On Nesting Graphs

Giacomo Bergami
University of Bologna
CSE Department
Bologna, Italy
giacomo.bergami2@unibo.it

André Petermann
University of Leipzig &
ScaDS Dresden/Leipzig
Leipzig, Germany
petermann@informatik.uni-leipzig.de

Erhard Rahm
University of Leipzig &
ScaDS Dresden/Leipzig
Leipzig, Germany
rahm@informatik.uni-leipzig.de

Danilo Montesi
University of Bologna
CSE Department
Bologna, Italy
danilo.montesi@unibo.it

ABSTRACT

[TODO]

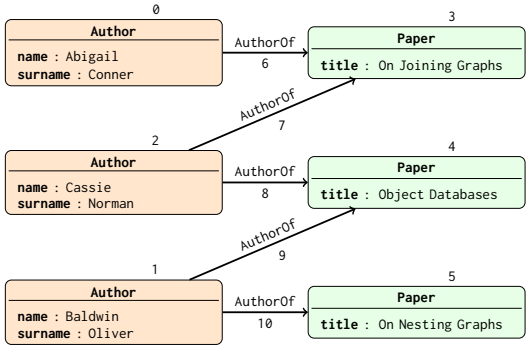
1. INTRODUCTION

Graph data management systems play an increasing role in present data analytics, because they allow flexible analyses of relationships among data objects. In fact, graphs have been already used as a fundamental data structure for representing data within different contexts, such as corporate data [21, 18], social networks [25, 6], statistical correlations [10] and linked data [24, 2]. Despite the increasing production of graph data, a general operator aggregating with a roll-up fashion one single graph is missing. Given one single input graph i, this operator should both preserve the vertices and edges that are not involved in the roll-up operation, and create **nested vertices** and **nested edges**, which may contain subgraphs of i; we would also like that the nested created components may be freely unnested, such that the original graph may be obtained back again. The edges connecting the nested vertices do not necessarily represent a partition of the input graph, but the presence of an edge between two summarized vertices represents the existence (within the input graph) of an edge (or more generally a path) between the partitions which the aforementioned vertices represent. Therefore, such desired outcome extends the *vertex summarization* class of aggregations: those latter strategies prefer to group the vertices as in relational `Group By` strategies, and then aggregate the edges accordingly [15]. These class of operations restrict the ability of aggregating by paths, that are limited to the set of edges connecting the vertices belonging to two (different) class of vertices. It follows that such class of aggregations do not allow to freely aggregate the edges into final nested edges.

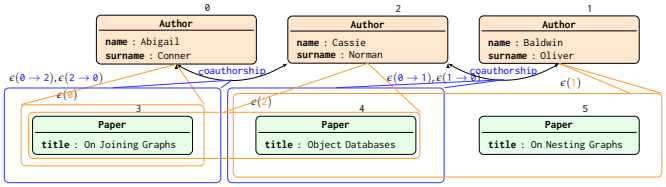
Since most of such vertex summarization techniques are graph-partition based, most of them fail to aggregate graphs that share some vertices and edges in common [26, 23, 14]. Among all those graph summarization techniques, only HEIDS [8] and Graph Cubes [27] perform graph summarizations of one single graph over a graph collection of non pairwise disjoint subgraphs. On the other hand, these last two techniques require that the set union of such graph collections must be equivalent to the original graph, while data integration and social networks scenarios even consider the possibility of outliers within the data, that are hereby represented in no subgraph belonging to the aforementioned graph collections.

The possibility of performing graph traversals allows *path summarizations* as well as vertex summarizations; path summarizations allow the aggregation of multiple paths among source and target vertices that share the same properties. The problem with both path and vertex summarizations is that no general class of either source and target vertices can be used as an outcome of a previous community detection [25, 4] or data cleaning and alignment phase [22] without rewriting the previously extracted data into an explicit query, thus requiring an additional pre-processing step and thus making such approach not as flexible as required by data integration scenarios. This problem is also reflected by pattern matching query languages (described in more detail in Section 2.2), where the vertex summarization phase must be necessarily separated from the path summarization one, thus thwarting the advantages of performing vertex and path summarizations contemporarily. In particular, while Cypher queries can perform distinct aggregations only within distinct `MATCH` clauses, SPARQL 1.1 implementations of the query plan combine these two phases with a `UNION` operator, thus forcing the algorithm to visit the same graph twice. As a result, the query plan optimizers of such query languages do not allow to avoid to visit one same graph more than once whether possible.

This paper shows that such limitations can be reduced by using a graph nesting operator, which embodies both vertex and path summarization queries, which are performed contemporarily within the same operator. We also investigate a data structure to optimize a specific graph nesting use case. To that end, we study the existence of specific graph patterns prone to algorithmic optimizations. Hereby, we show that such algorithmic approaches allow to reduce the time complexity of the visiting and nesting problem, which is going to performed over a data structure that requires less indexing time that its own competitors. In particular, we focus on intersecting graph visiting patterns: one pattern (g_V) is given for vertex summarization, and the other (g_E) is for path summarization into one single edge, thus subsuming which is the shared sub-pattern (α) among the two former ones. Consequently, α is going to be visited



(a) Input bibliographical network.



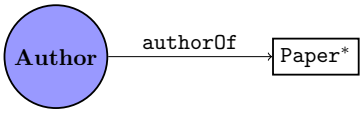
(b) Nested result: given two `Authors` a and a', there exist two `coauthorship` edges, $a \to a'$ and $a' \to a$ if and only if they share some authored paper contained respectively in $\epsilon(a \to a')$ and $\epsilon(a' \to a)$. Moreover, each author a is associated to the set of his authored papers $\epsilon(a)$.

Figure 1: Nesting a bibliographic network: the provenance information is nested within the original node.

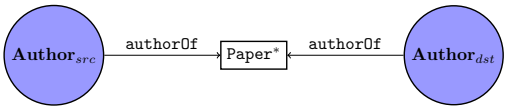
only once within the graph, after which either the vertex or the path summarization pattern can be visited in their entirety. We also perform some restrictions over these patterns enhancing such optimizations: for each vertex pattern we're going to elect one vertex called **vertex grouping reference** (γ_V), over which we're going to nest the elements matched by the graph pattern met during the visit. Similarly, each edge path summarization pattern is going to elect a source (γ_E^{src}) and a target (γ_E^{dst}) vertex, which are going to be called **edge grouping references**: such vertces must both coincide with the vertex grouping references, so that the newly generated edge will have as sources and target the previously vertex-nested elements.

Example 1. *Given a bibliographic network containing (at least)* Authors *and* Papers *as vertices, and where* authorOf *edges connect each author to the papers he has authored (Figure 1a), we want to "nest" the network into a coauthorship network, where each* Author *is connected by a* coAuthor *edge with another* Author *with which he has published some papers (Figure 1b). In particular, for each resulting* Author *vertex, nest inside it all of its papers as vertices, and nest inside each* coAuthor *edge all the papers coauthored by both the source and destination. We also want to exclude* coAuthor *hooks over the same vertex.*

This problem can be solved by visiting the graph starting from the vertices: if the current vertex is a Paper*, traverse backwards all the* authorOf *edges, thus reaching all of its* Authors*, that are going to be* coAuthor *for at least the current paper. Therefore, I can incrementally define the edge nesting between all such authors in a separated nesting index, since I know that all these authors share the same paper, and partially instantiate the nesting of each author with the papers that has authored via an external index. All the other vertices and edges may be discarded as a starting point for the graph visiting process. By doing this, only the edges are visited twice, but the vertices are visited only once. Hereby, with these patterns we reach the optimal solution by visiting the graph only once.*



(a) Vertex summarization pattern (g_V). Author is the vertex grouping reference γ_V.



(b) Path summarization pattern (g_E). Author_{src} and Author_{dst} are respectively edge grouping references γ_E^{src} and γ_E^{dst}.

Figure 2: Vertex and Path summarization patterns for the query expressed in Example 1. Vertex and edge grouping references are marked by a light blue circled node. As we can see, the vertex grouping reference depicts the same property expressed by edge grouping references.

Figure 2 represents the desired vertex and path summarization patterns. As we can see, the same vertex summarization pattern appears twice within the path summarization: this means that by visiting the path summarization pattern once, we may pre-istantiate two morphisms for the vertex summarization pattern. The two patterns have different key roles: while the vertex summarization retrieves all the papers that one author has published and nest them within one single matched author, the path summarizations return all the papers authored by two different authors and creates an edge between the two previously nested vertices. This construction implies that a join between the two paths must be carried out.

This pattern comparison remarks that, in order to reduce the graph time visit, we must start from visiting the `Paper`*, which is shared among the two distinct patterns, and then keep going with the graph visit by exploring the source and target vertices.*

Last, we implemented the aforementioned solution into a nested graph data model, which directly extends the graph data model used for graph joins in [3]. This chapter provides the following main contributions:

- **Graph Nesting Operator** (Section 3.1); we provide a general definition of the nesting operator which combines the path with the vertex summarization approaches to nesting graphs.
- Two Hop Separated Patterns **(THoSP) algorithm for a specific graph nesting task** (Section 4.1): we compare it to its implementation over both graph (SPARQL, Cypher), relational (SQL) and document oriented (AQL) query languages. The results

of such experiments shows that the sum of both indexing and query evaluation time of our proposed solution outperforms by at least one order of magnitude the aforementioned solutions evaluated on such databases with their respective query languages (Section 5). Consequently, our data model also proved to be curcial in providing an enhanced implementation of the specific graph nesting task.

- A general strategy on how to extend the THoSP algorithm for patterns having vertex and edge grouping references is provided (Section 4).

The source code is provided at `https://bitbucket.org/unibogb/graphnestingc`.

2. RELATED WORKS

2.1 Nested Graph Representations

Statecharts [12] were one of the first models representing nested graphs: they were used for representing complex systems at different abstraction levels, where each node represents a state or "configuration" of the system, and each edge represents a transaction between two different states on a given event. Each vertex and edge is labelled, but they do not come with attribute-value associations because this model was not designed for data representations. In order to represent different nesting levels, each node can contain other states and edges connecting such states. As a consequence, there is no distinction between (simple) states and states containing other states. This model allows both **external edges** and **internal edges**: we say that edge e is *external* if its source (or target) is contained by the target (or source) but neither of them contains e; the edge is called *internal* when the containing vertex (either its source or target) also contains the edge. Besides of state representation purposes, this model was been even used for both modelling the evolution of *pathophysiological* states and to describe the subsequent treatments to which the patient must undergo, where each treatment could be furtherly subdivided in smaller consequential steps to be followed [16].

This model was also adopted as a basis for the **hypernode** data model [19]: even if hypernodes are subsequent to statecharts, they are less expressive than the former ones, because they do not label the edges and they only allow edges between vertices which are contained within the same vertex: the model neither represents external edges nor internal ones. As previously stated for statecharts, even this model does not allow to fully represent a property graph, since the attribute-value association must be necessarily expressed as a relation between two different vertices [19]. Last, the fact that the vertex containments cannot overlap make such nested model affected from the same *data replication* representation problem described for semistructured and nested data. A first extension of the hypernode model towards data representation is represented by CoGITaNT [11], where any type of edge (thus including internal and external ones) are included and data is firstly contained inside a node. Nested graphs are also supported by GraphML [5] and GXL [13].

Two different approaches have been used to extend current graph data model for supporting nesting operations: the first ones try to overcome to the basic graph data structure limitations by simply extending the query language, while the other ones try to extend the data structures that are used for both input and intermediate computations.

Among the first type of approaches, the one outlined in [9] propose to define a RDF vocabulary over which the OLAP cube can be defined in RDF. On top of this "structured" RDF graph, an algorithm generates the SPARQL query that will allow to perform either the roll-up or the drill-down operation. This later approach implies that each possible computation has to be always recomputed on top of the row data like for classical ROLAP systems: as a consequence, this MOLAP approach does not benefit from the specific RDF representation, that is not able to represent different aggregation levels and to store intermediate computations.

The last type of approaches have been recently widely investigated, and seem to be more promising with respect to optimization techniques: in [23, 7, 20] graph data structures are associated with external graph indices, thus allowing to connect one graph to its broader one with respect to the roll-up query. As a consequence, these solutions do not allow to freely expand any aggregated component at a time, but they can only backtrack the aggregation to a previous known state.

2.2 Databases and Query Languages

We want now to discuss how current query languages can express graph nesting within their data model of choice. In particular, we must select query languages that either support collections or nested representations allowing to express the same query presented in our running example.

For these reasons, we select PostgreSQL's dialect which, by extending the SQL-3 syntax with JSON data supports, allows to create arrays as a result of a `GROUP BY` query via `array_agg`; therefore, instead of using function aggregating a collection into one single summarized value, we can list all the elements that have been aggregated. In particular, in PostgreSQL's dialect, we implement our graph by storing the triples defining an edge as the following relation:

`Edge`(*edgeId*, *sourceId*, *edgeLabel*, *targetId*)

Hereby, by grouping the edges by *sourceId* and collecting all the target's ids we obtain a representation of nested vertices. Similarly, if we join the `Edge` relation with itself and group the join result by two distinct *sourceId* and return the list of all the PAPERs that they have in common, we can return the list of all the PAPERs that they have coauthored. As showed in Listing 1, the overall graph nesting cannot be created in one single SQL query, because by the SQL language definition we cannot distinctively group the same dataset in different ways, but we must visit the same data twice and perform two distinct aggregations.

All the other query languages are going to be affected by the same problem, SPARQL included (Listing 2): despite the fact that this query language may represent the graph nesting query as a single statement, even in this case the `UNION` clause implies a separate visit for the two graph patterns. In particular, the first pattern allows to traverse those graph patterns matching the coauthorship statement in Figure 2b so that they can be nested within the created COAUTHORSHIP edge, while the second part returns all the associations of the PAPER that have been authored by one single AUTHOR. In particular, the `OPTIONAL...FILTER(!bound(...))` syntax is adopted instead of `FILTER NOT EXISTS`, because

Listing 1 Graph Nesting in PostgreSQL. Two distinct tables are created for both vertices and edges. The nesting is represented by nesting the elements' id within an array (`array_agg`). Please note that such sybtax is not SQL-3 standard.

```
-- Nesting Vertices
SELECT distinct T.sourceId as src,
       array_agg(distinct T.dst) as papers
FROM edges-i as T
GROUP BY T.sourceId;

-- Nesting Edges
SELECT distinct T.sourceId as src,
       T1.sourceId as dst,
       array_agg(distinct T1.targetId) as
           ↪ papers
FROM edges-i as T, edges-i as T1
WHERE     T.targetId = T1.targetId
      AND T.sourceId <> T1.sourceId
GROUP BY T.sourceId, T1.sourceId;
```

Listing 2 Graph Nesting in SPARQL. Given that the RDF List solution is inefficient and that named graphs cannot be used either in the RDF models as vertices or edges, we use other properties to associate to either vertices and edges the nesting content.

```
CONSTRUCT {
  ?autha ?newedge ?authb.
  ?newedge <http://cnt.io/nesting> ?paper1.
  ?authc <http://my.grph/edge> ?paper2.
} WHERE {
  {
   GRAPH <http://my.grph/g/i/> {
     ?autha <http://my.grph/g/edge> ?paper1.
     ?authb <http://my.grph/g/edge> ?paper1.
   }
   FILTER(?autha != ?authb).
   BIND(URI(CONCAT("http://my.grph/g/newedge/
       ↪ ",STRAFTER(STR(?autha),"http://my.
       ↪ grph/g/id/"),"-",STRAFTER(STR(?authb
       ↪ ),"http://my.grph/g/id/"))) AS  ?
       ↪ newedge).
  } UNION {
   GRAPH <http://my.grph/g/i/> {
     ?authc <http://my.grph/g/edge> ?paper2.
   }
   OPTIONAL {
     ?authd <http://my.grph/g/edge> ?paper2.
     FILTER (?authd != ?authc)
   }
   FILTER(!bound(?authd))
  }
}
```

the latter is only supported in SPARQL1.1, which is not supported by the current version of *librdf* used to query Virtuoso in our benchmarks. In this case, the edge nesting is performed via the association of different `<http://cnt.io/nesting>` properties departing from the `?newedge` CoAuthorship between two coauthors. Consequently, even in this case the graph visits two times the same graph patterns.

We also consider the AQL query language supported by ArangoDB, because ArangoDB is a NoSQL database relying on a document-oriented storage, which is hereby prone to

Listing 3 Graph Nesting in ArangoDB using AQL as a query lanugage. Please note that all the fields marked with an underscore represent externally indexed structures and. Therefore, only external indices are used within the query plan.

```
-- Nesting vertices
FOR b IN authorOf
  COLLECT au = b._from INTO groups = [b._to]
  RETURN {"author" : au,  "papers": groups }

-- Nesting edges
FOR x IN authorOf
  FOR y IN authorOf
    FILTER x._to==y._to && x._from!=y._from
    COLLECT src = x._from, dst = y._from
      INTO groups = [ x._to ]
    RETURN {"src": src, "dst": dst,
            "contain": groups}
```

Listing 4 Graph Nesting in Neo4J using Cypher as a Query Language. Please note that, even in this case, is it not possible to return one single nested graph immediately, and hence the nested vertices must be created before creating the nested edges. This implies that a greater number of joins is required to associate the previously nested data to the original operand.

```
MATCH (a1: Author)-->(p1:Paper)
WITH a1, collect(p1.UID) AS papers1
CREATE  p=(:Authors {authored: papers1, id:a1
    ↪ .UID})

MATCH (a1: Author)-->(p:Paper)<--(a2: Author)
    ↪ , (a1p: Authors), (a2p: Authors)
WITH a1, a2, a1p, a2p, collect(p.UID) AS
    ↪ common
WHERE a1.UID = a1p.id AND a2.UID = a2p.id AND
    ↪  a1.UID <> a2.UID
CREATE p=(a1p)-[:Papers {coauthored: common
    ↪ }]->(a2p)
```

both represent and return nested content. An example of how such graph nesting query can be carried out in AQL is presented in Listing 3: in this scenario we assume that we've previously loaded our graph data with the default format, where both vertices and edges are fully stored, and where the former are indexed by id, while the latter are also indexed by source and target vertex id. In particular, we can state that this algorithm provides the exact same result as the one produced by the SQL query, except that JSON documents are returned instead of relational tables containing JSON arrays.

Last, Listing 4 provides an example of Neo4J allowing to nest property graphs: even in this case the property graph model does not directly nest the graphs inside one element. Similar to the previous approaches, we can group by all the graphs returned by the graph pattern by selecting the vertices of interest, and nesting the to-be-grouped remaining objects inside a collection. In particular, we can first match the vertex summarization pattern in Figure 2a and group by Author, and nest the collection of authored Papers within the to-be-created nested vertex; similarly, we can first match the path summarization pattern presented in Figure

2b by source and destination AUTHOR, and then create an edge between the previously created nested vertices by collecting all the coauthored PAPERs appearing in the original graph. Moreover, the Neo4J property graph data model implies that we cannot create an edge if the vertices are not previously known beforehand and, therefore, we always must join the nested vertices with the original matched ones in order to reconstruct the original information and perform the actual matching operation. As it will be observed within the benchmarks, the solution of not separating the elements' ids from their data quickly leads to an intractable solution.

3. DATA MODEL

The term *property graph* usually refers to a directed, labelled and attributed multigraph (each vertex and edge is represented as a relational tuple). In a property graph a single *label* is associated to every vertex and edge (e.g., `Author` or `coAuthorsip`). Further on, vertices and edges may have arbitrary named attributes (*properties*) in the form of key-value pairs (e.g., `fullname:"Alice McKenzie"` or `institution:"Bengodi Ltd."`). Properties are set as tuples from the relational model. We define the *nested (property) graphs* as follows:

DEFINITION 1 (NESTED GRAPH DATABASE). *Given a set Σ^* of strings, a **nested (property) graph database** G is a tuple $G = \langle \mathcal{V}, \mathcal{E}, \lambda, \ell, \omega, \nu, \epsilon \rangle$, where $\mathcal{V}$ and $\mathcal{E}$ are disjoint sets, respectively referring to vertex and edge identifiers $(c, i) \in \mathbb{N}^2$. In particular, input data graphs have $c = 0$ while data views containing new elements may have new vertices or edges with $c > 0$.*

Each vertex and edge is assigned to multiple possible labels through the labelling function $\ell : \mathcal{V} \cup \mathcal{E} \to \wp(\Sigma^*)$. λ is a function $\mathcal{E} \to \mathcal{V}^2$ mapping each edge to its source and target vertex. ω is a function mapping each vertex and edge into a relational tuple.

*In addition to the previous components defining a property graph, we also introduce functions representing vertex content $\nu\colon (\mathcal{V} \cup \mathcal{E}) \to \wp(\mathcal{V})$ and edge content $\epsilon\colon (\mathcal{V} \cup \mathcal{E}) \to \wp(\mathcal{E})$. These functions induce the nesting by associating a set of vertices or edges to each vertex and edge. Each vertex or edge $o \in V \cup E$ induces a **nested (property) graph** as the following pair:*

$$G_o = \langle \nu(o), \{\, e \in \epsilon(o) \mid s(e), t(e) \in \nu(o) \,\} \rangle$$

Since the content functions ν and ϵ induce the expansion of each single vertex or edge to a graph, we must avoid recursive nesting to support expanding operations. Therefore, we additionally introduce the following constraints to be set at a nested property graph database level:

AXIOM 1 (RECURSION CONSTRAINTS). *For each correctly nested property graph, each vertex $v \in \mathcal{V}$ must not contain v at any level of containment of ϵ and, any of its descendants c on the underneath level must not contain v itself. That is:*

$$\forall v \in \mathcal{V}. \forall c \in \nu^+(v).\ c \neq v \wedge v \notin \nu^+(c)$$

Similarly to vertices, any edge shall not contain itself at any nesting level:

$$\forall e \in \mathcal{E}. \forall c \in \rho^+(e). c \neq e \wedge e \notin \rho^+(c)$$

A vertex v having a non-empty vertex or edge content is called **nested vertex**, while vertices with empty content are simply referred to **simple vertices**. For edges, we respectively use the terms **nested edges** and **simple edges**.

3.1 Graph Nesting

The graph nesting operator uses a classifier to group all the vertices and edges that shall appear as a member of a cluster C.

DEFINITION 2 (NESTED GRAPH CLASSIFIER, g_κ). *Given a set of cluster labels $\mathcal{C}$, a **nested graph classifier** operator g_κ maps a nested property graph G_o into a nested property graph collection $\{G_C\}_{C \in \mathcal{C}, G_C \neq \emptyset}$ of subgraphs of G_o. Such operator uses a classifier function $\kappa\colon \mathcal{V} \cup \mathcal{E} \to \wp(\mathcal{C})$ mapping each vertex or edge in either no graph or more than one non-empty subgraph. Each nested graph G_C is a pair $G_C = \langle \mathcal{V}_C, \mathcal{E}_C \rangle$ where $\mathcal{V}_C$ (and $\mathcal{E}_C$) is the set of all the vertices v (and edges e) in G_o having $C \in \kappa(v)$ (and $C \in \kappa(e)$). Therefore, the nested graph classifier is defined as follows:*

$$g_\kappa(G_o) = \{\, \langle \mathcal{V}_C, \mathcal{E}_C \rangle \mid C \in \mathcal{C}, \mathcal{V}_C \neq \emptyset, \mathcal{E}_C \neq \emptyset \,\}$$

The former definition is also going to express graph pattern evaluations, where κ may be represented as a graph (cf. Neo4J). In order to represent the subgraphs in $g_\kappa(G_o)$ as either vertices and edges, we may use the following USER-DEFINED FUNCTIONS:

DEFINITION 3 (USER-DEFINED FUNCTIONS). *An **object user defined function** μ_Ω maps each subgraph $G_C \in g_\kappa(G_o)$ into a pair $\mu_\Omega(G_C) = (L, t)$, where $L \in \wp(\Sigma^*)$ is a set of labels and t is a relational tuple.*

*An **edge user defined function** μ_E maps each subgraph $G_C \in g_\kappa(G_o)$ into a pair of identifiers $\mu_E(G_C) = (s, t)$ where $s, t \in \mathbb{N}^2$.*

While μ_Ω may be used for transforming subgraphs to both vertices and edges, μ_E is only used to map subgraphs to edges. In order to complete such transformation, we have to map each graph in $g_\kappa(G)$ into a new id $(\mathbf{c}, \mathbf{i}) \notin \mathcal{V} \cup \mathcal{E}$, for which an indexing function has to be defined as follows:

$$\iota_G(G_C) = (\max\{\, c \mid (c, i) \in V \cup E \,\} + 1, d(V_C \cup V_E))$$

where d is an arbitrary bijection associating a n-tuple in $\mathbb{N}^n$ into one single number $\mathbb{N}$ [17]. The combination of all the previous functions allow the definition of the following nesting operator:

DEFINITION 4 (GRAPH NESTING). *Given a nested graph $G_{(c,i)}$ within a nested graph database G, an object user defined function μ_Ω, an edge user defined function μ_E and an indexing function ι_G, the graph nesting operator $\eta^{\textbf{keep}}_{g_V, g_E, \mu_\Omega, \mu_E, \iota_G}$ converts each subgraph in $G_C \in g_V(G_{(c,i)})$ (and $G_C \in g_E(G_{(c,i)})$) into a nested vertex (and nested edge) $\iota_G(G_C)$ and adds them in the resulting nested graph:*

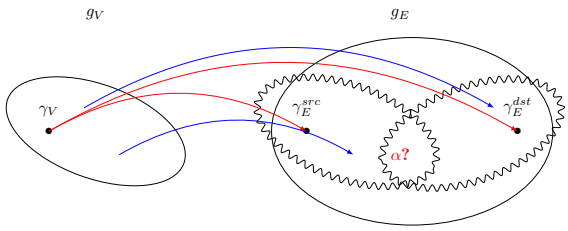
$$\begin{aligned}
&\eta^{\textbf{keep}}_{g_V, g_E, \mu_\Omega, \mu_E, \iota_G}(G_{(c,i)}) = \\
&\quad \langle \{v \in \nu(c,i) | V(v) = \emptyset \wedge \textbf{keep}\} \cup \iota_G(g_V(G_{(c,i)})), \\
&\quad\ \{e \in \epsilon(c,i) | E(e) = \emptyset \wedge \textbf{keep}\} \cup \iota_G(g_E(G_{(c,i)}))\rangle
\end{aligned}$$

*The vertices and edges in $G_{(c,i)}$ that appear neither in a nested vertex nor in a nested edge may be also returned if **keep** is set to `true`. As a result, the nested graph database is*

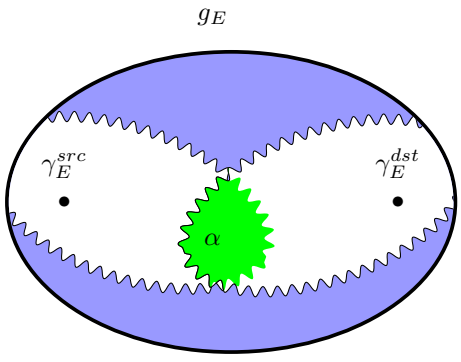
updated by using the nested graph classifier and user defined functions as follows:

$$\Big\langle \mathcal{V} \cup \iota_G(g_V(G_{(c,i)})),\ \mathcal{E} \cup \iota_G(g_E(G_{(c,i)})),$$
$$\lambda \oplus \bigoplus_{G_C \in g_E(G_{(c,i)})} \iota_G(G_C) \mapsto \mu_E(G_C),$$
$$\ell \oplus \bigoplus_{G_C \in g_E(G_{(c,i)}) \cup g_V(G_{(c,i)})} \iota_G(G_C) \mapsto \texttt{fst}\ \mu_\Omega(G_C),$$
$$\omega \oplus \bigoplus_{G_C \in g_E(G_{(c,i)}) \cup g_V(G_{(c,i)})} \iota_G(G_C) \mapsto \texttt{snd}\ \mu_\Omega(G_C),$$
$$\nu \oplus \bigoplus_{G_C \in g_E(G_{(c,i)}) \cup g_V(G_{(c,i)})} \iota_G(G_C) \mapsto V_C,$$
$$\epsilon \oplus \bigoplus_{G_C \in g_E(G_{(c,i)}) \cup g_V(G_{(c,i)})} \iota_G(G_C) \mapsto E_C \Big\rangle$$

where $f \oplus g$ defines the extension of the (finite) function f with another (finite) function g [3].



(a) Comparing the vertex summarization pattern and the path summarization patterns. We suppose that edge grouping references (γ_E^{src} and γ_E^{dst}) correspond to the same vertex appearing as a vertex grouping reference (γ_V) Such correspondence is directly marked my the user itself providing the query by drawing morphisms (correspondences) between the vertex and the path summarization patterns (red edges). The intersections between the two patterns may be directly outlined by the user itself that provides the query (blue edges, representing other morphisms).



(b) Path summarization pattern sharing an α area of common patterns shared between the patterns, which are necessarily not the edge grouping references by definition.

Figure 3: Vertex and Path summarization patterns.

4. CLASS OF OPTIMIZABLE GRAPH NESTING QUERIES

Prior to the analysis of the THoSP algorithm that is going to be provided later on in the next subsection, we want to discuss which is the class of vertex and path summarization patterns optimizable as discussed in the chapter's introduction. As we already observed, the exhaustive search of graph patterns in the most general scenario must be done at any rate, because the vertex and edge subgraphs may be extracted by an external tool of which we totally ignore its behaviour. As we also discussed in the introduction, the generation of the collections is only relevant with respect to actual data that is going to be nested and, in our case, we can only nest a subgraph of the graph resulting from the graph visiting process: consequently, within each pattern we must remark which elements are going to be nested at the final result.

First, we must provide a formal characterization of the grouping reference: we want to elect a subgraph γ_p for each graph pattern p such that each graph generated by $g_p(G_o)$ expose unique elements referring to γ_p. These grouping references allow to elect the subcomponents that identify an entity over which the aggregation will be performed during the graph matching process. Hereby, we can provide the following formal definition for grouping references:

Definition 5 (Grouping reference). *Given a graph pattern p generating subgraphs $g_p(G_o)$ over a nested graph G_o, a* ***grouping reference*** *γ_p is a subpattern $\gamma_p \subseteq g_p$ restricting the possible generated by $g_p(G_o)$ such that to each vertex (or edge) in γ_p corresponds one single vertex (or edge) in G_o and that $g_p(G_o)$ contains distinct graphs. Such correspondence may be denoted using a function f_C for each $G_C \in g_p(G_o)$ denoted as follows:*

$$m_{g_p}(G_o) = \left\{ f_C \;\middle|\; G : C \in g_p(G_o), p \xrightarrow{f_C} G_C \right\}$$

In particular, the elements appearing in SQL's `Group By`, AQL's `COLLECT` and Cypher's `WITH` (except from the parts where the aggregation is performed) are all grouping references of our matched graphs.

In order to reduce the computational complexity of aggregating the grouping reference, we can reduce the grouping references to one single vertex for vertex summarization patterns, and to two (distinct) vertices for path summarization patterns: in the first scenario, such vertex will identify the entity over which we can perform the nesting of all the other matched contents, while for the path summarization pattern the grouping references will identify the source (γ_E^{src}) and target (γ_E^{dst}) vertices corresponding to the vertex summarization patterns' grouping references, and hereby corresponding to the final vertices that are going to be nested in the final result. Hereby, the class of our algorithms create new nested edges only over vertices that have been previously matched as grouping references and then nested. Moreover, we can choose to mark with a specific ℓ label (e.g. "toNest") each vertex and edge within each pattern in order to remark which matched vertices and edges are going to be represented in the final nesting result; this implies that UDF functions are not required by such class of problems because they can be directly represented within the graph patterns.

Algorithm 1 Grouping Reference Optimizable Queries (GROQ)

1: **procedure** GROQ$((g_V, \gamma_V), (g_E, \gamma_E^{src}, \gamma_E^{dst}), m;\ G_o)$:
2: $\alpha := g_V \cap g_E \backslash (\gamma_V \cup \gamma_E)$;
3: lV := [];
4: **if** $\alpha \neq \emptyset$ **then**
5: **for each graph** g^i generated from $m_\alpha(G_o)$ **do**
6: lV := $\{f_i \in m_{g_V;\gamma_V}(G_o) | f_i(\alpha) = g^i\}$
7: GROQ$\alpha((g_V, \gamma_V), (g_E, \gamma_E^{src}, \gamma_E^{dst}), m;\ \text{lV}, G_o)$
8: **else**
9: lV := $m_{g_V;\gamma_V}(G_o)$
10: GROQ$\alpha((g_V, \gamma_V), (g_E, \gamma_E^{src}, \gamma_E^{dst}), m;\ \text{lV}, G_o)$
11:
12: **procedure** GROQ$\alpha((g_V, \gamma_V), (g_E, \gamma_E^{src}, \gamma_E^{dst}), m;\ \text{lV}, G_o)$
13: **for each morphism** $f_i \in$ lV **do**
14: $\{(c, i)\} := f_i(\gamma_V)$
15: $\omega(c+1, i) = \omega(c, i)$; $\ell(c+1, i) = \ell(c, i)$
16: $\nu(c+1, g) := \nu(c+1, g) \cup \{(c+1, i)\}$
17: $\nu(c+1, i) := \nu(c+1, i) \cup \{\ f_i(o) \mid o \in V \cap \mathcal{V}$, "toNest" $\in \ell(o)\ \}$
18: $\epsilon(c+1, i) := \epsilon(c+1, i) \cup \{\ f_i(o) \mid o \in V \cap \mathcal{E}$, "toNest" $\in \ell(o)\ \}$
19: **for each morphism** $f_i, f_j \in$ lV **do**
20: lE := $\{f_k \in m_{g_E;\gamma_E^{src},\gamma_E^{dst}}(G_o) | f_i(\gamma_V) = f_k(\gamma_E^{src}), f_j(\gamma_V) = f_k(\gamma_E^{dst})\}$
21: **for each morphism** $f_k \in$ lE **do**
22: $\{(c, s)\} := f_i(\gamma_E^{src})$; $\{(c, d)\} := f_i(\gamma_E^{dst})$
23: $j := dt(s, d)$
24: $\omega(c+1, j) := \omega(c, s) \cup \omega(c, d)$; $\xi(i_{c+1}) := \xi(i_c) \cup \ell(d_c)$
25: $\epsilon(c+1, j) := \epsilon(c+1, j) \cup f_k(\gamma_V)$
26: $\lambda(c+1, j) := ((c+1, s), (c+1, d))$
27: $\nu(c+1, j) := \nu(c+1, j) \cup \{\ f_k(o) \mid o \in E \cap \mathcal{V}$, "toNest" $\in \ell(o))\ \}$
28: $\epsilon(c+1, j) := \epsilon(c+1, j) \cup \{\ f_k(o) \mid o \in E \cap \mathcal{E}$, "toNest" $\in \ell(o)\ \}$

Figure 3 provides an example on how graph nesting queries based on grouping references can be optimized for both vertex (g_V) and path (g_E) summarization queries: given that the users are going to provide both the vertex and the path summarization queries, such users must directly draw the correspondences between vertex and edge pattern queries, so that the correspondences can be promptly identified by the query plan which can better optimize the whole query execution (Figure 3a). After doing so, we can start to perform the general graph visiting algorithm for graph nesting (Algorithm 1) by detecting which regions of both patterns are shared together in $\alpha = g_V \cap g_E$ (Figure 3b); given that source and destination vertices in path summarization patterns' grouping references are distinct by definition, source and destination vertices may not be represented in α (Algorithm 1, line 2). Consequently, in order to reduce the graph visiting process, we can first perform pattern matching over the input graph over α, thus allowing a partial instantiation of the g_V and g_E patterns, and then iteratively extend the nesting information after each visit of α and its own refinements. In particular, we can perform the algorithm as follows:

- Given a nested graph classifier g for graph pattern languages, we extract all the subgraphs g^i of G_o generated by $g_\alpha(G_o)$, when α is not empty (line 5). If α is otherwise an empty pattern, we must necessarily perform a complete visit of the vertex patterns g_V, and perform complete instantiations of such patterns (line 9).

- We can iteratively construct the nested graph without knowing the complete information by relying on the ids of the expected elements, and we can provide the greatest subgraph of g matching α after visiting each possible α matching result, represented as a correspondence f_i. For this reason, the GROQα subroutine may be called in both cases.

- After providing a partial instantiation of the vertex summarization patterns via α, we find a vertex (c, i) matching the grouping reference γ_V to which we are going to nest the remaining objects: from (c, i) we generate a newly derived vertex $(c+1, i)$ (line 14) preserving all the labels and tuples (line 15). The nesting content of $(c+1, i)$ derives from the partial instantiation of the correspondence f_i, by choosing the vertices and edges in G_o which corresponds to vertex summarization objects marked with "toNest" (lines 17 and 18).

- At this point we can use the same semi-instantiated correspondence in `lV` from α to partially instantiate the path summarization pattern, that is now going to be fully traversed (line 20). For each of these f_i instantiations, new edges are going to be generated, inheriting the labels and tuples from the matched edges grouping references, (c, s) and (c, d). In particular, we can directly create associate to such edge the soruces and the targets represented by nested vertices, which will respectively be $(c+1, s)$ and $(c+1, d)$.

- The procedure is iterated until the whole graph is not visited via subsequent correspondences, and hence all the matched elements are associated from the objects $(c+1, i)$ (either vertices or edges) generated from the ones matched by the grouping reference (c, i).

As we can see from the algorithm, the advantage of this approach is that the graph g^i and the instantiated correspondences (as a consequence of the graph matching phase) are promptly used to define the nested information (e.g., lines 16-18). It is evident that the aforementioned algorithm provides the best performances when γ_E^{src} and γ_E^{dst} are separated by one edge distance in α and both g_V and g_E create graph collections that are partitions of G_o. On the other hand, this class of algorithms was already discussed in literature and, consequently, an approach describing how to optimize such scenarios can be already found in literature [15]. Nevertheless, this chapter focuses on another types of algorithms, which are the ones where α contains two edges and one vertex; this class of problems, to the best of our knowledge, has not been discussed yet in current literature with respect to their optimizations.

4.1 Two HOp Separated Patterns Algorithm

We now want to focus on a specific instance of the problem stated in Algorithm 1: suppose to store a graph using adjacency lists `[similarly to the one proposed in the Graph Join algorithm]`; in particular, the previous data structure is now extended with both vertex and edge containment, plus with both ingoing and outgoing edges for each single graph vertex. The latter requirement is added in order to satisfy the possibility to visit the edges backwards, thus allowing to navigate the graph in each possible direction. The main data structure over which this algorithm relies is presented in Figure 4: it shows that minor changes have been applied to the original data structure that was used to serialize graph within the graph join scenario. Given

Algorithm 2 Two HOp Separated Patterns Algorithm (THoSP)

```
1: procedure PartitionHashJoin((g_V, γ_V), (g_E, γ_E^src, γ_E^dst); G_o)
2:   File AdjFile = Open(G_o);
3:   File Nesting = Open(new);
4:   Adjacency toSerialize =
       new Map<Vertex,<Edge,Vertex>>();
5:   α := g_V ∩ g_E \ (γ_V ∪ γ_E);
6:   for each vertex v in AdjFile do
7:     if α ⊨ v then
8:       for each γ_V ⊨ (u, e, v) do
9:         u' := dt(1, dt(0, u))
10:        NestingIndex.write(⟨u', u⟩)
11:        NestingIndex.write(⟨u', e⟩)
12:        NestingIndex.write(⟨u', v⟩)
13:        if γ_E ⊨ (v, e, u, e', w) then
14:          w' := dt(1, dt(0, w))
15:          ε := dt(1, dt(u, w))
16:          NestingIndex.write(⟨ε, u⟩)
17:          NestingIndex.write(⟨ε, e⟩)
18:          NestingIndex.write(⟨ε, w⟩)
19:          NestingIndex.write(⟨ε, e'⟩)
20:          NestingIndex.write(⟨ε, v⟩)
21:          toSerialize.put(u',⟨ε, w'⟩)
     AdjFile.serialize(toSerialize);
```

that the data structure requires a simple linear visit of the graph, no additional primary and secondary data structures are required. Nevertheless, during our serialization phase we provide both a primary index for accessing external informations (*VertexIndex*) and the serialization of all the vertices' adjacency lists, which is going to be used for traversing the graph (*VertexVals*). In our straightforward implementation, hash values are here used only as placeholders for the nodes' labels used within the patterns but, given that vertices are not sorted by hash value as for graph joins, we keep the hash fields for both backward compatibility and in order to make graph joins possible for nested graphs, too.

Let us now restrict α to one single vertex and two edges: for each vertex v matched by α ($\alpha \vDash v$) we know that we must (possibly) visit all the edges going from v towards the vertices γ_E^{src} and γ_E^{dst}, that substantially are γ_V. Please note that if in g_E there is no path connecting α to γ_E^{src} or γ_E^{dst}, the problem may quickly become cubic with respect to the size of the vertices, because we must create all the possible permutations where v is present alongside another element matching γ_E^{src} or γ_E^{dst}. Therefore, having an edge as a constraint in α linking v towards γ_E^{src} or γ_E^{dst} both in g_E and g_V can reduce all the possible computations to the actual edges traversed from v meeting the grouping references. Therefore, we know whether we finished visiting our patterns after exhaustively matching all the elements within the pattern. We can now reduce the cost to check when we finished traversing all the elements reaching γ_E^{src} and γ_E^{dst} from v after a linear scan of all the ingoing or outgoing nodes. Hereby, the most simple graph nesting example is where v is the middle node between a path between γ_E^{src} and γ_E^{dst} vertices.

Finally, Algorithm 2 provides the desired implementation of the THoSP algorithm: we can observe that THoSP does not include the data serialization preprocessing step because the data indexing provided at that step is not required by the present algorithm. The main memory is used to create the graph (represented as an adjacency list) that is going to be later on serialized using the same data structure used for providing the result for graph joins, that is an adjacency list where only the vertices' and edges' id appear. This choice is also done both for backward compatibility with respect to the graph join data structure [3] and for representing the nesting containment as a separate data structure. We can easily observe that this approach may slow down the whole algorithm, that can be quickened by directly storing the graph representation in secondary memory by using linear hashing. The nesting data structure is stored in a *NestingIndex* file as a set of pairs $\langle u, v\rangle$, where u represents the containing object and v represents the content. By doing so, we omit the `Group By` cost which affects the previously seen query languages, thus allowing to an overall better performance. Even in this case, a join is performed between the two nested patterns: this is evident from the two nested for loops appearing in the algorithm.

Table 1 provides a comparison between the general Nesting Algorithm `[Should we put the previous CIKM version on arXive, so that here we just focus on the main algorithm?]` and over the THoSP implementation of the query provided in our running example, under the assumptions that are going to be soon introduced in the next section. In particular, while THoSP increases linearly alongside the data size, the general nesting algorithm grows quadratically, thus quickly leading to a intractable time evaluation for big data scenarios. Hereby, the THoSP algorithm is going to be used in comparisons with other problem-specific queries on different query languages and data structures.

5. EXPERIMENTAL EVALUATIONS

Through the following experiments we want to prove that *(i)* both the time required to serialize our data structure and *(ii)* the query plan provided for the graph nesting query, outperform the same tasks performed on top on other databases, either graph, relational, or document oriented. For a first analysis we compare the loading time (the time required to store and index the data structure) and for the second one we time the milliseconds to perform the query over the serialized and indexed data structure. Moreover, we shall jointly compare the time required to query the data with the time spent on the loading phase, because in some cases better query performances may be lead to the creation of several indices.

The query plans are evaluated with respect to the data representation independently to the data attached to it: in this case there is no additional penalty over carrying the values associated to the indices, and only the algorithms used within the data structures are compared. We choose a graph where vertices are only represented by their vertex ids and their label, and edges are represented only by the source and target vertex, alongside with the edge's label. Consequently, within the relational model the whole information can be stored only within one single table regarding the edges. Similar approaches are used in Virtuoso for representing RDF triple stores over a relational engine. We performed our benchmark using a bibliography network using the gMark generator [1], as the previous dataset cannot be used for this other query task. To each `authorOf` relation a Zipf's Law distribution with parameter 2.5 is associated to the ingoing distribution, while a normal distribution between 0 and 80 is associated to the outgoing distribution. Consequently, each vertex represents either an `Author` or an authored `Paper`, and are represented by distinct vertex ids.

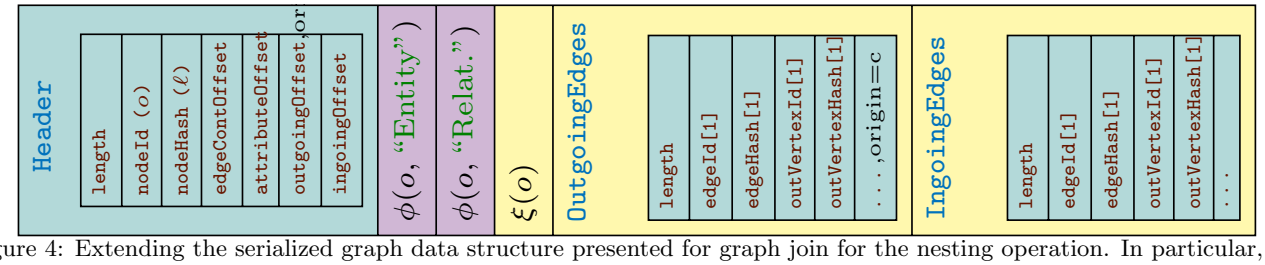


Figure 4: Extending the serialized graph data structure presented for graph join for the nesting operation. In particular, the present data structure extends each vertex representation in $VertexVals$ (Figure **??**) in order to fully supports the nested graph data model: entities and relationships may now be contained into another data node (either a vertex or an edge). The first block of the serialized data structure contains the pointers towards the memory regions containing data which may vary in size. The fuchsia nodes remark the memory spaces where such data containments may be stored. Moreover, ingoing edges are stored as well as outgoing edges.

Operands' Vertices	Matched Graphs	**General Nesting** (ms)	**THoSP** (ms)
10	3	0.57	0.11
10^2	58	0.73	0.14
10^3	968	2.78	0.46
10^4	8,683	152.11	4.07
10^5	88,885	14,015.00	43.81
10^6	902,020	1,579,190.00	563.02
10^7	8,991,417	>1H	8,202.93
10^8	89,146,891	>1H	91,834.20

Table 1: Comparing the performances of the THoSP algorithm with the naive General Nesting algorithm. This comparison shows that the previously defined algorithm has a worse performance than the THoSP one.

The resulting graph is represented as a list of triplets: source id, edge label (author of) and target id. The generator was configured to generate 8 experiments by incrementally creating a graph with vertices with a power of 10, that is from 10 to 10^8.

We performed our tests over a Lenovo ThinkPad P51 with a 3.00 GHz (until 4.00 GHz) Intel Xeon processor and 64 GB of RAM at 2.400 MHz. The tests were performed over a ferromagnetic Hard Disk at 5400 RPM with an NTFS File System. We evaluate THoSP using the two pattern matching queries provided in the running example. Given that the secondary memory representation is a simple extension of the one used for nested graphs, we assume that our data serialization is always outperforming with repsect to graph libraries as discussed in our previous work on graph joins, where a similar graph data structure was adopted [**?**]. We used default configurations for both **Neo4J**, **PostgreSQL** and **ArangoDB**, while we changed the cache buffer configurations for Virtuoso (as suggested in the configuration file) for 64 GB of RAM; we also kept default multithreaded query execution plan. For Neo4J and PostgreSQL we kept the same default policies as depicted for GCEA in Subsection **??**. Therefore, we must only describe the conditions under which we performed the time execution experiments for **ArangoDB**. Its AQL queries were evaluated directly through the `arangosh` client and benchmarked using the `getExtra()` method. Given that little documentation is provided with respect to the internal plan's implementations, we referred to the `explain` procedure provided by the shell itself. All the aforementioned conditions do not degradate the query evaluations. Last, given that all databases (except from Neo4J) were coded in C/C++ and that Neo4J provided the worst overall performances, we implemented our serialization and THoSP only in C++.

At first, we must discuss the operator loading time (Table 2a), directly loaded from the data generated by gMark. We shall compare Virtuoso and PostgreSQL first, since they are both based on a traditional relational database engine using one single table to store a graph. In both cases the graph was only represented in triples: Virtuoso automatically stores an RDF graph, while in PostgreSQL the graph was stored by a table of quadruples, where the first is an edge primary key and the remaining elements are the gMark representations. Given that Virtuoso is transactionless, it performed better at serializing and index data for very small data sets (from 10 to 10^3) while, afterwards, the triple indexing time takes over on the overall performances. On the other hand, ArangoDB has not a relational data representation, and it just serializes the data as JSON objects to which several indices are associated. Given that the only data used to serialize data into ArangoDB are the edges' labels, all the time required to store the data is the indexing time. On the other hand, the Neo4J serialization proves to be inefficient, both because there are no constraints for data duplication and we must always check if the to-be-inserted vertex already exists, and because Lucene invertex indices are more useful to index full text documents than indexing graph data. Finally, our nested graph data structure creates adjacency lists directly when serializing the data, while primary indices are only serialized for granting other possible data accesses, even though they are not directly used by the THoSP algorithm. Please also note that no external primary index will be used during the THoSP query evaluation, given that only the adjacency lists information is required to join the edges in a two hop distance scenario.

Let us now consider the graph nesting time (Table 2b): albeit no specific triplet or key are associated to the stored graph, PostgreSQL appears to be more performant than Vir-

Operands Size Vertices ($\|V\|$)	**Operand Loading Time (C/C++)** (ms) PostgreSQL	Virtuoso	ArangoDB	Neo4J (Java)	**Nested Graphs** (C++)
10	8	3.67	43	3,951	0.23
10^2	18	6.86	267	4,124	0.65
10^3	45	23.53	1,285	5,256	5.54
10^4	225	371.40	11,478	11,251	39.14
10^5	1,877	3,510.96	135,595	1,193,492	376.07
10^6	19,076	34,636.80	1,362,734	>1H	4,016.06
10^7	184,421	364,129.00	>1H	>1H	47,452.10
10^8	1,982,393	>1H	>1H	>1H	527,326.00

(a) *Operand Loading and Indexing Time*. PostgreSQL and Neo4J have transactions, while Virtuoso and ArangoDB are transactionless. Nested Graphs are our proposed method which is transactionless.

Operands Size Vertices ($\|V\|$)	Matched Graphs ($\|m_V(G_o)\| + \|m_E(G_o)\|$)	**Two HOp Separated Pattern Time (C/C++)** (ms) PostgreSQL	Virtuoso	ArangoDB	Neo4J (Java)	**THoSP** (C++)
10	3	2.10	11	3.89	681.40	0.11
10^2	58	9.68	63	12.34	1,943.98	0.14
10^3	968	17.96	63	15.00	>1H	0.46
10^4	8,683	69.27	364	46.74	>1H	4.07
10^5	88,885	294.23	4,153	508.87	>1H	43.81
10^6	902,020	2,611.48	50,341	7,212.19	>1H	563.02
10^7	8,991,417	25,666.14	672,273	922,590.00	>1H	8,202.93
10^8	89,146,891	396,523.88	>1H	>1H	>1H	91,834.20

(b) *Graph Nesting Time*. PLease note that the Graph Join Running Time. Each data management system is grouped by its graph query language implementation. This table clearly shows that the definition of our query plan clearly outperforms the default query plan implemented over those different graph query languages and databases.

tuoso in performing graph queries. Please also note that the Virtuoso query engine rewrites the SPARQL query into SQL and, hereby, two SQL queries were performed in both cases. Since both data were represented in a similar way in secondary memory, the completely different performance between the two databases must be attributed to an inefficient rewriting of the SPARQL query into SQL. In particular, the nested representation using JSON array for PostgreSQL proved to be more efficient than returning a full RDF graph represented as triplets, thus arguing in favour of document stores. The PostgreSQL's efficiency is attributable to the run-time indexing time of the relational tables, that is shared with ArangoDB, where the indices are created ad loading time instead: in both cases a single join operation is performed, plus some (either runtime or stored) index access time. Both PostgreSQL and ArangoDB use GroupBy-s to create collections of nested values, separately for both vertices and edges. As observed in the previous paragraph, no primary index is used while performing the THoSP query, and adjacency graphs are returned using the same data structure used for graph joins: one single vertex is returned alongside the set of outgoing edges. Moreover, the nesting result is not created by using group by-s, but by sparsely creating an index that associates the container to its content: as a result, our query plan does not generate an additional cost for sorting and collecting all the elements because it is provided during the graph traversal phase. Thus, the choice of representing the nesting information as a separate index proves to be particularly efficient.

6. CONCLUSIONS

This paper introduces an algorithm for graph nesting jointly with graph traversal queries. Moreover, a class of graph traversal queries that can be optimized is identified, among which a graph algorithm (THoSP) is proposed whenever the nesting vertices are generated by two graph vertices appearing at most at two steps within the original graph operand. By comparing the execution of our algorithm with our graph data structures, we also know that our choice of providing the containment as an external index provides better performances, thus avoiding an additional data grouping phase. Moreover, this algorithm shows that the definition of a separate index for providing the algorithmic result of graph nesting is beneficial, because it allows to return the nested graph as a simple graph having additional nesting informations. This solution was possible due to the assumptions of our proposed graph data model, where it is showed that it is possible to refer any time to the elements that are going to be created later on within the computation, by simply deterministically knowing which the id belonging to the element that is going to be created. Therefore, this paper proves that the representation of nested graph may lead to the solution of current graph querying problems in a tractable way. Nevertheless, we believe that further studies will have to be done on the class of GROQ problems, thus extending our work on THoSP.

7. REFERENCES

[1] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gMark: Schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, 29(4):856–869, 2017.

[2] G. Barabucci, A. Di Iorio, S. Peroni, F. Poggi, and F. Vitali. Annotations with earmark in practice: A fairy tale. In *Proceedings of the 1st International Workshop on Collaborative Annotations in Shared Environment: Metadata, Vocabularies and Techniques in the Digital Humanities*, DH-CASE '13, pages 11:1–11:8, New York, NY, USA, 2013. ACM.

[3] G. Bergami, M. Magnani, and D. Montesi. A join operator for property graphs. In *Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference (EDBT/ICDT 2017), Venice, Italy, March 21-24, 2017.*, 2017.

[4] M. Berlingerio, M. Coscia, and F. Giannotti. Finding redundant and complementary communities in multidimensional networks. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 2181–2184, New York, NY, USA, 2011. ACM.

[5] U. Brandes, M. Eiglsperger, J. Lerner, and C. Pich. Graph markup language (GraphML). In R. Tamassia, editor, *Handbook of Graph Drawing and Visualization*. CRC Press, 2007.

[6] P. Bródka and P. Kazienko. Multilayered social networks. In *Encyclopedia of Social Network Analysis and Mining*, pages 998–1013. 2014.

[7] C. Chen, X. Yan, F. Zhu, J. Han, and P. S. Yu. Graph olap: Towards online analytical processing on graphs. In *ICDM*, pages 103–112. IEEE Computer Society, 2008.

[8] G. Cheng, C. Jin, and Y. Qu. HIEDS: A generic and efficient approach to hierarchical dataset summarization. In *Procs. of IJCAI 2016*, pages 3705–3711, 2016.

[9] L. Etcheverry and A. A. Vaisman. Qb4olap: A vocabulary for olap cubes on the semantic web. In J. Sequeda, A. Harth, and O. Hartig, editors, *COLD*, volume 905 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[10] D. A. Freedman. *Statistical Models. Theory and practice*. Cambridge University Press, 2009.

[11] D. Genest and E. Salvat. A platform allowing typed nested graphs: How cogito became cogitant (research note). In *Conceptual Structures: Theory, Tools and Applications, 6th International Conference on Conceptual Structures, ICCS '98, Montpellier, France, August 10-12, 1998, Proceedings*, pages 154–164, 1998.

[12] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[13] R. C. Holt, A. Schürr, S. E. Sim, and A. Winter. Gxl: A graph-based standard exchange format for reengineering. *Sci. Comput. Program.*, 60(2):149–170, 2006.

[14] W. Jakawat, C. Favre, and S. Loudcher. OLAP Cube-based Graph Approach for Bibliographic Data. In *SOFSEM 2016*, Harrachov, Czech Republic, Nov. 2015.

[15] M. Junghanns, A. Petermann, and E. Rahm. Distributed grouping of property graphs with gradoop. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*, pages 103–122, 2017.

[16] C. A. Kulikowski and S. M. Weiss. Representation of expert knolwedge for consultation: The CASNET and EXPERT proejcts. In *Artificial Intelligence in Medicine*, number 2. Westview Press, Boulder, Colorado, 1982.

[17] P. Odifreddi. *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers (Studies in Logic and the Foundations of Mathematics)*. North Holland, new ed edition, Feb. 1992.

[18] M. Park, H. Ahn, and K. P. Kim. Workflow-supported social networks: Discovery, analyses, and system. *Journal of Network and Computer Applications*, 75:355 – 373, 2016.

[19] A. Poulovassilis and M. Levene. A nested-graph model for the representation and manipulation of complex objects. *ACM Trans. Information Systems*, 12(1):35–68, 1994.

[20] Q. Qu, F. Zhu, X. Yan, J. Han, P. S. Yu, and H. Li. *Efficient Topological OLAP on Information Networks*, pages 389–403. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[21] A. Richter, J. Heidemann, M. Klier, and S. Behrendt. Success measurement of enterprise social networks. *Wirtschaftsinformatik*, (20), 2013.

[22] A. Saeedi, E. Peukert, and E. Rahm. Comparative evaluation of distributed clustering schemes for multi-source entity resolution. *ADBIS*, 2017.

[23] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. SIGMOD, pages 567–580, 2008.

[24] E. Vasilyeva, M. Thiele, C. Bornhövd, and W. Lehner. Leveraging flexible data management with graph databases. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 12:1–12:6, New York, NY, USA, 2013. ACM.

[25] J. Xie, S. Kelley, and B. K. Szymanski. Overlapping community detection in networks: The state-of-the-art and comparative study. *ACM Comput. Surv.*, 45(4):43:1–43:35, Aug. 2013.

[26] D. Yin and H. Gao. A flexible aggregation framwork on large-scale heterogeneous information networks. In *Journal of Information Science*, pages 1–18, Feb. 2016.

[27] P. Zhao, X. Li, D. Xin, and J. Han. Graph cube: On warehousing and olap multidimensional networks. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 853–864, New York, NY, USA, 2011. ACM.