else  $req_i := req_i \cup \{t\}$ 
41:    end for
42:    for all  $q \in join$  s.t.  $q = \langle \langle u, \text{false} \rangle, l, l_x, all, opt, \langle v, agg_v \rangle \rangle$  do
43:       $t := \text{TOTABLE}(\text{cache}(i), u, v, l_x, all \text{ and } agg_u)$ 
44:      if  $opt$  then  $opt_i := opt_i \cup \{t\}$  else  $req_i := req_i \cup \{t\}$ 
45:    end for
46:  end for

```

6.3.1. Pseudocode Notation for L_i

We describe the notation used in our pseudocode being the machine readable representation of the query syntax discussed in Section 6.1.

We define each object variable as a pair $\langle x, \mathbf{agg} \rangle \in \mathcal{N} = \Sigma^* \times \{0, 1\}$, where x is the variable name and \mathbf{agg} denotes whether the vertex should be aggregated (1) or not (0). In our pseudocode, each matching L_i is represented as the tuple $\langle ep, in, out, join, hook \rangle$, where $ep \in \mathcal{N}$ specifies the pattern entry-point and each ingoing (or outgoing) edge is represented by a pair $\langle u, l, l_x, all, opt, v \rangle$, where $u \in \mathcal{N}$ remarks the variable associated with the container (or contained) object alongside the containment relationship through ϕ and t_ϕ , $l \in \wp(\Sigma^*)$ provides a potentially empty-set of containment relationships, l_x is an optional containment variable, $all \in \{0, 1\}$ determines whether the edges should be considered in the aggregation or not, and $opt \in \{0, 1\}$ determines whether the match should be considered as optional or not. The *join* edges extend such records as $\langle u, l, l_x, all, opt \rangle$ by specifying both the containment ($v \in \mathcal{N}$) and container ($u \in \mathcal{N}$) variable explicitly, and $hook \in \wp(\Sigma^*)$ determines whether the aggregated entry-points over the single incoming container should be subdivided in equivalence classes according to the containment labels in *hook*, and we perform no clustering otherwise.

6.3.2. Procedural Semantics for Matching and Caching Containments

We now narrate the operations required to match each containment occurring across patterns while representing those as relational tables expressing either required (*req_i*), optional (*opt_i*), or hook-driven equivalence relationships (*hook_i*) per pattern p_i .

First, we define the semantics associated with the matching of each object ego-net as described in TOTABLE from Algorithm 6 (Line 15): either containment $(src) - [l_x : L] - (dst)$ or $(dst) <- [l_x : L] - (src)$ are represented as records with fixed schema (Line 9) where each record refers to a single containment ι (edge) in $\phi(src, \ell)$ for a container *src*, where $\langle dst, w \rangle = t_\phi(\iota)$ and *dst* refers to the contained object; in this, we also retain the containing db as *dbid* and the containment $\ell \in L$ (*edgeLabel*). At this stage, all containments are associated with the same schema and are not specialised to abide by a specific schema induced by a matching specification. This allows us to easily cache PhiTable^x containments (Line 30).

Next, we discuss how we specialise the results from the cache according to the schema induced by the variables occurring in each matching L_i . This is carried out by renaming generic containing/containment/contained object labels (*src/edge/dst*) with the variable names in L_i associated with them; if the patterns also contain references to the edge variable (l_x), we also retain each ID in $\phi_x(u, L)$ as l_x , and l_x 's label (Line 17) and we discard such information otherwise (Line 16). If the edge expresses an aggregation from the container to the content (e.g., $(src) - [\forall L] - (\gg dst)$), we nest l_x (if available) and *dst* from the table obtained at the previous step (Line 18). This makes the major difference with existing graph query languages, as we consider containment identifiers as a separate class from object ones (thus differently from SPARQL) and we also produce nested morphisms according to the query annotations.

Last, we collect the tables while differentiating where those are associated with a required or optional pattern (Lines 35, 39, and 43), acting as the basic building step for defining nested morphisms as in the subsequent section.

6.3.3. Algorithmic Choices for Optimisation

After discussing the procedural semantics of the matching of containments occurring across all L_i -s, we describe the algorithmic choices for optimisation purposes. First, to minimize the access to a relational database, we ensure that each PhiTable^x is accessed at most once across all the edges by collecting all the labels of interest across table-matched containments in *queryCache* (Line 5). If some containments have no containment attribute specified, we remember the existence of such containment (Line 3) for which we will then require considering all the PhiTable^x records, as a containment for which no label is

specified forces to consider all the containments (Line 29). Only after this, we access the physical model to transform each PhiTable^κ per $\kappa \in \text{queryCache}$ to a containment table to be then composed into the final nested morphism table, thus ensuring minimal memory access (Line 30).

6.4. Morphism Instantiation and Indexing

Algorithm 7 combines each table produced from the previous phase to generate the morphisms describing the result of matching L_i through the generation of morphisms being recorded within an $MT[L_i, g_i]$ table for each GSM database g_i . Similarly to SPARQL's triple navigation semantics, we generate the morphisms P_i for all GSM databases by natural joining all the tables associated with the mandatory containments (Line 10), while left-joining P_i (the resulting table from the natural join of the required containments) against the optional patterns set and updating P_i with such a result (Line 13).

Algorithm 7 Morphism instantiation and indexing

```

1: procedure INSTANTIATEMORPHISMS( $gg, G = \{g_1, \dots, g_n\}$ ) ▷ global  $MT$ 
2:   for all  $p_i = L_i \rightarrow_{\Theta} R_i \in gg$  do
3:     global  $\text{req}_i, \text{opt}_i, \text{hook}_i$  ▷ Algorithm 6
4:      $\bar{e} := L.\text{entrypoint}$ 
5:     if  $|\text{req}_i| = 0$  then continue
6:     sort each  $t$  in  $\text{req}_i$  by  $|t|$  in ascending order
7:      $P_i := \text{req}_i(0)$ 
8:     for  $j = 1$  to  $|\text{req}_i| - 1$  do
9:       if  $|P_i| = 0$  then break
10:       $P_i := \text{NESTEDNATURALEQUIJOIN}_{\text{false}}(P_i, \text{req}_i(j))$ 
11:    end for
12:    if  $|P_i| = 0$  then continue
13:     $P_i := \Lambda(\text{NESTEDNATURALEQUIJOIN}_{\text{true}}, P_i, \text{opt}_i)$ 
14:    if  $L.\text{entrypoint} = (\gg z)$  and  $\exists x, \text{obj}. \langle -[\forall x: \dots] - \text{obj} \in \text{cont}$  then
15:       $\bar{e} := \text{obj}$ 
16:       $P_i := v_S(p_i) \setminus \{\text{dbid}, x, x\_label\} \rightarrow_{\star} (P_i)$ 
17:       $\text{hook} := \text{TRANS}(\text{SYMM}(\text{REFL}(\text{hook}_i)))$ 
18:       $\mathfrak{R} := \{\mathfrak{R}_i | g_i \in GF \wedge t_{\mathfrak{R}_i}.s \Leftrightarrow (i, t(z), s(z)) \in \text{hook}\}$ 
19:      for all  $\Gamma \in P_i$  and  $\gamma_i \in \Gamma(\star) / \mathfrak{R}_{\Gamma}(\text{dbid})$  do
20:         $MT[L_i, \Gamma(\text{graph})].\text{add}(\Gamma |_{\{\text{graph}, x, x\_label\}} \oplus [(\star, \gamma_i)])$ 
21:      end for
22:    else
23:      for all  $\Gamma \in P_i$  do  $MT[L_i, \Gamma(\text{graph})].\text{add}(\Gamma)$ 
24:    end if
25:    for all  $g_j \in G$  do
26:      sort each  $\Gamma$  in  $MT[L_i, g_j]$  by  $O_{\text{rtopo}}(\Gamma(\bar{e}))$  in ascending order
27:    end for
28:  end for
29: end procedure

```

We further nest the morphism if and only if the entry-point is aggregated via a single containment object obj (Line 14), for which we then nest in a fresh attribute \star all the attributes except the database where obj is contained, obj itself, and optionally the edge labels for the containment if the pattern exhibits its variable (Line 16).

Hooks derive an equivalence relationship \mathfrak{R}_i per GSM database g_i having a \star -nested morphism through which to optionally split the morphisms. We retain only the container and containment relationship and their containing database ID (Line 32 from Algorithm 6), for then obtaining a suitable equivalence relationship \mathfrak{R}_i by computing the reflexive, symmetric, and transitive closure of that relationship (Line 17). Then, we potentially split the table nested in \star according to the equivalence classes associated with each equivalence relationship \mathfrak{R}_i obtained from hooks (Line 19) and update the morphism table accordingly.

Concerning the composition of cached tables, to reduce the equi-join time we first sort the tables by increasing size (Line 6) for then stopping the joins as soon as we find or compute an empty morphism, thus entailing that no collection of objects across all dbs can match L_i (Lines 5 and 9). As a last optimisation, we populate MT collecting the morphisms by database ID and rule ID (Lines 20 and 23). Last, we sort each morphism in $MT[L_i, g_j]$ by entry-point (Line 4) reverse topological order (Line 26) or, if these were nested in \star , by their container object (Line 15). This induces a primary block indexing mapping each elected vertex \bar{e} to a set of morphisms containing it.

6.5. Graph Rewriting Operations (op from R_i)

Finally, we apply the transformation operations required by the rewriting side of the rule for each instantiated morphism in MT across all GSM databases. This works by keeping track of the desired changes within a GSM view per loaded GSM database.

We now discuss Algorithm 8. For updating GSM views, we apply the rewriting query for each database g_j as described by the rewriting patterns R_i in gg (Line 2): we visit per GSM database its objects according to the reverse topological order from $O_{\text{rtopo}}(g_i)$ (Line 4) while retaining the objects v appearing in the aforementioned primary block index of a non-empty morphism table $M[L_i, g_j]$ for each production rule $p_i = L_i \rightarrow_{\Theta} R_i \in gg$ (Line 5): we skip the morphisms Γ associated with v if either a previously matched vertex was deleted and not replaced with a new one (Line 6), or if the current morphism Γ does not satisfy a possible WHERE condition Θ associated with L_{Θ} (Line 8). For the remaining morphisms, we run the operations listed in R_i in order of appearance (Line 9).

Algorithm 8 Rewriting phase

```

1: function GENERATEGRAPHVIEWS( $gg, G = g_1, \dots, g_n$ )
2:   for all  $g_i \in G$  do
3:      $\Delta(g_i) := \text{new GraphView}(|V(g_i)|)$ 
4:     for all  $v \in O_{\text{rtopo}}(g_i)$  s.t.  $v \notin O_i^-$  do ▷ Section 4.1
5:       for all  $p_i = L_i \rightarrow_{\Theta} R_i \in gg$  s.t.  $MT[L, g_i] \neq \emptyset$  do
6:         for all  $\Gamma \in MT[L, g_i]$  s.t.  $\forall \text{col}. \neg \text{Optional}_L(\text{col}) \Rightarrow \Gamma(\text{col}) \notin O_i^-$  do
7:            $\Delta(g_i) := \text{START}_{\Delta(g_i)}(\Gamma)$  ▷ Equation (9)
8:           if  $\Theta \neq \text{true}$  and  $|\llbracket \Theta \rrbracket_{\Gamma, g_i}^{1, MT}| = 0$  then continue ▷ Figure A2
9:            $\langle \langle \rangle, T, p, \Delta(g_i), MT := v(R, T, p, \Delta(g_i), MT) \rangle$  ▷ Figure 8
10:        end for
11:     end for
12:   end for
13:   yield  $\Delta(g_i)$ 
14: end for
15: end function

```

We now discuss the definition of v through SOS semantics enclosed within the evaluation of Line 9 in detail. Figure 8 describes the interpretation of all rewriting rules R_j updating the GSM view $\Delta(g_i)$, where the first three updates and exploit the functions from Section 4.3. All the y -s are interpreted as evaluated expressions without intermediate assignments. Rule NEWOBJECT creates a new object and refers it to the variable j . Rule DEOBJECT deletes a pre-existing or a newly-created object, and Rule DEOBJECT deletes a single container-containment relationship that was defined at loading time. We can easily distinguish between variables referring to objects or containments by the simple type associated with the attribute. For now, we do not allow the explicit removal of containments at run-time unless the containment is explicitly specified via containment update.

We discuss the `set` update for the vertices' label values; we distinguish the following cases: if both the number of variable and values are in the same number, we assign for each i -th variable the u -th occurring resolved value (LABELZIP); otherwise, we assign to each object associated with the resolved variable the collapsed string values (LABELVALFLAT).

We can obtain similar rules for ζ which are omitted here for conciseness, but that can be retrieved in the appendix (Figure A4).

$$\begin{array}{c}
\frac{\langle \text{"}\iota\text{"}, \Gamma, p, \text{NEWOBJ}_j(\Delta(g)), MT \rangle \rightarrow_v R}{\langle \text{"new } j; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v R} \text{NEWOBJECT} \\
\\
\frac{\boxed{\text{TIDX}_\Gamma(j) = \text{ni}} \quad \langle \text{"}\iota\text{"}, \Gamma, p, F(\text{DELOBJ}, \Delta(g), \text{IDX}_\Gamma(j)), MT \rangle \rightarrow_v R}{\langle \text{"del } j; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v R} \text{DELOBJECT} \\
\\
\frac{\boxed{\text{TIDX}_\Gamma(j) = \text{ci}} \quad \langle \text{"}\iota\text{"}, \Gamma, p, F(\text{DELCONT}, \Delta(g), \text{IDX}_\Gamma(j)), MT \rangle \rightarrow_v R}{\langle \text{"del } j; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v R} \text{DELCONT} \\
\\
\frac{\langle x, \Delta(g), \text{true} \rangle \rightarrow_q \langle \text{Var}, \Delta(g'), I \rangle \quad \langle y, \Delta(g'), \text{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, J \rangle \quad \boxed{|\text{Var}| = |\text{Val}|} \quad \text{idx} \in \mathbb{N}}{\langle \text{"}\iota\text{"}, \Gamma, p, F((x, \langle i, u \rangle) \mapsto \text{UPDATE}_x^\ell(\langle i, \text{idx} \rangle, u), \Delta(g'), \zeta(\text{Var}, \text{Val})), MT \rangle \rightarrow_v R}{\langle \text{"set } \ell \text{ idx @ } x \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v R} \text{LABELZIP} \\
\\
\frac{\langle x, \Delta(g), \text{true} \rangle \rightarrow_q \langle \text{Var}, \Delta(g'), I \rangle \quad \langle y, \Delta(g'), \text{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, J \rangle \quad \boxed{|\text{Val}| \neq |\text{Var}|} \quad \text{idx} \in \mathbb{N}}{\langle \text{"}\iota\text{"}, \Gamma, p, F((x, i) \mapsto \text{UPDATE}_x^\ell(\langle i, \text{idx} \rangle, \lambda(\text{Val}))), \Delta(g'), \text{Var}), MT \rangle \rightarrow_v R}{\langle \text{"set } \ell \text{ idx @ } x \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v R} \text{LABELVALFLAT} \\
\\
\frac{\langle z, \Delta(g), \text{true} \rangle \rightarrow_q \langle \text{Var}, \Delta(g'), I_R \rangle \quad \langle t, \Delta(g'), \text{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Name}, I_N \rangle \quad \langle y, \Delta(g'), \text{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, I_L \rangle \quad \boxed{I_R = I_N = I_L}}{\langle \text{"}\iota\text{"}, \Gamma, p, F((x, \langle a, b, c \rangle) \mapsto \text{UPDATE}_x^\pi(\langle a, b \rangle, c), \Delta(g), \zeta(\text{Var}, \text{Name}, \text{Val}))), MT \rangle \rightarrow_v R}{\langle \text{"set } \pi t @ z \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v R} \text{PI3ZIP} \\
\\
\frac{\langle z, \Delta(g), \text{true} \rangle \rightarrow_q \langle \text{Var}, \Delta(g'), I_R \rangle \quad \langle t, \Delta(g'), \text{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Name}, I_N \rangle \quad \langle y, \Delta(g'), \text{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, I_L \rangle \quad \boxed{I_R = I_L}}{\langle \text{"}\iota\text{"}, \Gamma, p, F((x, \langle b, a, c \rangle) \mapsto \text{UPDATE}_x^\pi(\langle a, b \rangle, c), \Delta(g), \text{Name} \times \zeta(\text{Var}, \text{Val}))), MT \rangle \rightarrow_v R}{\langle \text{"set } \pi t @ z \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v R} \text{PI2ZIP} \\
\\
\frac{\langle z, \Delta(g), \text{true} \rangle \rightarrow_q \langle \text{Var}, \Delta(g'), I_R \rangle \quad \langle t, \Delta(g'), \text{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Name}, I_N \rangle \quad \langle y, \Delta(g'), \text{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, I_L \rangle \quad \boxed{I_R = I_N = -1 \vee I_R = I_N \wedge I_L = -1}}{\langle \text{"}\iota\text{"}, \Gamma, p, F((x, \langle a, b \rangle) \mapsto \text{UPDATE}_x^\pi(\langle a, b \rangle, \lambda(\text{Val}))), \Delta(g), \zeta(\text{Name}, \text{Var}))), MT \rangle \rightarrow_v R}{\langle \text{"set } \pi t @ z \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v R} \text{PI2'ZIP} \\
\\
\frac{\langle z, \Delta(g), \text{true} \rangle \rightarrow_q \langle \text{Var}, \Delta(g'), I_R \rangle \quad \langle t, \Delta(g'), \text{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Name}, I_N \rangle \quad \langle y, \Delta(g'), \text{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, I_L \rangle \quad \boxed{I_L = I_N \wedge I_R = -1}}{\langle \text{"}\iota\text{"}, \Gamma, p, F((x, \langle b, c \rangle) \mapsto \text{UPDATE}_x^\pi(\langle \text{Var}(0), b \rangle, c), \Delta(g), \zeta(\text{Name}, \text{Val}))), MT \rangle \rightarrow_v R}{\langle \text{"set } \pi t @ z \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v R} \text{PI2''ZIP}
\end{array}$$

Figure 8. Cont.

$$\begin{array}{c}
\frac{\langle z, \Delta(g), \mathbf{true} \rangle \rightarrow_q \langle \text{Var}, \Delta(g'), I_R \rangle \quad \langle t, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Name}, I_N \rangle}{\langle y, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, I_L \rangle \quad \boxed{I_L = I_N \wedge I_R = -1}} \\
\frac{\langle \text{"}\iota\text{"}, \Gamma, p, \text{UPDATE}_{\Delta(g)}^\pi(\langle \text{Var}(0), \lambda(\text{Name}) \rangle, \lambda(\text{Val})), MT \rangle \rightarrow_v R}{\langle \text{"set } \pi t @ z \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v R} \text{PIALLFLAT} \\
\frac{\langle z, \Delta(g), \mathbf{true} \rangle \rightarrow_q \langle \text{Var}, \Delta(g'), I_R \rangle \quad \langle t, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Name}, I_N \rangle}{\langle y, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, I_L \rangle \quad \boxed{I_L = I_N \neq -1 \wedge I_R = -1}} \\
\frac{\langle \text{"}\iota\text{"}, \Gamma, p, F((x, y) \mapsto \text{UPDATE}_x^\pi(\langle \text{Var}(0), y \rangle, \lambda(\text{Val})), \Delta(g), \text{Name}), MT \rangle \rightarrow_v R}{\langle \text{"set } \pi t @ z \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v R} \text{PIVAREXT} \\
\frac{\langle z, \Delta(g), \mathbf{true} \rangle \rightarrow_q \langle \text{Var}, \Delta(g'), I_R \rangle \quad \langle t, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Name}, I_N \rangle}{\langle y, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, I_L \rangle \quad \boxed{I_N = -1 \vee I_V = -1}} \\
\frac{\langle \text{"}\iota\text{"}, \Gamma, p, F((x, (a, b)) \mapsto \text{UPDATE}_x^\pi(\langle a, b \rangle, \lambda(\text{Val})), \Delta(g), \text{Id} \times \text{Name}), MT \rangle \rightarrow_v R}{\langle \text{"set } \pi t @ z \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v R} \text{PIEXTOTH} \\
\frac{p = \langle \langle p', \text{agg} \rangle, \text{in}, \text{out}, \text{join}, \text{hook} \rangle}{\langle \text{"}(p')\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v \langle \text{"}\text{"}, \Gamma, p, \Delta(g), MT \rangle} \text{NOREWR} \\
\frac{p = \langle \langle q, \mathbf{false} \rangle, \text{in}, \text{out}, \text{join}, \text{hook} \rangle \quad q \neq p'}{\langle p', \Delta(g), \mathbf{false} \rangle \rightarrow_o \langle \text{Repl}, \Delta(g), I \rangle \quad \langle q, \Delta(g), \mathbf{false} \rangle \rightarrow_o \langle \text{Orig}, \Delta(g), J \rangle} \\
\frac{\Delta' := F((x, (y, z)) \mapsto \text{REPLOBJ}_x(y, z), \Delta(g), \text{Orig} \times \text{Repl})}{\langle \text{"}(p')\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v \langle \text{"}\text{"}, \Gamma, p, \Delta', MT \rangle} \text{NOAGGREWR} \\
\frac{p = \langle \langle q, \mathbf{false} \rangle, \langle \langle \langle \text{cont}, \mathbf{false} \rangle, l, l_x, \text{all}, \text{opt}, v \rangle \rangle, \text{out}, \text{join}, \text{hook} \rangle \quad q \neq p'}{\langle p', \Delta(g), \mathbf{false} \rangle \rightarrow_q \langle \text{Repl}, \Delta(g), I \rangle \quad \langle q, \Delta(g), \mathbf{false} \rangle \rightarrow_q \langle \text{Orig}, \Delta(g), J \rangle} \\
\langle \text{cont}, \Delta(g), \mathbf{false} \rangle \rightarrow_q \langle \text{Cont}, \Delta(g), K \rangle \\
C := \{ \kappa \in \Sigma^* \mid \exists o \in K. \exists l \in \downarrow \rho_{\Delta(g)}^\phi(o). \downarrow \rho_{\Delta(g)}^{t\phi}(l) \cap \text{Orig} \neq \emptyset \} \\
\frac{\Delta' := F((x, (y, z)) \mapsto \text{REPLOBJ}_x(y, z), \Delta(g), \text{Orig} \times \text{Repl})}{\Delta'' := F((x, (y, z, t)) \mapsto \text{UPDATE}_x^\phi(\langle y, z \rangle, \text{Repl}), \Delta', \text{Cont} \times C)} \\
\frac{\Delta'' := F((x, (y, z, t)) \mapsto \text{UPDATE}_x^\phi(\langle y, z \rangle, \text{Repl}), \Delta', \text{Cont} \times C)}{\langle \text{"}(p')\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_v \langle \text{"}\text{"}, \Gamma, p, \Delta'', MT \rangle} \text{AGGREWR}
\end{array}$$

Figure 8. Graph Rewriting SOS for view update.

We treat the property π and the containment ϕ updates differently, as they deal with the resolution of three variables. We are also interested in whether different resolved variables belong to the same nested table within the morphism or not, with the rationale being that we can freely associate each value within the same nesting row-by-row (P2ZIP, P2'ZIP, and P2''ZIP), while we need to compute the cross-product between the assignments if the value belongs to distinct nestings (Clearly in P3ZIP). This is determined via the second parameter of expression evaluation \rightarrow_E (Appendix A.3) by transferring the attribute information originated by the variable resolution \rightarrow_ρ (Appendix A.1). In all the remaining occasions, we arbitrarily decide to flatten the associated expressions. Please observe that, if the querying user wants more control over the precise value to be associated, they can always refer to SCRIPT expressions, thus breaking the declarative assumptions for a more controlled output.

Even in this case, we can formulate the rules for setting the ϕ -s similarly to our previous discussion for the π -s. Therefore, we defer to Figure A5 provided in the Appendix A.4.

We conclude by analysing the semantics associated with the replacement via R_i 's return statement, the last and mandatory operation for declared rewriting operations. We apply no rewriting if the returned variable is the variable entry-point (NOREWR). Otherwise, if the entry-point variable is not aggregated, we resolve the replacement and entry-point (Repl and Orig, respectively) and we replace any object in Orig associated with

the entry-point variable p' with an object in Rep1 associated with the replacing variable p' (NOAGGRREWR). Otherwise (AGGRREWR), the rewriting occurs by replacing the objects in Orig , associated with the entry-point's container and the objects in Rep1 , associated with the returned variable p' . Furthermore, given C the containment labels for which the entry-point is contained by its aggregating object in cont , we also update the containing for the cont objects to also contain via C the replacing objects in Rep1 . As this provides the final update, we then consider this last resulting GSM view of the resulting view for our rewriting step.

As no SOS rule matches the empty string, no further operation is conducted, and the rewriting program terminates after considering the final rewriting statement.

7. Language Properties

Given the previous language description, we want to characterise its properties by juxtaposing them with Cypher's. Full proofs are provided in Appendix B.3. We start by showing that, unlike current graph query languages, we propose a rewriting language framed as *generalised graph grammars*: we relate our proposed language to the graph grammars as, similarly to these, the absence of a matched pattern leads to no view updates. Still, we claim that such language provides a generalisation of the former by supporting explicit data-aware update operations over the objects and containments, while also defining explicit semantics determining the order of the application of such rules, both across rules and within each GSM database.

Lemma 6. *If either we query all the GSM databases with no rules, or all the rules have no rewritings, or none of the matches returned a morphism, or none of the ones being matched pass the associated Θ condition, then we return the same GSM databases as the ones being originally loaded.*

Next, we ensure that the rules are applied in reverse topological order, thus minimising the restructuring cost of the GSM database while achieving declarativeness for rule application, as the user does not specify this within the query formulation, as no matched pattern leads to no view updates.

Property 1. *The rules are applied in (reversed) topological order while visiting each GSM.*

On the other hand, we can show that Cypher forces the user to specify in which order the updates shall be applied, thus breaking the declarative assumption for a good query language.

Lemma 7. *The application of the rewriting in Cypher in lexicographical order requires explicitly determining the order of the morphisms' application and the order of the graph's visit.*

Next, we state the usefulness of an explicit return statement within the interpretation of a rule as it allows us to propagate the updates on the subsequently evaluated morphisms.

Lemma 8. *If an entry-point match is deleted and a new object is re-created inheriting all of its properties, this new object will not be connected to the entry point's containers unless the newly-returned object was applied.*

To a minor extent, we also show a greater amount of declarativeness if compared to current graph query languages by automatically generating a new object if an update operation occurs with reference to an optionally matched variable that was not matched to an existing object.

Lemma 9. *The setting of properties to variables not associated with a newly-created object variable and not associated with an actual GSM object due to an optional match reduces to the creation of a new object associated with the variable, which then always receives this and any following update associated with the same variable.*

At this point, it could be argued that, although our proposed rewriting language performs queries that cannot be expressed in other graph query languages, this does not return the matched subgraphs as in such other languages, similar to Cypher's return statement due to the considerations from Lemma 6. The following Lemma shows otherwise. Thanks to this, this language is considered more expressive than Cypher.

Lemma 10. *The proposed graph grammar query language can express traversal queries retaining only the objects and containments being matched and traversed.*

Corollary 1. *The proposed graph query language is more expressive than current graph query languages.*

8. Time Complexity

We now study the computational complexity associated with the algorithms discussed in the previous section and infer this from the implementation details discussed while reasoning on the SOS discussion. Proofs are postponed to Appendix B.4. Please observe that, as previously noted from previous graph query language literature (Section 3.1.2), the following results do not intend to prove P vs NP, as we are deliberately expressing a sub-problem of the more generic subgraph isomorphism problem that can be easily captured through algebraic operations.

Lemma 11. *The time complexity of sorting the rules within the query is quadratic over the number of query rules.*

Lemma 12. *The intermediate edge result caching time complexity has a polynomial cost being linear in both the loaded databases and the number of available objects.*

Corollary 2. *The cost for generating the nested morphisms is polynomial with the size of the entire data loaded within the physical model.*

Lemma 13. *The rewriting cost is polynomial and linear with the number of rewriting operations and the number of the morphisms.*

Corollary 3. *The time complexity of the whole Generalised Graph Grammars is polynomial with the size of the loaded physical model.*

9. Empirical Evaluation

For our empirical evaluation, we study the use case of graph grammar in the context of rewriting graphs representing the grammatical structure of a natural-language sentence. Universal dependencies [53] capture these syntactical features across languages by exploiting a shared annotation scheme. In this context, the usual approach to graph rewriting boils down to rewriting a PROLOG-like logical program by applying declarative rewriting rules (<https://github.com/opencog/relex> accessed on 22 April 2024) via a unification algorithm [54], where *compound terms* are equivalently expressing binary relationship and properties associated with specific vertices. Given the general interest for such an approach within the domain of Natural Language Processing (NLP), the present paper is going to specifically focus on use case scenarios within such domain. This will also give us the opportunity to re-use freely available datasets for sentences for our experiments (https://osf.io/btjqw/?view_only=f31eda86e7b04ac886734a26cd2ce43d and <https://osf.io/rpu37/>, accessed on 21 April 2024), which can be then deemed as repeatable.

Notwithstanding the previous approaches, we want to achieve a more data-driven approach to sentence rewriting, where atoms can also be associated with properties and labels, thus motivating the definition of the proposed query language. Furthermore, the extension of the graph grammar language with a script can be used to compose data and define boolean conditions allowing us to break the declarativeness assumption only when the user wants more control over how data are processed. Thus, we postulate that our proposed query language extends the current literature in several aspects way beyond graph database literature while postulating the possibility of applying concurrently multiple rewriting rules over disparate sentences via a sole declarative query. The proposed approach shares some similarities with programming language research where, after representing a program in terms of its operational semantics, we can apply graph rewriting rules over *abstract semantic graphs* [55], which are usually represented as trees, for which similar considerations like the former can be applied.

We test the implementation of our physical model and associated query language as implemented in our novel object-oriented database, named *DatagramDB*, which source code associated with the current paper is freely available online (<https://github.com/datagram-db/datagram-db/releases/tag/v2.0>, accessed on the 22 April 2024). We conducted all our benchmarks on a Dell Precision mobile workstation 5760 running Ubuntu 22.04 LTS. The specifications of the machine include an Intel® Xeon(R) W-11955M CPU @ 2.60 GHz x16, 64 GB DDR4 3200 MHz RAM.

9.1. Comparing Cypher and Neo4j with Our Proposed Implementation

Given the problems being evidenced by the Cypher query language from Lemma 7, we cannot automate the benchmarking of Cypher for all the possible sentences coming from an online available dataset. By recalling the results of this lemma, we observe that, when the query is not able to capture a pattern albeit optionally, this will have a cascade effect on the entire query for which none of the following rewriting operations will be applied. Given this, the same query might not necessarily provide the same output across different sentences having a different sentence structure. Thus, we cannot use one single query for dealing with unpredictable sentence structure that, in the worst-case scenario, would require us to write one single query per input sentence. We then preferred to limit our comparison to two sentences, for which we designed the minimal Cypher query exactly capturing the syntactical features expressed by these sentences while using the same query for *DatagramDB* across all the sentences.

We considered two distinct dependency graphs: “*Alice and Bob play cricket*”, the one in Figure 3a, and the one resulting from the dependency parsing of the “*Matt and Tray...*” sentence from Figure 6a. We loaded them in both Neo4j v5.20 and our proposed GSM database. In Cypher, we then run the query as formulated in the previous section, while we construct a fully declarative query in our proposed graph query language syntax directly representing an extension of the patterns in Figure 2. From now on, when referring to Neo4j, we will always refer to version 5.20.

Examining Table 1, we can see our solution consistently outperforms the Neo4j solution by two orders of magnitude. Furthermore, the data materialisation phase does not significantly impact the overall running time, as its running times are always negligible compared to the other ones. Additionally, Neo4j does not consider a materialisation phase, as the graph resulting from the graph rewriting pattern is immediately returned and stored as a distinct connected component of the previously loaded graph. This then clearly remarks the benefit of the proposed approach for rewriting complex sentences into a more compact machine representation of the dependency graphs.

Table 1. Table displaying results from rewriting the aforementioned sentences.

Data Model		Loading/Indexing (avg. ms)	Querying (avg. ms)	Materialisation (avg. ms)	Total (ms)
Neo4j	Simple	1.83×10^0	7.03×10^0	N/A	8.86×10^0
	Complex	9.32×10^0	1.93×10^2	N/A	2.02×10^2
GSM	Simple	9.63×10^{-2}	4.82×10^{-1}	2.40×10^{-2}	6.02×10^{-1}
	Complex	6.91×10^{-1}	9.00×10^{-1}	6.67×10^{-1}	2.26×10^0

9.2. Scalability for the Proposed Implementation

For testing the scalability of our implemented system, we used a corpora of sentences used for natural-language common-sense question answering [56] which we rewrote into dependency graphs using Stanford NLP [57]. As its output is failing systematically at correctly recognising verbs in passive form when at the present time and at recognising negations due to its training-based nature [58], we provide a piece of software amending its output so that all the syntactic and the semantic information retained within the sentence could pertain in a graph structure. This server also transforms the internal Stanford NLP dependency graph into a collection of GSM databases serialised in textual format. The resulting server is available online (https://github.com/datagram-db/stanfordnlp_dg_server accessed on 19 April 2024).

Given the resulting GSM representation of the two sentences, we provide two distinct scalability tests: in the first one, we sample the dataset into sentences to determine the algorithm’s scalability in terms of both the number of GSM databases and the number of vertices, while in the second we will only consider scalability regarding the sole number of GSM databases while maintaining the sample distribution of the sentence’s lengths, thus reflecting the number of GSM objects within the database.

Concerning the first scenario, we choose the sentences containing $|O_i| \in \{5, 10, 15, 18\}$ words, and for each of them we choose 300 sentences each, thus obtaining sample sets $S_{300}^{|O_i|}$. Then, we further sample $S_{300}^{|O_i|}$ into three distinct subsets $S_i^{|O_i|}$ having cardinality of $|S_i^{|O_i|}| = 75 \cdot i$ for which $S_i^{|O_i|} \subset S_{i+1}^{|O_i|}$ for $n > 0$ and $1 \leq i < i + n \leq 4$. This will be useful to then plot the rewriting running times using for the x -axis either the number of sentences (or GSM databases) or the sequence length, to better analyze the overall time complexity. The results for these experiments are reported in Figure 9.

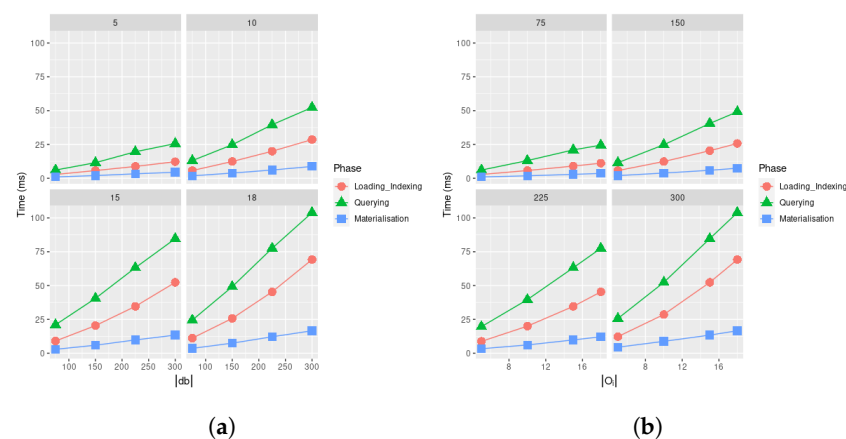


Figure 9. Analysing DatagramDB. (a) Results grouped by number of words per sentence. (b) Results grouped by number of databases.

From these plots, we can clearly remark that querying and materialisation times are clearly linear over the size of objects being loaded or GSM databases being matched and rewritten, thus remarking the efficiency for the overall envisioned approach: please observe that we cannot achieve a better time complexity that a linear scan without additional

heuristics, as we still have to perform the visit over each GSM database by also performing a linear scan of the database objects in reverse topological (layered) order. We can also motivate these results by having graphs in which the branching factor is relatively small compared to the overall number of available vertices, thus $\beta \ll |O_i|$ for each GSM database g_i . We also observe that, for these smaller datasets, the resulting materialisation time is almost negligible compared to the query time, which, across the board, dominates the loading and indexing time.

By comparing such running times with the ones from Neo4j, we can easily observe that, while we were able to process 300 GSM databases in 100 milliseconds, Neo4j could rewrite just one single graph in double time. Given this, our approach has a throughput of almost 600 times over Neo4j. This further remarks the impracticality of using the competing solution for analysing more sentences in the future, e.g., while considering sentences crawled from the web.

While it might initially seem that all phases (loading, querying, and materialisation) exhibit linear time complexity, we will try to consider a larger set of data to better outline the time complexity associated with our implementation.

Concerning the second scenario, we decided to sample the internet dataset in a subset of sentences S_1, \dots, S_4, S where $|S_i| = 10^i$. S represents the entire dataset while ensuring that each dataset of a lesser size is always contained in the larger dataset. Furthermore, we ensure that the number of words, and therefore, of objects on each sampled database reflects a similar frequency distribution of the number of objects per resulting GSM database (Figure 10). By doing so, for our final scalability tests in which we consider more data, we make up for the lack of long sentences with the number of sentences reflected in the number of the GSM databases to be processed.

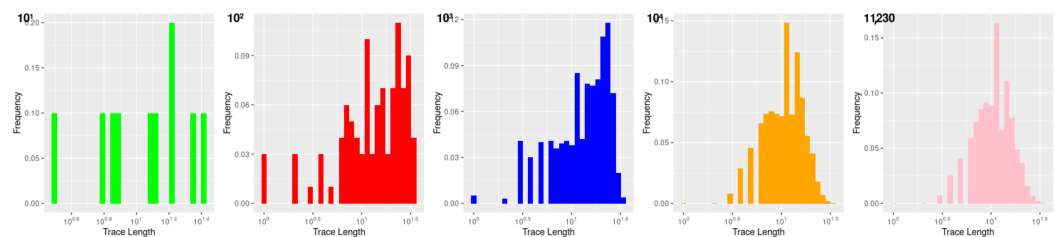


Figure 10. Sampled probability density function associated with the number of words within the sentences for each subset of traces.

Figure 11 provides the benchmarks for these experiments: a non-linear but polynomial loading time might be possibly related to the data parsing time and the time to store such data in primary memory, while the remaining running times keep a linear time complexity concerning the increase in the number of the GSM databases to be rewritten, similarly to the previous experiments. Querying time always dominates in indexing time by at least one order of magnitude, thus showing that most of the significant computations occur while matching and rewriting. Materialisation times are on the same order of magnitude as indexing time, thus also showing that this cost does not exceed the actual querying phase. Overall, the efficiency of our system is also reflected by a linear querying time for the datasets being considered.

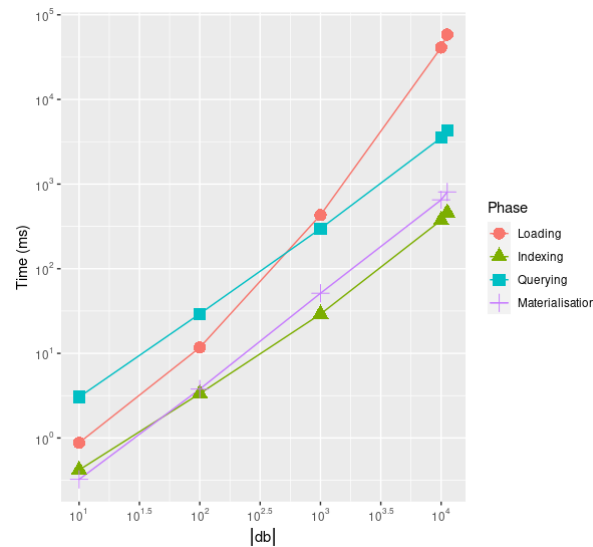


Figure 11. Running time of each algorithm with different sentence samples.

10. Conclusions and Future Works

This paper offers the definition of a matching and rewriting mechanism similar to the one provided by graph grammar implemented over object-oriented databases. As our definition supports both data matches and complex data update operations over the objects of interest, whose features were not considered in previous graph grammar formulations, we name our proposed language Generalised Graph Grammars. Our theoretical results prove the impossibility of expressing the same query with the same degree of generality of Cypher, which requires specifying a different query for any potential graph to be queried, thus posing a major limitation to automating rewriting operations over graphs. Empirical results prove that our query language offers an implementation faster than the ad-hoc query formalised in Cypher and run over Neo4j v5.20, in terms of both running time and throughput expressed in the number of queries runnable per comparable amount of time. These results aim to observe the inadequacy of graph-centric implementations of data representations since a large amount of literature now agrees in stating that more traditional and relational representations often offer better performances with respect to queries both natively supported by graph languages [59] and in representing new operations on graphs that require their rewriting [6,7].

At this stage, we considered nested morphisms of at most depth of 1, thus requiring that each cell of a morphism table should contain at most one non-nested table. Future works will investigate whether there might be any benefit for further generalising this nesting to arbitrary depth, thus requiring further extending the definition of the Nested Natural Equi-Join operator to arbitrary depths.

Notwithstanding the possibility of the current model expressing uncertainty within the data, the rewriting operations always assume to deal with perfect information, thus generating objects or containments containing no uncertainty. Future works will address this gap by further extending the SOS semantics of these rewriting steps while considering provenance information [44], thus relieving the user from explicitly defining the uncertainty of the generated data by adding further declarativeness to the query language here proposed.

Although Lemma 10 showed that the proposed graph query language is also able to remove the unmatched objects and contents, our current algorithm is not tailored for effectively matching this case, as the removing pattern will be forced to return all the objects and containments for then removing them. Future works will extend this by optimising testing conditions for our general language while matching the objects and containments. As a direct consequence of this, our returned morphisms are not pruned after testing the Θ condition, which is just evaluated in the rewriting phase due to the fact that any updates

to the GSM views will also alter the outcome of Θ . Future works will use static analysis approaches to determine when Θ can be effectively used for pruning morphisms before being returned in the matching phase, and condition rewriting strategies to push condition evaluations towards the generation of the morphism as discussed in Sections 6.3 and 6.4.

Although the current language supports the update of objects and containments within GSM objects, the provided query semantics do not consider the possibility that such updates can be still matched by the query, thus triggering additional rewriting operations. Future works will also consider further generalising the database matching and rewriting approach by considering the fix-point over the loaded database until convergence is met, thus meeting the former desiderata. We will also consider extending our containments with property-values associations similarly to the property graph model, and considering updating the objects' and containments' costs while performing the rewriting operations.

Last, preliminary experiments [34] conducted on a physical model being the direct mapping of the logical model in memory provide promising results showcasing the possibility of expressing not only JSON files but also representing indexing data structures in GSM that eventually lead to scalable query processing for JSON data. Future works will also compare the efficiency of DatagramDB if compared to other databases supporting object-oriented representations such as MongoDB or MySQL [60].

Author Contributions: Conceptualisation, G.B. and O.R.F.; methodology, G.B.; software, G.B. and O.R.F.; validation, G.B. and O.R.F.; formal analysis, G.B.; investigation, O.R.F.; resources, G.B.; data curation, O.R.F.; writing—original draft preparation, G.B.; writing—review and editing, O.R.F. and G.M.; visualisation, G.B. and O.R.F.; supervision, G.B. and G.M.; project administration, G.B.; funding acquisition, G.B. All authors have read and agreed to the published version of the manuscript

Funding: This research received no external funding.

Data Availability Statement: The datasets are available at the following repositories: https://osf.io/btjqw/?view_only=f31eda86e7b04ac886734a26cd2ce43d and <https://osf.io/rpu37/>. The codebase associated with the implementation of the data model and query language interpretation are available on GitHub: <https://github.com/datagram-db/datagram-db/releases/tag/v2.0>. All the URLs were accessed on the 21 April 2024.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

BNF	Backus-Naur Form
DAG	Direct Acyclic Graph
GSM	Generalised Semistructured Model
HOF	Higher-Order Function
NLP	Natural Language Processing
SOS	Structured Operational Semantics

Appendix A. Full Definitions

Appendix A.1. Variable Resolution

Before discussing value or boolean expression evaluation, we need to consider their base case first, which is the evaluation of variables occurring in such expressions. We, therefore, discuss these first before introducing the other steps. We consider variables referring to both objects and containments while supporting the update operations only for the latter. We, therefore, resolve the variables while returning the IDs for either matched objects or containments.

We, furthermore, want to achieve declarativeness while considering variables: if associated with NULL values as a result of a missed optimal match, we want to return empty values, while if we set values to them, we want to create implicitly new vertices. We need to then provide a new parameter to explicitly consider the creation of new objects when the expression context allows us to do so over unreferenced and unmatched variables. Furthermore, the single variable might be associated with multiple objects if referring to a

morphism attribute within a nested relationship. Since the resolution of variables within our language may require to access previously replaced or substituted values such as from $\Delta(g)$, we also have to refer to $\Delta(g)$ and g to carry out the variable resolution.

Since these operations can also involve updating the environment, we express these operations via SOS rather than using algebraic notation. Figure A1 shows the SOS for the sole variable resolution, returning a tuple containing (in order of appearance) the resolved variables, the possibly updated view due to object insertion, and the potentially nested attribute containing a nested table expressing the variable as an attribute.

If the variable belongs to the morphism’s schema and is associated with a non NULL value within the morphism while being associated with a basic type, we return the value stored in it (EXRES). If the variable refers to a nested attribute, we resolve the variable ($ID_{X_{\Gamma}}(x)$, Equation (11)) and return all the associated values via $\downarrow \rho_{\Delta(g)}^f$ (Definition 6, EXRESNEST). If the variable was declared within the rewriting pattern alongside the creation of a new object, we return the ID associated with the newly created object (NEWRES). If the variable is neither newly declared nor in the morphism’s schema, we return no result, as there is no binding and the query language is not expected to return a value (NORES). Last, if we require the object to be created ($new=true$) as we set values associated with an object and the morphism did not return an object ID associated with x due to an optional match, we then create a new object in the GSM view \bar{o} associated with a fresh value, while returning the updated view (FORCERES). This behaviour is completely disabled and no view is updated if the original expression does not force the creation of a new object due to an expression evaluation (NOFCRES). In all the other circumstances, we resolve no reference IDs (NOFCRES).

$$\begin{array}{c}
 \frac{x \in \text{dom}(\Gamma) \quad \mathcal{S}(\Gamma)(x) \in \mathcal{B}}{\langle x, \Delta(g), \text{new} \rangle \rightarrow_{\varrho} \langle \downarrow \rho_{\Delta(g)}(\Gamma(x)), \Delta(g), -1 \rangle} \text{EXRES} \\
 \\
 \frac{x \in \text{dom}(\Gamma) \quad \mathcal{S}(\Gamma)(x) \notin \mathcal{B}}{\langle x, \Delta(g), \text{new} \rangle \rightarrow_{\varrho} \langle \mu(\downarrow \rho_{\Delta(g)}, ID_{X_{\Gamma}}(x)), \Delta(g), ID_{NEST_{\mathcal{S}(\Gamma)}}(x) \rangle} \text{EXRESNEST} \\
 \\
 \frac{x \in \text{dom}(\Gamma^v)}{\langle x, \Delta(g), \text{new} \rangle \rightarrow_{\varrho} \langle \Gamma^v(x), \Delta(g), -1 \rangle} \text{NEWRES} \quad \frac{x \notin \text{dom}(\Gamma^v) \quad x \notin \text{dom}(\mathcal{S}(\Gamma))}{\langle x, \Delta(g), \text{new} \rangle \rightarrow_{\varrho} \langle \emptyset, \Delta(g), -1 \rangle} \text{NORES} \\
 \\
 \frac{x \notin \text{dom}(\Gamma) \quad x \in \mathcal{S}(\Gamma) \quad \mathcal{S}(\Gamma)(x) = \text{ni} \quad g' := \text{NEWOBJ}_{\Delta(g)}(x) \quad \bar{o} := \max g'.O}{\langle x, \Delta(g), \text{true} \rangle \rightarrow_{\varrho} \langle \bar{o}, g', -1 \rangle} \text{FORCERES} \\
 \\
 \frac{x \notin \text{dom}(\Gamma) \quad x \in \mathcal{S}(\Gamma) \quad \mathcal{S}(\Gamma)(x) \in \mathcal{B}}{\langle x, \Delta(g), \text{false} \rangle \rightarrow_{\varrho} \langle \emptyset, \Delta(g), -1 \rangle} \text{NOFCRES} \\
 \\
 \frac{x \notin \text{dom}(\Gamma) \quad x \in \mathcal{S}(\Gamma) \quad \mathcal{S}(\Gamma)(x) = \text{ci}}{\langle x, \Delta(g), \text{true} \rangle \rightarrow_{\varrho} \langle \emptyset, \Delta(g), -1 \rangle} \text{NOFCRES}
 \end{array}$$

Figure A1. Variable resolution with potential view updates (ϱ).

Appendix A.2. Predicate Evaluation (Θ)

We now discuss boolean predicate evaluation: given that the variables might refer to nested attributes referring to multiple objects or containment, for which we are interested in comparing the labels, values, or properties associated with them, we might be interested in returning not only one single boolean value but one single boolean value per comparison outcome. Given this, we need a notion of sets explicitly remarking that some elements were not inserted. A **maximum cardinality set** is a pair of a set S and a natural number n denoted as $\wr S, n \wr$ such that $|S| \leq n$. This type of set can be used to efficiently represent how

many elements satisfy a given condition if the number of elements is previously known. So, if $|S| < n$, we know that not all the n elements of interest satisfy a given condition. We also constrain such sets to contain at most n items. We can easily override traditional set operations over such sets as follows

$$\begin{aligned} \wr S, n \wr \cap \wr S', m \wr &= \wr S \cap S', \max(n, m) \wr \\ \wr S, n \wr \cup \wr S', m \wr &= \wr S \cup S', \max(n, m) \wr \\ \wr S, n \wr \setminus \wr S', m \wr &= \wr S \setminus S', \max(n, m) \wr \end{aligned}$$

We say a maximum cardinality set $\wr S, n \wr$ is **full** if the cardinality of S is equal to n : $\text{FULL}(\wr S, n \wr) \Leftrightarrow |S| = n \wedge n \neq 0$. We say that such maximum cardinality set is empty if S is empty: $\wr S, n \wr \simeq \emptyset \Leftrightarrow S = \emptyset$. The cardinality of the maximum cardinality set is the cardinality of its constituent set: $|\wr S, n \wr| = |S|$.

Figure A2 provides the definition providing the considered semantics. For this, we require that two original variables refer to the same cardinality of values, or that at least one of them is associated with one single value (TESTCOMP). For this, we can then return a maximal cardinality set $\wr S, M \wr$ where S indicates the resulting tuple indices associated with a true value, and M refers to the maximum size of the morphisms.

$$\llbracket \text{TEST } \mathit{script} \rrbracket_{\Gamma, g, \Delta(g)}^{MT} := \begin{cases} \wr S, |S| \wr & \text{eval}(\mathit{script}) \text{ holds, } S = \bigcup_{x \in SE(\Gamma)} \text{Idx}_{\Gamma}(x) \\ \wr \emptyset, M \wr & \text{oth.} \end{cases} \quad (\text{TestScript})$$

$$\llbracket \text{arg}_1 \leq \text{arg}_2 \rrbracket_{\Gamma, g, \Delta(g)}^{MT} := \text{let } t_1 := \eta(\text{arg}_1, \Delta(g), \Gamma, MT) \text{ and } t_2 := \eta(\text{arg}_2, \Delta(g), \Gamma, MT) \text{ in } \begin{cases} \wr \{i \in \mu(\leq, \zeta(t_1, t_2)) \mid i = \text{true}\}, |t_1| \wr & |t_1| = |t_2| \\ \wr \{i \in \mu(x \mapsto t_1(0) \leq x, t_2) \mid i = \text{true}\}, |t_2| \wr & |t_1| = 1 \\ \wr \{i \in \mu(x \mapsto x \leq t_2(0), t_2) \mid i = \text{true}\}, |t_1| \wr & |t_2| = 1 \end{cases} \quad (\text{TestComp})$$

$$\llbracket \text{FILL } \Theta \rrbracket_{\Gamma, g, \Delta(g)}^{MT} := \begin{cases} \wr S, |S| \wr & \left| \llbracket \Theta \rrbracket_{\Gamma, g, \Delta(g)}^{MT} \right| > 0, S = \bigcup_{x \in SE(\Gamma)} \text{Idx}_{\Gamma}(x) \\ \wr \emptyset, M \wr & \text{oth.} \end{cases} \quad (\text{TestFill})$$

$$\llbracket \text{X matched L.Y} \rrbracket_{\Gamma, g, \Delta(g)}^{MT} := \text{IDX}_{\Gamma}(X) \cap \bigcup_{\Gamma' \in M[L, g]} \text{IDX}_{\Gamma'}(Y) \quad (\text{TestMatch})$$

$$\llbracket \text{X unmatched L.Y} \rrbracket_{\Gamma, g, \Delta(g)}^{MT} := \text{IDX}_{\Gamma}(X) \setminus \bigcup_{\Gamma' \in M[L, g]} \text{IDX}_{\Gamma'}(Y) \quad (\text{TestUmatch})$$

$$\llbracket \Theta_1 \wedge \Theta_2 \rrbracket_{\Gamma, g, \Delta(g)}^{MT} := \llbracket \Theta_1 \rrbracket_{\Gamma, g, \Delta(g)}^{MT} \cap \llbracket \Theta_2 \rrbracket_{\Gamma, g, \Delta(g)}^{MT} \quad (\text{TestConj})$$

$$\llbracket \Theta_1 \vee \Theta_2 \rrbracket_{\Gamma, g, \Delta(g)}^{MT} := \llbracket \Theta_1 \rrbracket_{\Gamma, g, \Delta(g)}^{MT} \cup \llbracket \Theta_2 \rrbracket_{\Gamma, g, \Delta(g)}^{MT} \quad (\text{TestDisj})$$

Figure A2. Predicate Evaluation semantics $\llbracket \Theta \rrbracket_{\Gamma, g, \Delta(g)}^{MT}$.

Furthermore, we might also be interested in determining whether objects being matched in the current morphism also appear (TESTMATCH) or not (TESTUMATCH) in another morphism L associated with a variable y . Given that we are interested in the objects actually being matched and not in later changes provided by subsequent transformations, we can simply refer to $\text{IDX}_{\Gamma}(X)$ listing the reference IDs associated with an attribute X in Γ . This can be easily represented as a maximum cardinality set $\wr \{x \in \text{Idx}_{\Gamma}(X) \mid x \neq \text{NULL}\}, |\text{Idx}_{\Gamma}(X)| \wr$ by stripping the NULL values, while returning the intersection (TESTMATCH) or the difference (TESTUMATCH) for those IDs.

As the tests in the second last paragraph will return no sets whose indices are released to neither object nor containment ID while the former will return such indices, we need an intermediate predicate where these two set-boolean results are computed, to return for the former a full maximum cardinality set containing all the ID references from the morphism Γ if the underlying expression will not return an empty maximum cardinality set (TESTFILL).

We associate a similar behaviour to the typecasting of a script evaluation to a boolean value, for which we return an empty set if this is false and tall the occurring references within the morphism otherwise (TESTSCRIPT).

Last, we interpret the conjunction and the disjunction of such multivalued boolean semantics as the intersection (TESTCONJ) or the union (TESTDISJ) at the set of reference ID satisfying the base case candidates.

Appendix A.3. Expression Evaluation (expr)

Last, we are interested in evaluating expressions exploiting the variables withheld by each morphism. Please also observe that these expressions will have different operational semantics if compared to the ones associated with assignments, as the former will only retrieve values associated with the expressions, and the latter describe how to update the view with the newly assigned value. For this, these expressions will only return the final value associated with them. As the morphisms might be also nested, their final representation will be a one-column table with an arbitrary attribute, for which time is one of the basic types.

Figure A3 shows the SOS associated with the expression evaluation. From this, we provide η as a shorthand for the above relationship:

$$\eta(x, \delta, \Gamma, NT) := T \text{ s.t. } \langle x, \delta, \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle T, I \rangle$$

In other words, η computes the first projection of \rightarrow_E , being the evaluation of an expression x .

$$\begin{array}{c} \frac{\langle x, \Delta(g), \mathbf{false} \rangle \rightarrow_q \langle T, \Delta(g), I \rangle}{\langle \text{"x"}, \Delta(g), \mathbf{true}, \Gamma, MT \rangle \rightarrow_E \langle T, I \rangle} \text{VARE} \\ \frac{}{\langle \text{"x"}, \Delta(g), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle x, -1 \rangle} \text{STRE} \\ \frac{\text{TID}_{\Gamma}(x) = \text{ci} \quad \langle \text{"x"}, \Delta(g), \mathbf{true}, \Gamma, MT \rangle \rightarrow_E \langle V, I \rangle}{\langle \text{"label x"}, \Delta(g), f, \Gamma, MT \rangle \rightarrow_E \langle \mu(\ell_{g_i}, V), I \rangle} \text{LABELE} \\ \frac{\text{TID}_{\Gamma}(x) = \text{ni} \quad \text{idx} \in \mathbb{N} \quad \langle \text{"x"}, \Delta(g), \mathbf{true}, \Gamma, MT \rangle \rightarrow_E \langle V, I \rangle}{\langle \text{"\zeta idx@ x"}, \Delta(g), f, \Gamma, MT \rangle \rightarrow_E \langle \mu(\downarrow \rho_{\Delta(g_i)}^{\zeta}, V), I \rangle} \text{XIE} \\ \frac{\text{TID}_{\Gamma}(x) = \text{ni} \quad \text{idx} \in \mathbb{N} \quad \langle \text{"x"}, \Delta(g), \mathbf{true}, \Gamma, MT \rangle \rightarrow_E \langle V, I \rangle}{\langle \text{"\ell idx@ x"}, \Delta(g), f, \Gamma, MT \rangle \rightarrow_E \langle \mu(\downarrow \rho_{\Delta(g_i)}^{\ell}, V), I \rangle} \text{ELLE} \\ \frac{\langle \text{"x"}, \Delta(g), \mathbf{true}, \Gamma, MT \rangle \rightarrow_E \langle V, I \rangle \quad \langle \text{"str"}, \Delta(g), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle P, J \rangle}{|P| = |V|} \text{CONTZIPE} \\ \frac{}{\langle \text{"\phi str, x"}, \Delta(g), f, \Gamma, MT \rangle \rightarrow_E \langle \mu(\downarrow \rho_{\Delta(g_i)}^{\phi}, \zeta(P, V)), I \rangle} \end{array}$$

Figure A3. Cont.

$$\begin{array}{c}
\frac{\langle "x", \Delta(g), \mathbf{true}, \Gamma, MT \rangle \rightarrow_E \langle V, I \rangle \quad \langle "str", \Delta(g), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle P, J \rangle}{|P| = 1 \quad |P| < |V|} \text{ CONTE} \\
\langle "\phi \text{ str}, x", \Delta(g), f, \Gamma, MT \rangle \rightarrow_E \langle \mu(y \mapsto \downarrow \rho_{\Delta(g)}^\phi(y, P(0)), V), I \rangle \\
\frac{\llbracket \Theta \rrbracket_{\Gamma, g_i}^{1, MT} = \wr S, n \wr \quad |S| = n \quad \langle y, \Delta(g), f, \Gamma, MT \rangle \rightarrow_E \langle R, I \rangle}{\langle "\mathbf{if} \Theta \text{ over } x \text{ then } y \text{ else } z", \Delta(g), \mathbf{true}, \Gamma, MT \rangle \rightarrow_E \langle R, I \rangle} \text{ ALLTRUEE} \\
\frac{\exists M. \llbracket \Theta \rrbracket_{\Gamma, g_i}^{1, MT} = \wr \emptyset, M \wr \quad \langle z, \Delta(g), f, \Gamma, MT \rangle \rightarrow_E \langle R, I \rangle}{\langle "\mathbf{if} \Theta \text{ over } x \text{ then } y \text{ else } z", \Delta(g), \mathbf{true}, \Gamma, MT \rangle \rightarrow_E \langle R, I \rangle} \text{ ALLFALSEE} \\
\frac{\llbracket \Theta \rrbracket_{\Gamma, g_i}^{1, MT} = \wr T, n \wr \quad \langle x, \Delta(g), f, \Gamma, MT \rangle \rightarrow_E \langle X, I \rangle \quad \langle y, \Delta(g), f, \Gamma, MT \rangle \rightarrow_E \langle L, J \rangle \quad \langle z, \Delta(g), f, \Gamma, MT \rangle \rightarrow_E \langle R, K \rangle}{\langle "\mathbf{if} \Theta \text{ over } x \text{ then } y \text{ else } z", \Delta(g), \mathbf{true}, \Gamma, MT \rangle \rightarrow_E \langle \mu(i \mapsto \text{ifte}(X(i) \in T, L(i), R(i)), \zeta(\text{dom}(X), L, R)), I \rangle} \text{ ZIPE}
\end{array}$$

Figure A3. Expression evaluation (E) with no view update, where $\text{ifte}(x, y, z)$ is an expression returning y if x holds and z otherwise.

For the evaluation of the attribute associated with a containment ID (LABLE), we directly apply λ to all the non-NULL matches, as containment IDs are never updated in this version of the language. For extracting values (XIE) or labels (ELLE) associated with the object x at the numerical index idx not associated with an evaluated expression, we resort instead to the Property Resolution function also encompassing the changes in the GSM view (Definition 6). The interpretation of $\phi(x, \text{str})$ considers both x and str as expressions, where only the former is forced to be interpreted as a variable if is a string: if x and str are associated with a tuple of values V and P of the same cardinality, we then return $\phi(y, z)$ for $(y, z) \in \zeta(V, P)$ (CONTZIPE), and if otherwise $|P| = 1$ we return $\phi(y, P(0))$ for $y \in V$ (ContE).

Last, for conditional expressions, we first evaluate a condition Θ which, as per the forthcoming discussion, will return a set of object or containment IDs for which the entire expression holds. If such a set is full, we return the evaluation of the expression associated with the “then” branch (ALLTRUEE), if it is empty, we return the evaluation of the expression associated with the “else” branch (ALLFALSEE), and otherwise, we first interpret Θ over the IDs referenced by a variable x , and then return the i -th value from the left expression if the MES associated with Θ contains the i -th object reference after x , and the i -th value from the else branch otherwise.

If we need to evaluate the string as a variable as it appears as the leftmost argument of a label, ζ , ℓ , or ϕ , then we use variable resolution ρ (Appendix A.1 on page 42) to evaluate the values associated with the variable (VARE), and we consider this as a simple string otherwise (STRE). Please observe that this invented value is then associated with an unexisting nested morphism -1 .

Appendix A.4. Full SOS Rewriting Specifications

This section describes through Figures A4 and A5 the remaining set-update operations that were not reported in Figure 8 for conciseness. These refer to ζ updates (Figure A4), similar to the ones for ℓ , and ϕ (Figure A5), similar to the ones for π .

$$\begin{array}{c}
\langle x, \Delta(g), \mathbf{true} \rangle \rightarrow_{\varrho} \langle \text{Var}, \Delta(g') \rangle \quad \langle y, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_{\text{Val}} \\
|\text{Var}| = |\text{Val}| \quad \text{idx} \in \mathbb{N} \\
\langle \text{"}\iota\text{"}, F((x, \langle i, u \rangle) \mapsto \text{UPDATE}_x^{\xi}(\langle i, \text{idx} \rangle, u), \Delta(g'), \zeta(\text{Var}, \text{Val})), MT \rangle \rightarrow_{\nu} R \\
\hline
\langle \text{"set } \xi \text{ idx@ } x \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_{\nu} R \quad \text{XIZIP}
\end{array}$$

$$\begin{array}{c}
\langle x, \Delta(g), \mathbf{true} \rangle \rightarrow_{\varrho} \langle \text{Var}, \Delta(g') \rangle \quad \langle y, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_{\text{Val}} \\
|\text{Val}| \neq |\text{Var}| \quad \text{idx} \in \mathbb{N} \\
\langle \text{"}\iota\text{"}, F((x, i) \mapsto \text{UPDATE}_x^{\xi}(\langle i, \text{idx} \rangle, \lambda(\text{Val}))), \Delta(g'), \text{Var}, MT \rangle \rightarrow_{\nu} R \\
\hline
\langle \text{"set } \xi \text{ idx@ } x \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_{\nu} R \quad \text{XIVALFLAT}
\end{array}$$

Figure A4. Remaining setting functions for ξ -s being the extension of Figure 8.

$$\begin{array}{c}
\langle z, \Delta(g), \mathbf{true} \rangle \rightarrow_{\varrho} \langle \text{Var}, \Delta(g'), I_R \rangle \quad \langle t, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Name}, I_N \rangle \\
\langle y, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, I_L \rangle \quad I_R = I_N = I_L \\
\langle \text{"}\iota\text{"}, \Gamma, p, F((x, \langle a, b, c \rangle) \mapsto \text{UPDATE}_x^{\phi}(\langle a, b \rangle, c), \Delta(g), \zeta(\text{Var}, \text{Name}, \text{Val})), MT \rangle \rightarrow_{\nu} R \\
\hline
\langle \text{"set } (\varphi t, z) \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_{\nu} R \quad \text{PHI3ZIP}
\end{array}$$

$$\begin{array}{c}
\langle z, \Delta(g), \mathbf{true} \rangle \rightarrow_{\varrho} \langle \text{Var}, \Delta(g'), I_R \rangle \quad \langle t, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Name}, I_N \rangle \\
\langle y, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, I_L \rangle \quad I_R = I_L \\
\langle \text{"}\iota\text{"}, \Gamma, p, F((x, \langle b, a, c \rangle) \mapsto \text{UPDATE}_x^{\phi}(\langle a, b \rangle, c), \Delta(g), \text{Name} \times \zeta(\text{Var}, \text{Val})), MT \rangle \rightarrow_{\nu} R \\
\hline
\langle \text{"set } (\varphi t, z) \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_{\nu} R \quad \text{PHI2ZIP}
\end{array}$$

$$\begin{array}{c}
\langle z, \Delta(g), \mathbf{true} \rangle \rightarrow_{\varrho} \langle \text{Var}, \Delta(g'), I_R \rangle \quad \langle t, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Name}, I_N \rangle \\
\langle y, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, I_L \rangle \quad I_R = I_N = -1 \vee I_R = I_N \wedge I_L = -1 \\
\langle \text{"}\iota\text{"}, \Gamma, p, F((x, \langle a, b \rangle) \mapsto \text{UPDATE}_x^{\phi}(\langle a, b \rangle, \lambda(\text{Val})), \Delta(g), \zeta(\text{Name}, \text{Var})), MT \rangle \rightarrow_{\nu} R \\
\hline
\langle \text{"set } (\varphi t, z) \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_{\nu} R \quad \text{PHI2'ZIP}
\end{array}$$

$$\begin{array}{c}
\langle z, \Delta(g), \mathbf{true} \rangle \rightarrow_{\varrho} \langle \text{Var}, \Delta(g'), I_R \rangle \quad \langle t, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Name}, I_N \rangle \\
\langle y, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, I_L \rangle \quad I_L = I_N \wedge I_R = -1 \\
\langle \text{"}\iota\text{"}, \Gamma, p, F((x, \langle b, c \rangle) \mapsto \text{UPDATE}_x^{\phi}(\langle \text{Var}(0), b \rangle, c), \Delta(g), \zeta(\text{Name}, \text{Val})), MT \rangle \rightarrow_{\nu} R \\
\hline
\langle \text{"set } (\varphi t, z) \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_{\nu} R \quad \text{PHI2''ZIP}
\end{array}$$

$$\begin{array}{c}
\langle z, \Delta(g), \mathbf{true} \rangle \rightarrow_{\varrho} \langle \text{Var}, \Delta(g'), I_R \rangle \quad \langle t, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Name}, I_N \rangle \\
\langle y, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, I_L \rangle \quad I_L = I_N \wedge I_R = -1 \\
\langle \text{"}\iota\text{"}, \Gamma, p, \text{UPDATE}_{\Delta(g)}^{\phi}(\langle \text{Var}(0), \lambda(\text{Name}) \rangle, \lambda(\text{Val})), MT \rangle \rightarrow_{\nu} R \\
\hline
\langle \text{"set } (\varphi t, z) \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_{\nu} R \quad \text{PHIALLFLAT}
\end{array}$$

$$\begin{array}{c}
\langle z, \Delta(g), \mathbf{true} \rangle \rightarrow_{\varrho} \langle \text{Var}, \Delta(g'), I_R \rangle \quad \langle t, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Name}, I_N \rangle \\
\langle y, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, I_L \rangle \quad I_L = I_N \neq -1 \wedge I_R = -1 \\
\langle \text{"}\iota\text{"}, \Gamma, p, F((x, y) \mapsto \text{UPDATE}_x^{\phi}(\langle \text{Var}(0), y \rangle, \lambda(\text{Val})), \Delta(g), \text{Name}), MT \rangle \rightarrow_{\nu} R \\
\hline
\langle \text{"set } (\varphi t, z) \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_{\nu} R \quad \text{PHIVAREXT}
\end{array}$$

$$\begin{array}{c}
\langle z, \Delta(g), \mathbf{true} \rangle \rightarrow_{\varrho} \langle \text{Var}, \Delta(g'), I_R \rangle \quad \langle t, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Name}, I_N \rangle \\
\langle y, \Delta(g'), \mathbf{false}, \Gamma, MT \rangle \rightarrow_E \langle \text{Val}, I_L \rangle \quad I_N = -1 \vee I_V = -1 \\
\langle \text{"}\iota\text{"}, \Gamma, p, 1F((x, \langle a, b \rangle) \mapsto \text{UPDATE}_x^{\phi}(\langle a, b \rangle, \lambda(\text{Val})), \Delta(g), \text{Id} \times \text{Name}), MT \rangle \rightarrow_{\nu} R \\
\hline
\langle \text{"set } (\varphi t, z) \text{ as } y; \iota\text{"}, \Gamma, p, \Delta(g), MT \rangle \rightarrow_{\nu} R \quad \text{PHIEXTOTH}
\end{array}$$

Figure A5. Remaining setting functions for ϕ -s being the extension of Figure 8.

Appendix A.5. Converting GSM from Any Data Representation

Listing A1. Python code showing the conversion of an arbitrary Python object representation of some data to a GSM representation. In particular, we showcase the conversion of (possibly nested) Pandas dataframes, XML data, NetworkX graphs, and XML trees. As we also showcase the representation of arbitrary Python objects, thus including linear collections and dictionaries, we also support JSON conversion.

```

1 import xml
2 from collections import defaultdict
3 from collections.abc import Iterable
4 import xml.etree.ElementTree as ET
5 import gsm_stream_serialize
6 import networkx
7 from pandas import DataFrame
8
9
10 class MyCallable:
11
12     @abstractmethod
13     def call(self, *args, **kwargs)->object:
14         pass
15
16 # Notes: gsm_stream_serialize.gsm_object_xi_content refers to the creation of a containment relationship
17 #         gsm_stream_serialize.gsm_object           refers to the creation of a GSM object
18
19 def fullname(o):
20     klass = o.__class__
21     module = klass.__module__
22     if module == 'builtins':
23         return klass.__qualname__ # avoid outputs like 'builtins.str'
24     return module + '.' + klass.__qualname__
25
26 primitives = (bool, str, int, float, type(None))
27
28 def is_primitive(obj):
29     return type(obj) in primitives
30
31
32 import itertools
33 _cont = itertools.count()
34
35 def next_id():
36     return next(_cont)
37
38 def skipFor(i):
39     for x in range(i):
40         next(_cont)
41
42 class GSMSimpleSerializer:
43     def gsm_conversion(self, obj, id=None, ff="record", scoring=None, acc=None, label="label"):
44         if is_primitive(obj):
45             return self.basic_object(obj, id, scoring, acc)
46         elif isinstance(obj, ET.Element):
47             return self.xml_to_gsm(obj, id, scoring, acc)
48         elif isinstance(obj, networkx.classes.multidigraph.MultiDiGraph) or \
49             isinstance(obj, networkx.classes.digraph.DiGraph) or \
50             isinstance(obj, networkx.classes.graph.Graph) or \
51             isinstance(obj, networkx.classes.multidigraph.MultiGraph):
52             return self.graph_to_gsm(obj, id, scoring, acc, label)
53         elif isinstance(obj, DataFrame):
54             return self.list_to_gsm(obj.to_dict('records'), id, ff, scoring, acc)
55         elif isinstance(obj, dict):
56             return self.dict_to_gsm(obj, id, ff, scoring, acc)
57         elif isinstance(obj, list) or isinstance(obj, tuple):
58             return self.list_to_gsm(obj, id, ff)
59         else:
60             if obj is not None and hasattr(obj, '__dict__'):
61                 return self.dict_to_gsm(vars(obj), id, fullname(obj), scoring, acc)
62             else:
63                 return self.object_to_gsm(obj, id, scoring, acc)
64
65     def graph_to_gsm(self, G, id=None, scoring=None, acc=None, label="label"):
66         if isinstance(G, networkx.classes.digraph.DiGraph) or \
67             isinstance(G, networkx.classes.graph.Graph) or \
68             isinstance(G, networkx.classes.multidigraph.MultiGraph):
69             G = networkx.MultiDiGraph(G)
70         if acc is None:
71             acc = []
72         if id is None:
73             id = next_id()
74         if G is not None:
75             ell = [fullname(G)]
76         else:
77             ell = []
78         xi = []
79         if scoring is not None:
80             scores = scoring(G)
81         else:
82             scores = [1.0]
83         content = defaultdict(list)
84         pi = defaultdict(list)
85         mapp = dict()
86         if G is not None:
87             for node_id, node_data in G.nodes(data=True):
88                 xid = self.dict_to_gsm(node_data, None, str(node_id), scoring, acc).id
89                 mapp[node_id] = xid
90                 if scoring is not None:
91                     xscore = scoring(node_data)
92                 else:
93                     xscore = [1.0]
94                 orig_id = id
95                 containment = gsm_stream_serialize.gsm_object_xi_content(xid, xscore, orig_id, dict())

```



```

96         content["nodes"].append(containment)
97     for source, target, pi in G.edges(data=True):
98         edge_id = next_id()
99         if label is not None and label in pi:
100             edge_ell = [pi[label]]
101         else:
102             edge_ell = ["__label"]
103         edge_xi = []
104         edge_content = {"src": [gsm_stream_serialize.gsm_object_xi_content(mapp[source], [1.0], edge_id, dict())],
105                        "dst": [gsm_stream_serialize.gsm_object_xi_content(mapp[target], [1.0], edge_id, dict())]}
106         if scoring is not None:
107             xscore = scoring(pi)
108         else:
109             xscore = [1.0]
110         obj = gsm_stream_serialize.gsm_object(edge_id, edge_ell, edge_xi, xscore, edge_content, pi)
111         acc.append(obj)
112         containment = gsm_stream_serialize.gsm_object_xi_content(edge_id, xscore, id, dict())
113         content["edge"].append(containment)
114     obj = self.finalizeGeneration(acc, content, ell, id, pi, scores, xi)
115     return obj
116
117 def xml_to_gsm(self, root, id=None, scoring=None, acc=None):
118     assert isinstance(root, ET.Element)
119     if acc is None:
120         acc = []
121     if id is None:
122         id = next_id()
123     if root is None:
124         ell = []
125     else:
126         ell = [root.tag]
127     if scoring is not None:
128         scores = scoring(root)
129     else:
130         scores = [1.0]
131     if root.text is not None:
132         xi = [root.text.strip()]
133     else:
134         xi = []
135     pi = root.attrib
136     content = defaultdict(list)
137     for child in root:
138         self.extractContent(acc, content, child.tag, id, scoring, child)
139     obj = gsm_stream_serialize.gsm_object(id, ell, xi, scores, dict(content), pi)
140     acc.append(obj)
141     return obj
142
143 def dict_to_gsm(self, d, id=None, fullname="record", scoring=None, acc=None):
144     if acc is None:
145         acc = []
146     if id is None:
147         id = next_id()
148     ell = [fullname]
149     xi = []
150     if scoring is not None:
151         scores = scoring(d)
152     else:
153         scores = [1.0]
154     content = defaultdict(list)
155     pi = defaultdict(list)
156     if d is not None:
157         assert isinstance(d, dict)
158         for k, v in d.items():
159             self.extractField_global(acc, content, k, v, id, pi, scoring)
160     obj = self.finalizeGeneration(acc, content, ell, id, pi, scores, xi)
161     return obj
162
163
164 def list_to_gsm(self, ls, id=None, fullname="list", scoring=None, acc=None, fieldname="arg"):
165     if acc is None:
166         acc = []
167     if id is None:
168         id = next_id()
169     ell = [fullname]
170     xi = []
171     if scoring is not None:
172         scores = scoring(ls)
173     else:
174         scores = [1.0]
175     content = defaultdict(list)
176     pi = defaultdict(list)
177     if ls is not None:
178         allInXi = all(map(lambda x: (x is not None) and (isinstance(x, str) or
179                                isinstance(x, float) or
180                                isinstance(x, int) or
181                                isinstance(x, bool)), ls))
182
183         if allInXi:
184             for fromField in ls:
185                 if fromField is not None:
186                     xi.append(fromField)
187         else:
188             for fromField in ls:
189                 if fromField is not None:
190                     self.extractContent(acc, content, fieldname, id, scoring, fromField)
191     obj = self.finalizeGeneration(acc, content, ell, id, pi, scores, xi)
192     return obj
193
194 def finalizeGeneration(self, acc, content, ell, id, pi, scores, xi):
195     p2 = dict()
196     for k, v in pi.items():
197         if len(v) == 1:
198             p2[k] = v[0]
199         elif len(v) > 1:
200             p2[k] = "[" + ",".join(map(str, v)) + "]"
201     obj = gsm_stream_serialize.gsm_object(id, ell, xi, scores, dict(content), p2)
202     acc.append(obj)
203     return obj

```

```

204
205 def basic_object(self, obj, id=None, scoring=None, acc=None):
206     if acc is None:
207         acc = []
208     if id is None:
209         id = next_id()
210     if obj is None:
211         ell = [fullname(obj)]
212     else:
213         ell = []
214     xi = [str(obj)]
215     if scoring is not None:
216         scores = scoring(obj)
217     else:
218         scores = [1.0]
219     obj = self.finalizeGeneration(acc, defaultdict(list), ell, id, defaultdict(list), scores, xi)
220     return obj
221
222 def object_to_gsm(self, obj, id=None, scoring=None, acc=None):
223     if obj is not None and hasattr(obj, '__dict__'):
224         return self.dict_to_gsm(vars(obj), id, fullname(obj), scoring, acc)
225     if acc is None:
226         acc = []
227     if id is None:
228         id = next_id()
229     if obj is None:
230         ell = [fullname(obj)]
231     else:
232         ell = []
233     xi = []
234     if scoring is not None:
235         scores = scoring(obj)
236     else:
237         scores = [1.0]
238     content = defaultdict(list)
239     pi = defaultdict(list)
240     if obj is not None:
241         fields = [f for f in dir(obj) if not callable(getattr(obj, f)) and not f.startswith('__')]
242         for field in fields:
243             fieldName = field
244             fromField = getattr(obj, fieldName)
245             self.extractField_global(acc, content, fieldName, fromField, id, pi, scoring)
246     obj = self.finalizeGeneration(acc, content, ell, id, pi, scores, xi)
247     return obj
248
249
250 def extractField_global(self, acc, content, fieldName, fromField, id, pi, scoring):
251     if fromField is not None:
252         if isinstance(fromField, str) or isinstance(fromField, float) or isinstance(fromField, int) or isinstance(
253             fromField, bool):
254             pi[fieldName].append(fromField)
255         elif isinstance(fromField, Iterable):
256             for x in fromField:
257                 self.extractBasicField(acc, content, fieldName, id, pi, scoring, x)
258         else:
259             self.extractBasicField(acc, content, fieldName, id, pi, scoring, fromField)
260
261
262 def extractBasicField(self, acc, content, fieldName, id, pi, scoring, x):
263     if x is not None:
264         if isinstance(x, str) or isinstance(x, float) or isinstance(x, int) or isinstance(x, bool):
265             pi[x].append(x)
266         else:
267             self.extractContent(acc, content, fieldName, id, scoring, x)
268
269 def extractContent(self, acc, content, fieldName, id, scoring, x):
270     xid = self.gsm_conversion(x, None, ff="record", scoring=scoring, acc=acc).id
271     # xid = self.object_to_gsm(x, None, scoring, acc).id
272     if scoring is not None:
273         xscore = scoring(x)
274     else:
275         xscore = [1.0]
276     orig_id = id
277     containment = gsm_stream_serialize.gsm_object_xi_content(xid, xscore, orig_id, dict())
278     content[fieldName].append(containment)
279
280
281 class GSMSerializer(MyCallable):
282
283     def __init__(self, type, ff="record", scoring=None, label="label"):
284         super(GSMSerializer, self).__init__()
285         self.type = type
286         self.ff = ff
287         self.scoring = scoring
288         self.label = label
289         self.parent = GSMSimpleSerializer()
290
291     def skipFor(self, i):
292         self.parent.skipFor(i)
293
294     def call(self, obj, id=None) -> object:
295         assert isinstance(obj, self.type)
296         acc = []
297         res = self.parent.gsm_conversion(obj, id, self.ff, self.scoring, acc, self.label)
298         return acc, res
299
300
301 if __name__ == "__main__":
302
303     class Node:
304         def __init__(self, data):
305             self.left = None
306             self.right = None
307             self.data = data
308
309         def PrintTree(self):
310             print(self.data)
311

```

```

312     root = Node(10)
313     root.right = Node(120)
314     root.left = Node(5)
315     root.left.left = Node(1)
316     root.left.right = Node(7)
317     root.right.right = Node(130)
318
319     ser = GSMSerializer()
320     acc = []
321     # ser.object_to_gsm(root, acc=acc)
322
323     p = "" <?xml version="1.0"? >
324     <data>
325         <country name="Liechtenstein">
326             <rank>1</rank>
327             <year>2008</year>
328             <gdppc>141100</gdppc>
329             <neighbor name="Austria" direction="E"/>
330             <neighbor name="Switzerland" direction="W"/>
331         </country>
332         <country name="Singapore">
333             <rank>4</rank>
334             <year>2011</year>
335             <gdppc>59900</gdppc>
336             <neighbor name="Malaysia" direction="N"/>
337         </country>
338         <country name="Panama">
339             <rank>68</rank>
340             <year>2011</year>
341             <gdppc>13600</gdppc>
342             <neighbor name="Costa Rica" direction="W"/>
343             <neighbor name="Colombia" direction="E"/>
344         </country>
345     </data>""
346     root = ET.fromstring(p)
347     # ser.gsm_conversion(root, acc=acc)
348
349     # Create a Graph object
350     import networkx as nx
351     G = nx.MultiDiGraph()
352
353     # Add the nodes to the graph, with properties
354     G.add_node("Max", age=20, gender="male")
355     G.add_node("Alice", age=22, gender="female")
356     G.add_node("Bob", age=21, gender="male")
357
358     # Add the edges to the graph
359     G.add_edge("Max", "Alice", label="knows")
360     G.add_edge("Alice", "Max", label="knows")
361     G.add_edge("Alice", "Bob", label="knows")
362
363     ser.graph_to_gsm(G, acc=acc)
364     print(acc)

```

Appendix B. Proofs

Appendix B.1. Transformation Isomorphism

Proof (for Lemma 1). Proving this reduces to prove that $\langle g_i \rangle_{i \leq n} = \text{SERIALISATION}(\text{LOADING}(\langle g_i \rangle_{i \leq n}))$ and $db = \text{LOADING}(\text{SERIALISATION}(db))$. We can prove this by extending the definition of the transformations being given in Algorithm 2.

$\langle \tilde{g}_i \rangle_{i \leq n} = \text{SERIALISATION}(\text{LOADING}(\langle g_i \rangle_{i \leq n}))$: We can consider one GSM database \tilde{g}_i being returned at a time, for which we consider its constituents $\langle \tilde{O}_i, \tilde{l}_i, \tilde{\zeta}_i, \tilde{\epsilon}_i, \tilde{\pi}_i, \tilde{\phi}_i, \tilde{t}_{i,\phi} \rangle$. To do this, we want to show that such a returned database is equivalent to the originally loaded g_i . To conduct this, we need to show that each of the constituents is equivalent.

We can prove that $\tilde{O}_i = O_i$ as follows:

$$j \in \tilde{O}_i \Leftrightarrow \exists l, p, x. \langle l, i, j, p, x \rangle \in \text{ActivityTable} \Leftrightarrow j \in O_i$$

We can prove that $\tilde{l}_i = l_i$, $\tilde{\zeta}_i = \zeta_i$, and $\tilde{\epsilon}_i = \epsilon_i$ as follows:

$$\tilde{l}_i(j) = L_i(j) = l_i(j) \quad \tilde{\zeta}_i(j) = X_i(j) = \zeta_i(j) \quad \tilde{\epsilon}_i(j) = C_i(j) = \epsilon_i(j)$$

We can also prove that the properties are preserved:

$$\begin{aligned}
 \tilde{\pi}_i(j, \kappa) = v &\Leftrightarrow \langle l_i(j)[0], v, r \rangle \in \text{AttributeTable} \wedge \exists p, x. \text{ActivityTable}[r] = \langle l_i(j)[0], i, j, p, x \rangle \\
 &\Leftrightarrow \langle l_i(j)[0], v, r \rangle \in \text{AttributeTable} \wedge \text{ActivityTable}[r](2) = j \wedge j \in O_i \\
 &\Leftrightarrow \pi(j, \kappa) = v \wedge j \in \tilde{O}_i
 \end{aligned}$$

For $\tilde{\phi}_i(j, \kappa)$, we can easily show that this is associated with no value only if there are no records referring to a containment for j in the loaded database, which we can easily show that this derives from an originally empty contained from the loaded database g_i . On the other hand, this function returns a set S of identifiers only if there exists at least one containment record in PhiTable^κ , for which we can derive the following:

$$\begin{aligned} \tilde{\phi}_i(j, \kappa) = S &\Leftrightarrow S = \{ \iota \mid \exists l, w, d. \langle l, i, j, w, d, \iota \rangle \in \text{PhiTable}^\kappa \} \\ &\Leftrightarrow S = \{ \iota \mid \iota \in \phi_i(j, \kappa) \} \\ &\Leftrightarrow S = \phi_i(j, \kappa) \end{aligned}$$

Given that $\tilde{\phi}_i(j, \kappa)$ and $S = \phi_i(j, \kappa)$, this sub-goal is closed by transitivity closure over the equality predicate.

We can prove similarly the equivalence $\tilde{t}_{i,\phi} \doteq t_{i,\phi}$ as a corollary of the previous sub-cases:

$$\begin{aligned} \tilde{t}_{i,\phi}(\iota) = \langle d, w \rangle &\Leftrightarrow \exists l, j. \langle l, i, j, w, d, \iota \rangle \in \text{PhiTable}^\kappa \\ &\Leftrightarrow \iota \in \phi_i(j, \kappa) \wedge t_{i,\phi}(\iota) = \langle d, w \rangle \\ &\Leftrightarrow \iota \in \tilde{\phi}_i(j, \kappa) \wedge t_{i,\phi}(\iota) = \langle d, w \rangle \\ &\Leftrightarrow t_{i,\phi}(\iota) = \langle d, w \rangle \end{aligned}$$

$\overline{db} = \text{LOADING}(\text{SERIALISATION}(db))$: In this case, we need to show that each table in \overline{db} is equivalent to those in db . We can prove similarly to the previous step that $\overline{L} = L$, $\overline{X} = X$, and $\overline{C} = C$.

Next, we need to prove that the ActivityTables being returned are equivalent. We can achieve this by guaranteeing that both tables should contain the same records. After remembering that the values of p and x are determined through the indexing phase, from which we determine the offset where the objects with immediately preceding and following IDs are stored, we can then guarantee that these values will be always the same under the condition that the two tables will always contain the same records, for which these values will be the same. After remembering from the previous proof that $\tilde{l}_i(j) \in \equiv L_i(j) \in db$, for $\tilde{l}_i \in \text{SERIALISATION}(db)$ and $L \in db$, we can also have that $\tilde{l}_i(j)(0) \equiv L_i(j)(0)$:

$$\begin{aligned} \langle l, i, j, p, x \rangle \in \overline{\text{ActivityTable}} &\Leftrightarrow \tilde{g}_i \in db \wedge \tilde{l}_i(j)[0] = l \wedge j \in \tilde{O}_i \\ &\Leftrightarrow \tilde{g}_i \in db \wedge \tilde{l}_i(j)[0] = l \wedge \exists l', p', x'. \langle l', i, j, p', x' \rangle \in \text{ActivityTable} \\ &\quad \wedge L_i(j)[0] = l' \\ &\Leftrightarrow \exists p', x'. \langle l, i, j, p', x' \rangle \in \text{ActivityTable} \end{aligned}$$

Given that the first three components of the record always correspond, this entails that we will preserve the number of records across the board, hence preserving the same record ordering, thus also guaranteeing that $x = x'$ and $x = x'$.

By adopting the equivalence of the previous and next offset for the AttributeTable from the previous sub-proof, we can then also prove that each record of the $\overline{\text{AttributeTable}}$ always corresponds to an equivalent one in the AttributeTable.

$$\begin{aligned} \langle l, v, r \rangle \in \overline{\text{AttributeTable}}^\kappa &\Leftrightarrow \overline{\text{ActivityTable}}(r) = \langle l, i, j, p, x \rangle \wedge \tilde{\pi}_i(j, \kappa) = v \\ &\quad \exists l', r'. \langle l', v, r' \rangle \in \text{AttributeTable}^\kappa \wedge \text{ActivityTable}[r'] = \langle l', i, j, p, x \rangle \end{aligned}$$

Given that we can prove that $\tilde{l} = l$ also by the previous sub-case and $\tilde{r} = r$ as the records' index will always correspond after sorting and indexing, we can close this sub-goal.

Last, we need to prove that the PhiTable^k tables are equivalent, which can be closed as follows:

$$\begin{aligned} \langle l, i, j, w, d, \iota \rangle \in \overline{\text{PhiTable}}^k &\Leftrightarrow \ell_i(j)(0) = l \wedge \iota \in \phi_i(j, \kappa) \wedge t_{i,\phi}(\iota) = \langle d, w \rangle \\ &\Leftrightarrow \langle l, i, j, w, d, \iota \rangle \in \text{PhiTable}^k \end{aligned}$$

□

Appendix B.2. Nested Equi-Join Properties

Proof (for Lemma 2). If $L \neq \emptyset$ with $\vec{L} = \langle A_1, A_2, \dots, A_n \rangle$, then we can rewrite the definition of $\vec{L}_{\bowtie S}$ as follows:

$$t^{\vec{L}_{\bowtie S}} = \left\{ \tilde{t}|_{S \setminus A_1} \oplus (\tilde{t}(A_1)^{\langle A_2, \dots, A_n \rangle}_{\bowtie S}) \mid \tilde{t} \in t \right\}$$

If $A_1 \notin \text{dom}(\tilde{t})$, then $\tilde{t}(A_1) = \emptyset$ by assuming to represent a NULL value as an empty table by default. While doing so and by $\tilde{t}|_{S \setminus \{A_1\}} = \tilde{t}$ by former assumption, the former result can be rewritten as:

$$= \left\{ \tilde{t}|_{S \setminus \{A_1\}} \mid \tilde{t} \in t \right\} = t$$

□

Proof (for Lemma 3). Given the hypothesis and with reference to Algorithm 3, we have $IR = \emptyset$, which then yields $L \times R$ (Line 4). □

Proof (for Lemma 4). Given the hypothesis and with reference to Algorithm 3, this satisfies the condition at Line 5, for which we can then immediately close our goal. □

Proof (for Lemma 5). As $\text{dom}(S) \cap \text{dom}(U) = \{N\}$, this violates the condition for the rewriting Lemma 3, which is not then rewritten accordingly. Furthermore, the condition $\text{dom}(S(N)) \cap \text{dom}(U) \neq \emptyset$ violates the condition for the rewriting Lemma 4, which is then not applied. Given IR as defined in Line 3 of Algorithm 3, we show that this algorithm computes the following:

$$\left\langle \tilde{t}|_{IR} \oplus \left(\tilde{t}(N) \oplus \tilde{s}|_{U \setminus IR} \right) \mid \tilde{t} \in L, \tilde{s} \in R, \tilde{t} \dot{=}_{IR} \tilde{s} \right\rangle$$

This equates to equi-joining the tables over the IR records where $N \notin IR$ by construction, and by nesting all the records from the right table sharing the same IR values with the records from the left table. Last, we can easily observe that this cannot be easily expressed in terms of $L^{\langle N \rangle}_{\bowtie R}$, as the rewriting of the former expression in the following way, for which it is evident that the former operator did not consider the recursive natural joining of the records by considering the commonly-shared attributes during its descent:

$$\begin{aligned} L^{\langle N \rangle}_{\bowtie R} &= \left\langle \tilde{t}|_{S \setminus \{N\}} \oplus (\tilde{t}(N) \diamond_{\bowtie S}) \mid \tilde{t} \in t \right\rangle \\ &= \left\langle \tilde{t}|_{IR} \oplus (\tilde{t}(N) \oplus \tilde{s}) \mid \tilde{t} \in t, \tilde{s} \in R \right\rangle \end{aligned}$$

This can be easily observed by the lack of $\dot{=}_{IR}$ component that captures the essence of such a natural join condition. □

Appendix B.3. Language Properties

Proof (for Lemma 6). As by the conditions stated in the current Lemma we either generate an empty morphism table $MT[_,]$ (no rules, no rewritings, no matches) or all the generated morphisms are ignored as all the Θ values are falsified (Line 8 from Algorithm 8), then we will always have an empty view $\Delta(g_i)$ for all the GSM databases g_i loaded in the physical model. Considering this, the proof is an application of Equation (10), by which the materialisation of a GSM database g_i with a view $\Delta(g_i)$ containing no changes simply returns g_i . As g_i is stored in a physical model, this result is also validated via Lemma 1. □

Proof (for Property 1). This condition is generated by Line 4 from Algorithm 8, as we use the reverse topological order index $O_{\text{rtopo}}(g_i)$ associated with each GSM DB g_i to visit the entry-point objects or, when nested, their containing object associated with the reported morphism in $MT[L_i, G_i]$. \square

Proof (for Lemma 7). As per previous discussions, the proof for this lemma will be given intuitively by analysing the Cypher representation of the graph grammar represented visually in Figure 2 and previously represented by our proposed Generalised Graph Grammar language in Listing 1. We then provide the query required for rewriting the sentence expressed in Figure 6a:

The current Neo4j v5.20 implementation does not support the theorised graph incremental views for Cypher [32]. As we require to update the graph while querying, it is not possible to entirely create a new graph without restructuring or expanding a previously loaded one as per graph joins [7] or nesting [6] by simply returning some newly-created relationships or vertices; returning a new graph and rewriting a previous match will come at the cost of either restructuring the previously loaded graph, thus requiring additional overhead costs for re-indexing and updating the database while querying, or by creating a new distinct connected component within the loaded graph (CREATE statements from Listing A2). As it is impossible to refer by the vertices and edges through their ID, thus exploiting graph provenance techniques for mapping the newly created vertices to the ones from the previously loaded graph [45], we are forced to join the loaded vertices (e.g., Lines 35–37, 50–52, and 67) with the newly created ones (e.g., Lines 38, 53, and 68, respectively) by their property values (e.g., Lines 39, 54, and 70, respectively). Our proposed approach avoids such cost via the aforementioned objects' and containments' ID-based morphism representation while keeping track of the restructuring operations (property UPDATE, insertion NEWOBJ, deletion DELOBJ and DELCONT, and substitution REPLOBJ) over a graph g within an incremental view $\Delta(g)$ (Section 4.3).

Listing A2. Cypher representation for Figure 2 to rewrite the sentence from Figure 6a.

```

1  \Create the grouped vertices
2  MATCH (a)-[:subj]->(b)-[:cc]->(c)
3  WHERE (b)<-[:conj]-() OR (b)-[:conj]->()
4  WITH a, Collect(b.name) as names, c
5  WHERE size(names) > 1 // To make sure it is only grouping
   ↪ vertices with multiple names
6  OPTIONAL MATCH (a)-[:neg]->(d)
7  CREATE (x {name: apoc.text.join(names, ' '), cc: c.name, dobjRel
   ↪ : a.name})
8  WITH a, c, d, x, names
9  CALL apoc.create.addLabels(x, names) YIELD node
10 CALL apoc.do.when(
11 d IS NOT NULL,
12 'SET x.neg = d.name',
13 '',
14 {c:c, d:d, a:a, x:x})
15 YIELD value as~neg
16
17 // Create the origins
18 WITH neg
19 MATCH (n) // Match only vertices with no relationships
20 WITH LABELS(n) as nameLabels, Collect(DISTINCT n) as nameNodes
21 WHERE size(nameLabels) > 0
22 FOREACH (nNode IN nameNodes |
23 FOREACH (nLabel IN nameLabels |
24 CREATE (y {name: nLabel})<-[:orig]->(nNode)))

```

```

25
26 // Create dobj node
27 WITH *
28 MATCH (d)-[:dobj]-(a)
29 WITH Collect(DISTINCT d.name) as objects, a~FOREACH (obj IN
    ↪ objects |
30 CREATE (y {name: obj, dobjRel: a.name}))
31
32 // Check for 'det'
33 WITH *
34 MATCH (n) WHERE NOT (n)-[]->( ) AND NOT (n)-<[]-( )
35 WITH n
36 MATCH (d)-[:dobj]-(a)
37 MATCH (e)-[:orig]->( )
38 WHERE e.dobjRel = a.name AND d.name = n.name
39 OPTIONAL MATCH (d)-[:det]->(f)
40 CALL apoc.do.when(
41 f IS NOT NULL,
42 'SET n.det = f.name',
43 '',
44 {n:n, f:f})
45 YIELD value as~det
46
47 // Create relationships
48 WITH det
49 MATCH (m) WHERE NOT (m)-[]->( ) AND NOT (m)-<[]-( )
50 WITH m
51 MATCH (d)-[:dobj]-( )
52 MATCH (e)-[:orig]->( )
53 WHERE e.dobjRel = m.dobjRel
54 WITH DISTINCT e, m
55 CALL apoc.do.when(
56 e.neg IS NOT NULL,
57 'CALL apoc.create.relationship(e, e.neg + '' '' + e.dobjRel, {},
    ↪ m) YIELD rel RETURN rel',
58 'CALL apoc.create.relationship(e, e.dobjRel, {}, m) YIELD rel
    ↪ RETURN rel',
59 {e:e, m:m})
60 YIELD value as~dobjRel
61
62 // Create ? acl node
63 WITH dobjRel
64 MATCH (p) WHERE NOT (p)-[]->( ) AND NOT (p)-<[]-( )
65 WITH p
66 MATCH (a)-[:acl]->(b)
67 MATCH (c)-[:orig]-(d)-[]->(e)
68 WITH DISTINCT p, a, e
69 WHERE e.name = a.name
70 CREATE (y {name: '?'})-<[:acl]-(e)
71 WITH p, y
72 CALL apoc.create.relationship(y, p.dobjRel, {}, p) YIELD rel as~
    ↪ aclRel
73
74 // Group the groups

```

```

75 WITH ac1Rel
76 MATCH (a)-[:conj]->(b)-[:cc]->(c)<-[:cc]-(a)
77 MATCH ()<-[:orig]-(d)-[r]->()
78 WHERE type(r) CONTAINS a.name OR type(r) CONTAINS b.name
79 WITH a, b, c, d, r
80 ORDER BY d.name ASC
81 WITH Collect(DISTINCT d.name) as names, c
82 CREATE (x {name: apoc.text.join(names, ' ' + c.name + ' ')})
83 WITH x, names
84 CALL apoc.create.addLabels(x, names) YIELD node as~groupedGroup
85
86 // Add properties to rels
87 // mark
88 WITH groupedGroup
89 MATCH (c)<-[:mark]-(a)-[:dobj]->(b)
90 MATCH (d)-[r]->(e)
91 WHERE type(r) CONTAINS a.name AND b.name = e.name
92 CALL apoc.create.setRelProperty(r, 'mark', c.name) YIELD rel as~
    ↪ markRel
93
94 // aux
95 WITH markRel
96 MATCH (d)<-[:aux]-(a)-[:subj]->()-[:cc]->()
97 MATCH ()<-[:orig]-(f)-[r]->()
98 WHERE type(r) CONTAINS a.name
99 CALL apoc.create.setRelProperty(r, 'aux', d.name) YIELD rel as~
    ↪ auxRel
100
101 // Create rels to orig groups
102 WITH auxRel
103 MATCH (a)-[:orig]->(b)
104 MATCH (x) WHERE NOT (x)-[]->() AND NOT (x)<-[]-()
105 WITH DISTINCT a, x
106 WHERE apoc.text.join(LABELS(a), ' ') IN LABELS(x)
107 CALL apoc.create.relationship(x, 'orig', {}, a) YIELD rel as~
    ↪ origRel
108
109 // Rel from M+T to A+B+C|C+D
110 WITH x
111 MATCH (d)<-[:subj]-(a)-[:ccomp]->(b)-[:subj]->(c)
112 MATCH (e)-[:orig]->(f)
113 WITH a, b, c, d, e, f, x
114 ORDER BY d.name ASC, c.name ASC
115 WITH Collect(DISTINCT c.name) as endNames, x, Collect(DISTINCT d
    ↪ .name) as startNames, e, a~WHERE apoc.text.join(startNames
    ↪ , ' ') = e.name AND apoc.text.join(endNames, ' ') IN
    ↪ LABELS(x)
116 CALL apoc.create.relationship(e, a.name, {}, x) YIELD rel as~
    ↪ finalRel
117
118 WITH finalRel
119 MATCH q=()-[]->()
120 MATCH (z)

```



```

121 CALL apoc.create.removeLabels(z, LABELS(z)) YIELD node // Remove
      ↪ all labels from vertices
122 WITH q, z
123 RETURN q, z

```

Cypher does not ensure to apply the graph rewriting rules as intended in our scenarios: let us consider the dependency graph generated from the recursive sentence “*Matt and Tray believe that either Alice and Bob and Carl play cricket or Carl and Dan will not have a way to amuse themselves*” and let us try to express patterns in Figure 2b,c as two distinct **MATCH**-es with their respective update operations as per the following Listing:

```

1 MATCH (a)-[b:cc]->(c)
2 WITH Collect(a.name) as names, Collect(DISTINCT a) as nameNodes,
      ↪ c
3 CREATE (x {name: apoc.text.join(names, ' '), cc: c.name})
4 FOREACH (p IN nameNodes | CREATE (y {name: p.name})<-[:orig]->(x)
      ↪ )
5 WITH x
6 MATCH (a)-[:dobj]->(b)
7 CREATE (y {name: b.name})
8 WITH a, x, y
9 CALL apoc.create.relationship(x, a.name, {}, y) YIELD rel
10 WITH y
11 MATCH q=( )-[]->( )
12 RETURN q

```

Instead of generating one single connected component representing the result, we will generate as many distinct connected components as subgraphs being identified as matching the patterns, while this does not occur with a simple sentence structure (Figure 3a) where we achieve the correct result as in Figure 5. We must **MATCH** elements of the graph multiple times, constantly rejoining on data previously **MATCH**-ed in earlier stages of the query for establishing relationships over previously grouped vertices (Lines 108 and 118 from Listing A2). This then postulates the inability of such language to automatically apply an order of visit for restructuring the loaded graph (e.g., we need to tell the query language to first group-by the vertices—Lines 2–15—and then establish the *orig* relationships—Lines 18–24) while not expressing an automated way to merge each distinct transformed graph into one cohesive, connected component. This then forces the expression of a generic graph matching and rewriting mechanism to be dependent on the specific recursive structure of the data. Thus, requiring the creation of a broader query, where we need to explicitly instruct the query language on the correct way to visit the data while instructing how to reconcile each generated subgraph from each morphism within one final graph.

During the delineation of the final Cypher query succeeding in obtaining the correct rewritten graph (also Listing A2), we also highlighted the impossibility of Cypher propagating the temporary result generated by a rewriting rule and propagating it to another rule to be applied upstream: this requires carrying out intermediate sub-queries establishing connections across patterns sharing intermediate vertices, and the re-computation of the same intermediate solutions, such as vertex grouping (cfr. Line 4 and Line 116). Since Cypher also does not support the explicit grouping of vertices based on a pattern as in [46], this required us to identify the vertices satisfying each specific pattern, label them appropriately in a unique way (e.g. Line 9), and then compare the result obtained (e.g., Line 20 for generating *orig* relationships). This limitation can be overcome by providing two innovations: first, using nested relational tables for representing morphisms, where each nest will contain the sub-pattern of interest possibly to be grouped. Second, we track any vertex substitution for entry-point vertex matches via incremental views. This substitution can be easily propagated at any level by considering the transitive closure of the substitution

function (Definition 5), while the order of visit in the graph guarantees the correctness of the application of such substitution (Algorithm 8).

Listing A2, constructed for the specific matches referring to the sentence “Matt and Tray...”, will not fully execute on a different sentence without the given dependencies, as no match is found, and therefore, no rewriting can occur. Current graph query languages are meant to return a subgraph from the given patterns. In Cypher, you must abide by what is contained within the data, if the data are not there we need to remove the match from the query, which we cannot forecast in advance. This results in constant analysis of the data. For us, the intention is to have graph grammar rewriting rules whereby if a match is not made, no rewriting occurs.

By leveraging such limitations of Cypher while juxtaposing the desired behaviour of the language, we derive a declarative graph query language where patterns can be expressed similarly to Figure 5. □

Proof (for Lemma 8). This is a direct application of the SOS rules from Figure 8: any removed vertex will not be replaced by a newly inserted vertex within the matched entry-point ego-net if not explicitly updating the containment to also add the newly created object. If an entry-point was removed, the only way to preserve the connectivity of the GSM objects is to exploit the replacement, through which we will explicitly state that, for any explicitly declared container within the matched pattern, we will insert the created object or any other previously matched object of choice within the container’s containment relationships also containing the object. □

Proof (for Lemma 9). This requires discussing the SOS rules from Figure 8, from which we set or update values, labels, containments, and properties associated with objects. Concerning label updates, such updates occur over variable x , in which variable resolution ρ is always in the form $\langle x, \Delta(g), \mathbf{true} \rangle$: if the variable does not appear in the morphism, we expand the first two cases from Figure A1. We need to exclude the case it was declared with a new statement from Figure 8, as we will have otherwise x in Γ^v from $\Delta(g)$. As we have the parameter \mathbf{true} , this also excludes the rule NOFRES (Figure A1). We can then see that we do not create an optimally matched containment, as expected by the intended semantics. Thus, we restrict our case to FORCERES (Figure A1), on which we see that a new object is created, thus updating $\Delta(g)$, and that we know the ID of this object as it will be naturally the next incrementally number being available. Then, the label update will always occur, which will then preserve the update in $\Delta(g)$. These choices are reflected in the materialisation phase by extending each database g and prioritising the changes described in the view $\Delta(g)$. □

Proof. Given the possibility of visiting several patterns L_1, \dots, L_n , we can express the matching of those in our proposed query language as rules $p_i = L_i \rightarrow \emptyset$ for $1 \leq i \leq n$, where both objects and containments must be explicitly referenced with a variable. Still, this formulation will not explicitly remove any object or containment not being visited. Enabling this requires the extension of the former query with two additional rules, one for removing all the vertices not visited in the different pattern (om), and the other for explicitly removing unmatched containments (cm). Given the variables A^o, \dots, W^o referring to matched objects and A^c, \dots, W^c to matched containments in L_1, \dots, L_n , we can then express om and cm as the following rules being defined immediately after p_n :

```
om = (Z)
where (Z unmatched p1 . A^n ^ (... Z unmatched pn . W^n)
↪ del Z (Z);
```

```
cm = (X)--[Y:]->(Z)
where (Y unmatched p1 . A^c ^ (... Y unmatched pn . W^c)
↪ del Z (Z);
```

As we rewrite the same matching object, no replacement will occur and given that the matching (Z) and $(X) \dashrightarrow [Y:] \rightarrow (Z)$ will return all the objects and containments across the databases, we have to further test those to delete only the ones being not matched in L_1, \dots, L_n . \square

Proof (for Corollary 1). This follows from our previous proof, for which we clearly showed that our proposed language can match and rewrite graphs declaratively while considering optional rewrites. Cypher has some limitations in this regard, as it forces the user to specify the order in which the matching and rewriting rules should be applied. Furthermore, our language can return the matched morphisms similarly to SPARQL while allowing the generation of multiple morphism tables rather than just one, and selecting just the objects and containments being matched while removing the remaining ones similarly to Cypher. Therefore, the proposed language over GSM generalizes over current graph query languages over a novel generalised semistructured model enabling this. \square

Appendix B.4. Time Complexity

Proof (for Lemma 11). Regarding Algorithm 5, as we defined a graph connecting each rule appearing in the query, which will be then represented as a vertex, in the worst-case scenario, we will have a fully connected graph with $E = V \times V$. Thus, the cost of creating this graph is quadratic, as $O(|V| + |V|^2)$ is in $O(|V|^2)$. Given that the approximate topological sort uses a DFA visit of the resulting graph and that the layering is linear over the size of the vertices, and given that $|V| = |gg|$ by construction we, therefore, obtain an overall global quadratic time complexity over the size of the query when expressed via the number of available rules. \square

Proof (for Lemma 12). Suppose that each rule has at most c containment relationships, which will be provided by disjunction reference to all the k containment labels recorded in the physical model. Thus, the caching phase will take at most $ck|gg| + k$ time, as we might still consider all of these labels if we have containments for which the containment relationship is not specified.

Thus, the caching mechanism will guarantee sole access to each PhiTable^k once. By estimating an average branching factor β across the loaded GSM in the physical model and by assuming that, in the worst case scenario, all the objects will contain containments for all the k labels, then the cost of caching the tables to make them ready to the morphism representation takes $k|db|\bar{O}\beta$ time, where \bar{O} is the average number of objects stored across GSM databases.

Now, we consider the cost associated with a table of size $|db|\bar{O}\beta$. We can freely assume that rewriting operations are in $O(1)$, as in our implementation, morphisms are striped by schema information and are merely represented as tuples while associating the schema to the sole table. Similarly, the projection costs are linear over the size of the table, while the nesting operation can be performed in linear time while reducing the size of the table to $|db|\bar{O}$ due to the ego-net assumptions enforced within the structure of the matching pattern. Overall, this comes at $O(|db|\bar{O}\beta)$ time.

In the worst-case scenario, the association of containment to a table will take cost $k|db|\bar{O}\beta$, thus totalling an overall cost in $O(ck|db|\bar{O}\beta)$, which also dominates the time complexity of the other phases. \square

Proof (for Corollary 2). This proof refers to the time complexity of Algorithm 7, which can be seen as a corollary of the previous lemma, for which we already derived the estimated table size $m = |db|\bar{O}\beta$ for each required table composing the final morphism. Let us denote r (and o) as the maximum number of required (and optional) matches appearing across all L_i from gg -s. As the nested relational algebra can be always expressed in terms of “flat” relational algebra, we can upper-bound the time complexity of Algorithm 3 by $O(m^2)$. This gives us a worst-case scenario time complexity of $O(m^{r+o})$ for computing P_i for each

rule $p_i \in gg$, which is a linear time complexity over the size of the worst-case scenario yielded table.

The nesting operator $\nu_{B \rightarrow A}(t)$ from Equation (5) being optionally used to nest morphisms if entry-point vertices are grouped by a direct ancestor can be easily implemented with a linear time complexity over the size of the table that we want to nest: this boils down to computing the equivalence class of $\dot{=}^R$ over the fields $R = \text{dom}(\mathcal{S}(t)) \setminus \{A, B\}$ and holding such information as a map from the values $\tilde{t}(R)$ for each $\tilde{t} \in t$ to the collection of rows $\tilde{t}|_B$ for which $\tilde{t}|_R = \tilde{t}|_R$. Thus, Line 16 comes with a linear cost over the size of the table P_i .

Given that the time complexity of computing the symmetric closure of a relationship is trivially linear while the time complexity of computing the transitive closure for a relationship is upper-bounded by the Floyd–Warshall algorithm with a cubic time, this leads to a worst-case time complexity of $O(|db|\overline{O}^3)$ time for computing each \mathfrak{R}_i (Lines 17 and 18).

We can freely assume that the insertion cost of each morphism within the $MT[\cdot, \cdot]$ table comes at a linear cost, while the sorting of each $MT[L_i, g_j]$ comes with a cost of $O((r+o)m^{r+o} \log(m))$ with $m = |db|\overline{O}\beta$. This phase clearly dominates over all the previous ones, and thus we can freely assume that the time complexity of computing each morphism is in $O((r+o)m^{r+o} \log(m))$. This leads to an overall time complexity of $O(|gg|(r+o)m^{r+o} \log(m))$ for generating all the morphisms, which can be still upper-bounded by a polynomial time complexity. \square

Proof (for Lemma 13). In Section 4.3, we observed that all the functions that were there introduced can be computed in $O(1)$ time via the GSM view update; these operations also occur with the definition of \rightarrow_ν at the basis of the graph rewriting operations outlined in Section 6.5 and Figure 8: the worst case scenario for the evaluation of such expressions refers to the evaluation of variables associated with nested relationships, thus referring to at most β object per morphism Γ . Given that each rewriting operation considers one single morphism at a time and that, within the worst-case scenario, we consider the cross-product of the objects associated with both variables, the computation of each operation shall take at most $O(\beta^2)$ time.

Given $|gg|m^{r+o}$ the number of possible nested morphisms as determined from the previous corollary, this overall leads to an overall time complexity of $O(|gg|m^{r+o}\beta^2)$ for overall computing Algorithm 8. \square

Proof (for Corollary 3). This is a corollary for all the previous lemmas, as the composition of polynomial-time algorithms leads to an overall algorithm of polynomial-time complexity. \square

References

- Schmitt, I. QQL: A DB&IR Query Language. *VLDB J.* **2008**, *17*, 39–56. [\[CrossRef\]](#)
- Chamberlin, D.D.; Boyce, R.F. SEQUEL: A Structured English Query Language. In Proceedings of the 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, MI, USA, 1–3 May 1974; Altshuler, G., Rustin, R., Plagman, B.D., Eds.; ACM: New York, NY, USA, 1974; Volume 2, pp. 249–264. [\[CrossRef\]](#)
- Rodriguez, M.A. The Gremlin graph traversal machine and language (invited talk). In Proceedings of the 15th Symposium on Database Programming Languages, New York, NY, USA, 27 October 2015; pp. 1–10. [\[CrossRef\]](#)
- Robinson, I.; Webber, J.; Eifrem, E. *Graph Databases*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2013.
- Angles, R.; Gutierrez, C. *The Expressive Power of SPARQL*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 114–129.
- Bergami, G.; Petermann, A.; Montesi, D. THoSP: An algorithm for nesting property graphs. In Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), New York, NY, USA, 10–15 June 2018; GRADES-NDA '18. [\[CrossRef\]](#)
- Bergami, G. On Efficiently Equi-Joining Graphs. In Proceedings of the 25th International Database Engineering & Applications Symposium, New York, NY, USA, 14–16 July 2021; IDEAS '21, pp. 222–231. [\[CrossRef\]](#)
- Ehrig, H.; Habel, A.; Kreowski, H.J. Introduction to graph grammars with applications to semantic networks. *Comput. Math. Appl.* **1992**, *23*, 557–572. [\[CrossRef\]](#)
- Bergami, G. A New Nested Graph Model for Data Integration. Ph.D. Thesis, University of Bologna, Bologna, Italy, 2018.

10. Das, S.; Srinivasan, J.; Perry, M.; Chong, E.I.; Banerjee, J. A Tale of Two Graphs: Property Graphs as RDF in Oracle. In Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, 24–28 March 2014; pp. 762–773. [\[CrossRef\]](#)
11. Bergami, G.; Appleby, S.; Morgan, G. Quickening Data-Aware Conformance Checking through Temporal Algebras. *Information* **2023**, *14*, 173. [\[CrossRef\]](#)
12. Turi, D.; Plotkin, G.D. Towards a Mathematical Operational Semantics. In Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, 29 June–2 July 1997; IEEE Computer Society, pp. 280–291. [\[CrossRef\]](#)
13. Codd, E.F. A relational model of data for large shared data banks. *Commun. ACM* **1970**, *13*, 377–387. [\[CrossRef\]](#)
14. Kahn, A.B. Topological sorting of large networks. *Commun. ACM* **1962**, *5*, 558–562. [\[CrossRef\]](#)
15. Sugiyama, K.; Tagawa, S.; Toda, M. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Trans. Syst. Man Cybern.* **1981**, *11*, 109–125. [\[CrossRef\]](#)
16. Hölsch, J.; Grossniklaus, M. An Algebra and Equivalences to Transform Graph Patterns in Neo4j. In Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016, Bordeaux, France, 15 March 2016; Volume 1558.
17. Gutierrez, C.; Hurtado, C.A.; Mendelzon, A.O.; Pérez, J. Foundations of Semantic Web databases. *J. Comput. Syst. Sci.* **2011**, *77*, 520–541. [\[CrossRef\]](#)
18. Fionda, V.; Pirrò, G.; Gutierrez, C. NautiLOD: A Formal Language for the Web of Data Graph. *ACM Trans. Web TWEB* **2015**, *9*, 1–43. [\[CrossRef\]](#)
19. Hartig, O.; Pérez, J., Chapter LDQL: A Query Language for the Web of Linked Data. In Proceedings of the Semantic Web-ISWC 2015 14th International Semantic Web Conference, Bethlehem, PA, USA, 11–15 October 2015; Proceedings, Part I; Springer International Publishing: Cham, Switzerland, 2015; pp. 73–91. [\[CrossRef\]](#)
20. Carroll, J.J.; Dickinson, I.; Dollin, C.; Reynolds, D.; Seaborne, A.; Wilkinson, K. Jena: Implementing the Semantic Web Recommendations. In Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters, New York, NY, USA, 17–20 May 2004; WWW Alt. '04, pp. 74–83. [\[CrossRef\]](#)
21. Sirin, E.; Parsia, B.; Grau, B.C.; Kalyanpur, A.; Katz, Y. Pellet: A practical OWL-DL reasoner. *Web Semant. Sci. Serv. Agents World Wide Web* **2007**, *5*, 51–53. [\[CrossRef\]](#)
22. Angles, R.; Arenas, M.; Barceló, P.; Hogan, A.; Reutter, J.L.; Vrgoc, D. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* **2017**, *50*, 1–40. [\[CrossRef\]](#)
23. Fan, W.; Li, J.; Ma, S.; Tang, N.; Wu, Y. Adding regular expressions to graph reachability and pattern queries. *Front. Comput. Sci.* **2012**, *6*, 313–338. [\[CrossRef\]](#)
24. Barceló, P.; Fontaine, G.; Lin, A.W., Expressive Path Queries on Graphs with Data. In *Logic for Programming, Artificial Intelligence, and Reasoning, Proceedings of the 19th International Conference, LPAR-19, Stellenbosch, South Africa, 14–19 December 2013*; McMillan, K., Middeldorp, A., Voronkov, A., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; pp. 71–85. [\[CrossRef\]](#)
25. Junghanns, M.; Kießling, M.; Averbuch, A.; Petermann, A.; Rahm, E. Cypher-based Graph Pattern Matching in Gradoop. In Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, 14–19 May 2017; pp. 1–8. [\[CrossRef\]](#)
26. Ghrab, A.; Romero, O.; Skhiri, S.; Vaisman, A.A.; Zimányi, E. GRAD: On Graph Database Modeling. *arXiv* **2016**, arXiv:1602.00503.
27. Ghrab, A.; Romero, O.; Skhiri, S.; Vaisman, A.; Zimányi, E. Advances in Databases and Information Systems. In Proceedings of the 19th East European Conference, ADBIS 2015, Poitiers, France, 8–11 September 2015; Chapter A Framework for Building OLAP Cubes on Graphs; Springer International Publishing: Cham, Switzerland, 2015; pp. 92–105. [\[CrossRef\]](#)
28. Junghanns, M.; Petermann, A.; Teichmann, N.; Gomez, K.; Rahm, E. Analyzing Extended Property Graphs with Apache Flink. In Proceedings of the SIGMOD Workshop on Network Data Analytics (NDA), San Francisco, CA, USA, 1 July 2016 .
29. Wolter, U.; Truong, T.T. Graph Algebras and Derived Graph Operations. *Logics* **2023**, *1*, 10. [\[CrossRef\]](#)
30. Rozenberg, G. (Ed.) *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*; WSP: Anchorage, AK, USA, 1997.
31. Pérez, J.; Arenas, M.; Gutierrez, C. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst. (TODS)* **2009**, *34*, 16:1–16:45. [\[CrossRef\]](#)
32. Szárnyas, G. Incremental View Maintenance for Property Graph Queries. In Proceedings of the 2018 International Conference on Management of Data, SIGMOD, Houston, TX, USA, 10–15 June 2018; ACM: New York, NY, USA, 2018; pp. 1843–1845.
33. Consens, M.P.; Mendelzon, A.O. GraphLog: A Visual Formalism for Real Life Recursion. In Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS, Nashville, TN, USA, 2–4 April 1990; ACM: New York, NY, USA, 1990; pp. 404–416.
34. Bergami, G.; Zegadło, W. Towards a Generalised Semistructured Data Model and Query Language. *SIGWEB Newsl.* **2023**, *2023*, 1–22. [\[CrossRef\]](#)
35. Shmedding, F. Incremental SPARQL Evaluation for Query Answering on Linked Data. In Proceedings of the Second International Workshop on Consuming Linked Data, Bonn, Germany, 23 October 2011; COLD2011.
36. Huang, J.; Abadi, D.J.; Ren, K. Scalable SPARQL Querying of Large RDF Graphs. *Proc. VLDB Endow. PVLDB* **2011**, *4*, 1123–1134. [\[CrossRef\]](#)
37. Atre, M. Left Bit Right: For SPARQL Join Queries with OPTIONAL Patterns (Left-outer-joins). In Proceedings of the SIGMOD Conference, Melbourne, Australia, 31 May–4 June 2015; ACM: New York, NY, USA, 2015; pp. 1793–1808.

38. Colby, L.S. A recursive algebra and query optimization for nested relations. In Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 31 May–2 June 1989; SIGMOD '89, pp. 273–283. [[CrossRef](#)]
39. Liu, H.C.; Ramamohanarao, K. Algebraic equivalences among nested relational expressions. In Proceedings of the Third International Conference on Information and Knowledge Management, Gaithersburg, MD, USA, 29 November–1 December 1994; CIKM '94, pp. 234–243. [[CrossRef](#)]
40. Leser, U.; Naumann, F. *Informationsintegration: Architekturen und Methoden zur Integration Verteilter und Heterogener Datenquellen*; dpunkt.verlag: Heidelberg, Germany, 2006.
41. Elmasri, R.A.; Navathe, S.B. *Fundamentals of Database Systems*, 7th ed.; Pearson: New York, NY, USA, 2016.
42. Atzeni, P.; Ceri, S.; Paraboschi, S.; Torlone, R. *Database Systems—Concepts, Languages and Architectures*; McGraw-Hill Book Company: New York, NY, USA, 1999.
43. den Bussche, J.V. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theor. Comput. Sci.* **2001**, *254*, 363–377. [[CrossRef](#)]
44. Green, T.J.; Karvounarakis, G.; Tannen, V. Provenance semirings. In Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Beijing, China, 11–13 June 2007; PODS '07, pp. 31–40. [[CrossRef](#)]
45. Chapman, A.; Missier, P.; Simonelli, G.; Torlone, R. Capturing and querying fine-grained provenance of preprocessing pipelines in data science. *Proc. VLDB Endow.* **2020**, *14*, 507–520. [[CrossRef](#)]
46. Junghanns, M.; Petermann, A.; Rahm, E. Distributed Grouping of Property Graphs with GRADOOP. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*; Gesellschaft für Informatik: Bonn, Germany, 2017; pp. 103–122.
47. Bellatreche, L.; Kechar, M.; Bahloul, S.N. Bringing Common Subexpression Problem from the Dark to Light: Towards Large-Scale Workload Optimizations. In Proceedings of the 25th International Database Engineering & Applications Symposium, IDEAS, Montreal, QC, Canada, 14–16 July 2021; ACM: New York, NY, USA, 2021.
48. Aho, A.V.; Lam, M.S.; Sethi, R.; Ullman, J.D. *Compilers: Principles, Techniques, and Tools*, 2nd ed.; Addison-Wesley Longman Publishing Co., Inc.: San Francisco, CA, USA, 2006.
49. Ulrich, H.; Kern, J.; Tas, D.; Kock-Schoppenhauer, A.; Ückert, F.; Ingenerf, J.; Lablans, M. QL4MDR: A GraphQL query language for ISO 11179-based metadata repositories. *BMC Med. Inform. Decis. Mak.* **2019**, *19*, 1–7. [[CrossRef](#)]
50. Zhang, T.; Subburathinam, A.; Shi, G.; Huang, L.; Lu, D.; Pan, X.; Li, M.; Zhang, B.; Wang, Q.; Whitehead, S.; et al. GAIA—A Multi-media Multi-lingual Knowledge Extraction and Hypothesis Generation System. In Proceedings of the 2018 Text Analysis Conference, TAC 2018, Gaithersburg, MD, USA, 13–14 November 2018.
51. Parr, T. *The Definitive ANTLR 4 Reference*, 2nd ed.; Pragmatic Bookshelf: Raleigh, NC, USA, 2013.
52. Tarjan, R.E. Edge-Disjoint Spanning Trees and Depth-First Search. *Acta Inform.* **1976**, *6*, 171–185. [[CrossRef](#)]
53. de Marneffe, M.C.; Manning, C.D.; Nivre, J.; Zeman, D. Universal Dependencies. *Comput. Linguist.* **2021**, *47*, 255–308. [[CrossRef](#)]
54. Martelli, A.; Montanari, U. *Unification in Linear Time and Space: A Structured Presentation*; Technical Report Vol. IEI-B76-16; Consiglio Nazionale delle Ricerche, Pisa: Pisa, Italy, 1976.
55. Rozenberg, G. (Ed.) *Handbook of Graph Grammars and Computing by Graph Transformation: Volume II*; WSP: Anchorage, AK, USA, 1999.
56. Talmor, A.; Herzig, J.; Lourie, N.; Berant, J. CommonsenseQA: A Question Answering Challenge Targeting Commonsense Knowledge. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, 2–7 June 2019; Volume 1 (Long and Short Papers); Burstein, J., Doran, C., Solorio, T., Eds.; Association for Computational Linguistics: Stroudsburg, PA, USA, 2019; pp. 4149–4158. [[CrossRef](#)]
57. de Marneffe, M.; MacCartney, B.; Manning, C.D. Generating Typed Dependency Parses from Phrase Structure Parses. In Proceedings of the Fifth International Conference on Language Resources and Evaluation, LREC 2006, Genoa, Italy, 22–28 May 2006; Calzolari, N., Choukri, K., Gangemi, A., Maegaard, B., Mariani, J., Odijk, J., Tapias, D., Eds.; European Language Resources Association (ELRA): Paris, France, 2006, pp. 449–454.
58. Chen, D.; Manning, C.D. A Fast and Accurate Dependency Parser using Neural Networks. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, Doha, Qatar, 25–29 October 2014; A Meeting of SIGDAT, a Special Interest Group of the ACL; Moschitti, A., Pang, B., Daelemans, W., Eds.; ACL: Doha, Qatar, 2014; pp. 740–750. [[CrossRef](#)]
59. Kotiranta, P.; Junkkari, M.; Nummenmaa, J. Performance of Graph and Relational Databases in Complex Queries. *Appl. Sci.* **2022**, *12*, 6490. [[CrossRef](#)]
60. Györödi, C.; Györödi, R.; Pecherle, G.; Olah, A. A comparative study: MongoDB vs. MySQL. In Proceedings of the 2015 13th International Conference on Engineering of Modern Electric Systems (EMES), Oradea, Romania, 11–12 June 2015; pp. 1–6. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.