THoSP: an Algorithm for Nesting Property Graphs

.

.

.

.

ABSTRACT

Despite the existance of valuable graph operations creating new vertices and edges from existing graphs, such as graph transformations and graph summarization, an operator allowing both to summarize a subgraph content into one single vertex or edge and to keep the information of the contained vertices and edges is missing. This deficiency is not only due to the operators' definitions, but has to be sought on the current graph data model definition not supporting nestings for overlapping subgraphs. In this paper we propose a novel graph data structure for nested property graphs. Based thereon, we also propose a graph nesting operator and a specific algorithm, THoSP, outperforming the solution propoed by graph (Cypher, SPARQL) relational (SQL) and document oriented (ArangoDB) databases.

ACM Reference Format:
, , , and . 2018. THoSP: an Algorithm for Nesting Property Graphs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Graphs allow flexible analyses of relationships among data objects. Thus, graph data management systems play an increasing role in present data analytics. Graphs have been already used as a fundamental data structure to represent data within different contexts such as corporate data [14, 17], social networks [3, 20] and linked data [19]. Despite an increasing number of applications, a general operator that aggregates a single graph in a roll-up fashion is still missing. The operation of adding structural aggregations to an existing graph is called *graph nesting*. A respective operator shall not only create a new graph of *nested vertices* and *nested edges*, each containing subgraphs of the original input graph, but also preserve the vertices and edges that are not affected by the actual operation. Further on, the operator must ensure that the nested elements can be freely unnested such that the original graph may be obtained back again. Vertices or edges of the original graph will be called *members* of a nested vertex or edge, if they appear in its underlying subgraph.

In a resulting nested graph, edges connecting nested vertices express that members of the nested vertices are connected by an edge or, more general, by a path in the original graph. In contrast to this general approach, current literature distinguishes between *vertex summarization* and *path summarization*. Thus, it is not possible to define a single algorithm that evaluates both kinds of patterns at the same time. Before outlining our proposed algorithmic solution, let's have a look on these existing approaches:

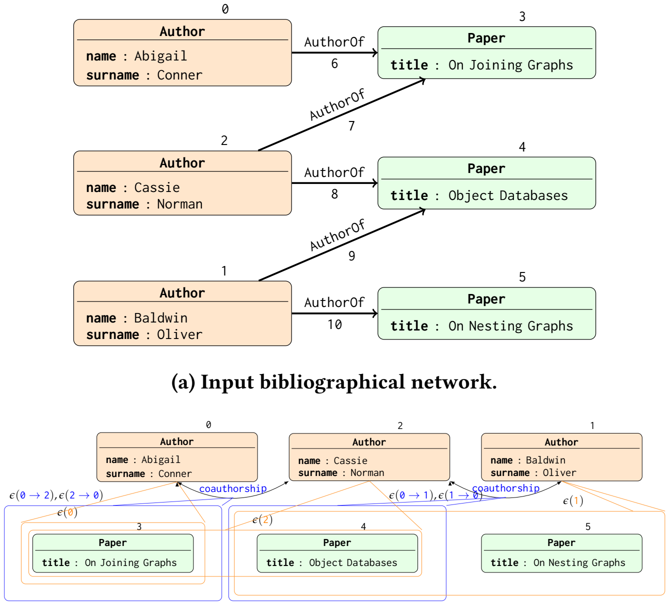
The *vertex summarization* strategies group vertices in the manner of the relational `GROUP BY` operation and aggregate edges accordingly [11]. In this class of operations summarized edges can only be formed by edges that directly connect members of summarized vertices in the original graph. In other words, these approaches cannot freely nest edges, for example, it is not possible to aggregate paths. Since most of vertex summarization techniques are based on graph partitioning, they further provide no support for nested vertices and edges with overlapping members [10, 18, 21]. Exceptions are HEIDS [5] and Graph Cube [22], which perform graph summarizations of one single graph over a collection of non pairwise disjoint subgraphs. However, the union of these underlying subgraphs must be equivalent to the original graph, i.e., it is not possible to take vertices and edges of the original graph over to the summarized graph or to represent outliers that belong to no group.

By contrast, *path summarization* techniques allow the aggregation of multiple paths among pairs of source and target vertices that share the same properties. Currently, approaches to path summarization can only be found within graph query languages. These languages also support vertex summarization, but no combination of both approaches in a single step. Cypher, the query language of the productive graph database Neo4j, can perform distinct aggregations only within distinct `MATCH` clauses. SPARQL 1.1, the standard query language of the resource description framework (RDF), requires to combine vertex and path aggregation with a `UNION` operator: this implies that the same input graph must be visited twice.

This paper shows that such query language limitations can be reduced by using a graph nesting operator, which performs both vertex and path summarization queries concurrently with only a single visit of the input graph. We study specific graph patterns and propose algorithmic optimizations and a specific physical model for efficient graph nesting. Hereby, we show that our algorithmic approaches reduce the time complexity of the visiting and nesting problem. Further on, our optimized data structure requires less indexing time than our competitors.

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 "roll*

Conference'17, July 2017, Washington, DC, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) Input bibliographical network.

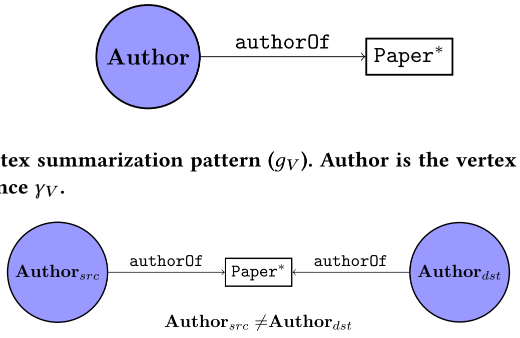
(b) Nested result: given two Authors a and a', there exist two coauthorship edges, $a \to a'$ and $a' \to a$ if and only if they share some authored paper contained respectively in $\epsilon(a \to a')$ and $\epsilon(a' \to a)$. Moreover, each author a is associated to the set of his authored papers $\epsilon(a)$.

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

up" the network into a coauthorship network (Figure 1b). In the resulting graph, each Author *appearing in the input graph is transformed into a nested* Author(2) *vertex containing all its authored papers. Moreover, a* coAuthor *edge between two* Author(2) *is drawn if the two disinct authors are co-authors; each one of those edges contains their co-authored* Papers*. Last, we also want to exclude* coAuthor *hooks over the same vertex.*

Figure 2 represents the desired vertex (g_V) and path (g_E) summarization patterns: the former will create a nested Author(2) *nested vertex and the latter will create a* coAuthor *nested edge. Given that g_V appears twice in g_E, we may also pre-istantiate the pattern g_V by visiting g_E once. 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 summarization returns all the papers authored by two different authors and creates an edge between the two previously nested vertices.*

This problem can be solved by visiting the graph only once; if the current vertex is a Paper*, traverse backwards all the* authorOf *edges, thus reaching all of its* Authors*, which are going to be* coAuthor *for at least the current paper. Instead of associating the nesting content at the end of the graph visiting process, we can incrementally define the subgraph to be nested by using a separated nesting index: by visiting the two distinct* Author *vertices adjacent to the current* Paper*, the latter one shall be contained in both final* Author(2) *vertices, thus allowing the definition of a* coAuthor *edge. 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.

The fulfilment of the former scenario is achieved by this paper via the following three contributions:

- We implement the aforementioned solution into a nested graph data model, where the logical model differs from the physical one.
- **Graph Nesting Operator** (Section 4); 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.

The source code for THoSP is provided at `[Link removed for double-blind review]`.

2 RELATED WORKS

2.1 Nested Graph Representations

Statecharts [8] represent one of the first applications of nested graphs to complex systems modelling. This choice allowed the representation of multiple abstraction levels at the same time: each node represents a state or "configuration" of the system, and each edge represents a transaction between two different states on a given event. 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. Given that this model was not designed for data representations, vertices and edges are labelled but cannot contain any property-value association. 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 has also been used for both modelling the evolution of *pathophysiological* states and to describe the clinical treatments to which the patient must undergo. This model also allowed to subdivide each treatment into smaller and consequential procedures [12].

Statecharts were also adopted as a basis for the subsequent **hypernode** data model [15]. Unlike statecharts, hypernodes allows neither edge labelling nor external and internal edges. 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. A first extension of the hypernode model towards data representation is represented by CoGITaNT [7], 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 [2] and GXL [9].

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, [6] proposes the definition of 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 implies that each possible computation over the data view has to be always recomputed on top of the raw data like for classical ROLAP systems, thus thwarting the benefits of updating the intermediate query result. On the other hand, the last type of approaches have been recently widely investigated, and seem to be more promising with respect to optimization techniques. Graph data structures are now [4, 16, 18] 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, the `array_agg` aggregation function allows to group the result-set into arrays. We represent our graph by storing the triples defining an edge as the following relation:

$$\texttt{Edge}(\mathit{edgeId}, \mathit{sourceId}, \mathit{edgeLabel}, \mathit{targetId})$$

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. The overall graph nesting cannot be created in one single SQL query, because we cannot distinctively group the same dataset in different ways, but we must perform two distinct aggregations (see Listing 1 in Appendix).

Despite the fact that SPARQL may represent the graph nesting query as a single statement, a `UNION` clause implies a separate visit for the two graph patterns. The first pattern presented in Listing 2 (see Appendix) allows to traverse those 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 the latter is not supported by *librdf* which is 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 one single coAuthorship edge (`?newedge`).

Given that ArangoDB is a NoSQL database relying on a document-oriented storage, which is hereby prone to both represent and return nested content, we also consider its query language, AQL. 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 ArangoDB format, where vertices are indexed by id while edges are also indexed by source and target vertex id. Even though AQL returns JSON documents instead of relational table, We can state that the resulting query plan provides the exact same result as the one produced by the SQL query.

Last, Listing 4 provides an example of Neo4J: even if Neo4J's property graph model does not directly nest the graphs inside one vertex or edges, we can associate to each vertex or edge the vertices' and edges' ids that it may contain. This solution can be achieved by first matching the vertex summarization pattern in Figure 2a and then perform an Author group by (`with`), and then nest the collection of authored Papers via `collect`. Last, we match the path summarization pattern presented in Figure 2b and group it by source and destination Author, then we create the coAuthorship edge containing the co-authored Paper's id. 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 NESTED GRAPHS

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."`). Property-value associations are represented as 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**.

4 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 C, a* ***nested graph classifier*** *operator g_κ maps a nested property graph G_o into a nested property graph collection $\{G_C\}_{C \in C, G_C \neq \emptyset}$ of subgraphs of G_o. Such operator uses a classifier function $\kappa\colon \mathcal{V} \cup \mathcal{E} \to \wp(C)$ 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:*

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

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:

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

where dt is an arbitrary bijection associating a n-tuple in $\mathbb{N}^n$ into one single number $\mathbb{N}$ [13]. 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 $\eta^{\textbf{\textit{keep}}}_{g_V, g_E, \mu_\Omega, \mu_E, \iota_G}$ 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:*

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

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:*

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

where $f \oplus g$ denotes the extension of the (finite) function f with another (finite) function g.

4.1 Two HOp Separated Patterns Algorithm

In order to solve our specific graph nesting problem as presented in the introduction, we want to evaluate ν when the non-traversed

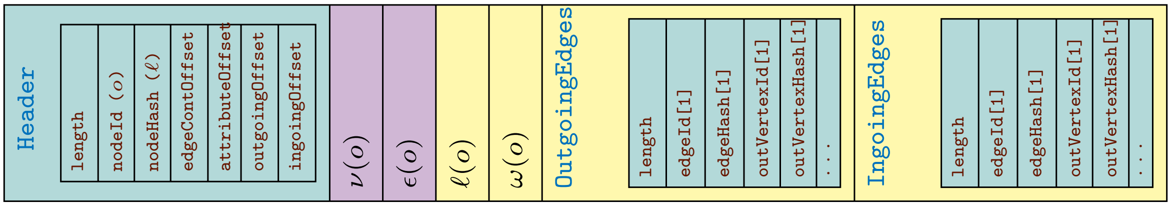


Figure 3: Serialized data structure representing an extended adjacency list for one nested vertex o. The header contains some basic information (such the representation size of o, its id and associated hash) and the offsets to the remaining fields. ν and ϵ are empty when the serialized graph represents a basic property graph as the one in Figure 1a.

vertices and edges are not preserved (**keep** = `false`), where g_V and g_E are the ones represented in Figure 1 and the UDFs associate to each nested vertex the authors' informations and creates nested edges with `coAuthorship` label and no associated tuple. In particular:

$$\mu_E(G_C) = \left\langle f_C(\gamma_E^{src}), f_C(\gamma_E^{dst}) \right\rangle \quad s.t. \quad g_E \xrightarrow{f_C} G_C$$

$$\mu_\Omega(G_C) = \begin{cases} ([\text{“coAuthor”}],\ \emptyset) & G_C \in g_E(G_{(c,i)}) \\ (\ell(f_C(\gamma_V))),\ \omega(f_C(\gamma_V))) & G_C \in g_V(G_{(c,i)}) \end{cases}$$

The *input data represnetation* for the THoSP algorithm is presented in Figure 3, which represents an extension of the usual graph adjacency lists: in addition to the information pertaining to both the ingoing and outgoing edges, each vertex o has an associated `header`, where its id (o), its associated hash and the offset pointing to other serialzied fields, such as the information of the nested vertices and edges a set of labels ($\ell(o)$), and eventually its property-value representation ($\omega(o)$). Last, for each edge we store its id and hash value, as well as the hash and the id of the adjacent vertex. Hash values are used within the proposed THoSP algorithm to store the correspondences with the graph patterns in Figure 2; therefore, each $\ell(o)$ is associated to a distinct hash value. We also suppose that the input graph data to be serialized does not represent an exact adjacency list: for this reason, the graph is firstly created in primary memory without the offset information, and then serialized into secondary memory. This first preprocessing step provides the loading and indexing step of our algorithm, where no further ancillary indexing data structures are serialized.

Algorithm 1 provides the desired solution as sketched in the paper's introduction: we focus on intersecting a specific class of graph visiting patterns that can be possibly optimized as in the former solution. For each vertex pattern we're going to elect one vertex as a **vertex grouping reference** (γ_V), in which we're going to nest the matched vertices and edges during the graph traversal. 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. In particular, this paper focuses on g_E where γ_E^{src} and γ_E^{dst} are separated by a two-edge (hop) distance (Line 5). We must first identify a sub-pattern α that is going to be visited only once within the graph (Line 7), 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 (c, v) matched by α (Line 7) we know that we must (possibly) visit all the edges going from (c, v) towards the vertices γ_E^{src} and γ_E^{dst}, that substantially are γ_V. 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 (c, v) meeting the grouping references (Line 14). Therefore, we know whether we finished visiting our patterns after exhaustively matching all the elements within the pattern. As a consequence, a "path join" is performed between the two nested patterns (Line 13): this is evident from the two vertex nested for loops appearing in the algorithm.

Algorithm 1 Two HOp Separated Patterns Algorithm (THoSP)

```
 1: procedure PartitionHashJoin(($g_V$, $\gamma_V$), ($g_E$, $\gamma_E^{src}$, $\gamma_E^{dst}$); $G_o$)
 2:     File $AdjFile$ = Open($G_o$);
 3:     File $Nesting$ = Open(new);
 4:     Adjacency $toSerialize$ =
            new Map<Vertex,<Edge,Vertex»();
 5:     $\alpha := g_V \cap g_E \backslash (\gamma_V \cup \gamma_E^{src} \cup \gamma_E^{dst})$;
 6:     for each vertex $(c, v)$ in $AdjFile$ do
 7:         if $(c, v) \vDash \alpha$ then
 8:             for each $((c', u), e, (c, v)) \vDash \gamma_V$ do
 9:                 $u' := dt(1, dt(c', u))$
10:                 $NestingIndex$.write($\langle u', u \rangle$)
11:                 $NestingIndex$.write($\langle u', e \rangle$)
12:                 $NestingIndex$.write($\langle u', v \rangle$)
13:                 for each $((c'', w), e, (c, v)) \vDash \gamma_V$ do
14:                     if $((c', u), e, (c, v), e', w) \vDash \gamma_E$ then
15:                         $w' := dt(1, dt(0, w))$
16:                         $\varepsilon := dt(1, dt(u, w))$
17:                         $NestingIndex$.write($\langle \varepsilon, u \rangle$)
18:                         $NestingIndex$.write($\langle \varepsilon, e \rangle$)
19:                         $NestingIndex$.write($\langle \varepsilon, w \rangle$)
20:                         $NestingIndex$.write($\langle \varepsilon, e' \rangle$)
21:                         $NestingIndex$.write($\langle \varepsilon, v \rangle$)
22:                         $toSerialize$.put($u'$,$\langle \varepsilon, w' \rangle$)
        $AdjFile$.serialize($toSerialize$);
```

Our physical data model differentiates the **input data represnetation** from the **query result**. We suppose that the latter is only used by the user to read the outcome of the nesting process as in other query languages (such as SPARQL and SQL) and does not have to produce "materialized views". Therefore, the result of the graph query itself can postpone the creation of a complete "materialized view", which will later use the same representation of the input data by using both the id information and the application of the User Defined Functions. In particular, the dovetailing function is used to associate both the nested vertices, u' and w', and the nested edge ε to their grouping references, thus allowing to easily go back to the original grouping references by using the inverse function of dt, thus allowing the application of the user defined functions.

Last, the association between the nested vertices (and edges) f and its content within the input graph c is stored in a $NestingIndex$ file as a set of pairs $\langle f, c\rangle$. By doing so, we omit the Group By cost which affects the previously seen query languages, thus allowing to an overall better performance.

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 (c, v) is present alongside another element matching γ_E^{src} or γ_E^{dst}.

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. 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. We used default configurations for **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. PostgreSQL queries were evaluated through the `psql` client and benchmarked using both `explain analyze` and `\timing` commands. Virtuoso was benchmarked through the Redland RDF library using directly the `librdf_model_query_execute` function. AQL queries over ArangoDB were evaluated directly through the `arangosh` client and benchmarked using the `getExtra()` method. Cypher queries were evaluated using the Java API through the `execute` method of an `GraphDatabaseService` object. 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 1a), 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 1b): albeit no specific triplet or key are associated to the stored graph, PostgreSQL appears to be more performant than Virtuoso 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,

Operands Size	**Operand Loading and Indexing Time (C/C++)** (ms)				
Vertices ($\lvert V\rvert$)	PostgreSQL	Virtuoso	ArangoDB	Neo4J (Java)	**Nested Graphs** (C++)
10	8	3.67	43	3,951	0.13
10^2	18	6.86	267	4,124	0.33
10^3	45	23.53	1,285	5,256	3.51
10^4	225	371.40	11,478	11,251	31.83
10^5	1,877	3,510.96	135,595	1,193,492	337.00
10^6	19,076	34,636.80	1,362,734	>1H	3,694.56
10^7	184,421	364,129.00	>1H	>1H	44,063.50
10^8	1,982,393	>1H	>1H	>1H	518,361.00

(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		**Two HOp Separated Pattern Time (C/C++)** (ms)				
Vertices	Matched subraphs	SQL Dialect	SPARQL	AQL	Cypher	**THoSP** (C++)
($\lvert V\rvert$)	($\lvert m_V(G_o)\rvert + \lvert m_E(G_o)\rvert$)	(PostgreSQL)	(Virtuoso)	(ArangoDB)	(Neo4J – Java)	
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.*** **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.**

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 `GROUP BY`-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.

REFERENCES

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

[2] Ulrik Brandes, Markus Eiglsperger, Jürgen Lerner, and Christian Pich. 2007. Graph Markup Language (GraphML). In *Handbook of Graph Drawing and Visualization*, Roberto Tamassia (Ed.). CRC Press.

[3] Piotr Bródka and Przemyslaw Kazienko. 2014. Multilayered Social Networks. In *Encyclopedia of Social Network Analysis and Mining*. 998–1013. https://doi.org/10.1007/978-1-4614-6170-8_239

[4] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and Philip S. Yu. 2008. Graph OLAP: Towards Online Analytical Processing on Graphs.. In *ICDM*. IEEE Computer Society, 103–112. http://dblp.uni-trier.de/db/conf/icdm/icdm2008.html#ChenYZHY08

[5] Gong Cheng, Cheng Jin, and Yuzhong Qu. 2016. HIEDS: A Generic and Efficient Approach to Hierarchical Dataset Summarization. In *Procs. of IJCAI 2016*. 3705–3711. http://www.ijcai.org/Abstract/16/521

[6] Lorena Etcheverry and Alejandro A. Vaisman. 2012. QB4OLAP: A Vocabulary for OLAP Cubes on the Semantic Web.. In *COLD (CEUR Workshop Proceedings)*, Juan Sequeda, Andreas Harth, and Olaf Hartig (Eds.), Vol. 905. CEUR-WS.org.

[7] David Genest and Eric Salvat. 1998. 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*. 154–164. https://doi.org/10.1007/BFb0054912

[8] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (June 1987), 231–274. https://doi.org/10.1016/0167-6423(87)90035-9

[9] Richard C. Holt, Andy Schürr, Susan E. Sim, and Andreas Winter. 2006. GXL: A graph-based standard exchange format for reengineering. *Sci. Comput. Program.* 60, 2 (2006), 149–170.

[10] Wararat Jakawat, Cécile Favre, and Sabine Loudcher. 2015. OLAP Cube-based Graph Approach for Bibliographic Data. In *SOFSEM 2016*. Harrachov, Czech Republic.

[11] Martin Junghanns, André Petermann, and Erhard Rahm. 2017. 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*. 103–122.

[12] Casimir A. Kulikowski and Sholom M. Weiss. 1982. Representation of Expert Knolwedge for Consultation: The CASNET and EXPERT proejcts. In *Artificial Intelligence in Medicine*. Westview Press, Boulder, Colorado.

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

[14] Minjae Park, Hyun Ahn, and Kwanghoon Pio Kim. 2016. Workflow-supported social networks: Discovery, analyses, and system. *Journal of Network and Computer Applications* 75 (2016), 355 – 373.

[15] A. Poulovassilis and M. Levene. 1994. A Nested-Graph Model for the Representation and Manipulation of Complex Objects. *ACM Trans. Information Systems* 12, 1 (1994), 35–68.

[16] Qiang Qu, Feida Zhu, Xifeng Yan, Jiawei Han, Philip S. Yu, and Hongyan Li. 2011. *Efficient Topological OLAP on Information Networks*. Springer Berlin Heidelberg, Berlin, Heidelberg, 389–403.

[17] Alexander Richter, Julia Heidemann, Mathias Klier, and Sebastian Behrendt. 2013. Success Measurement of Enterprise Social Networks. *Wirtschaftsinformatik* 20 (2013).

[18] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. 2008. Efficient Aggregation for Graph Summarization *(SIGMOD)*. 567–580.

[19] Elena Vasilyeva, Maik Thiele, Christof Bornhövd, and Wolfgang Lehner. 2013. Leveraging Flexible Data Management with Graph Databases. In *First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*. ACM, New York, NY, USA, Article 12, 6 pages.

[20] Jierui Xie, Stephen Kelley, and Boleslaw K. Szymanski. 2013. Overlapping Community Detection in Networks: The State-of-the-art and Comparative Study. *ACM Comput. Surv.* 45, 4, Article 43 (Aug. 2013), 35 pages.

[21] Dan Yin and Hong Gao. 2016. A flexible aggregation framwork on large-scale heterogeneous information networks. In *Journal of Information Science*. 1–18.

[22] Peixiang Zhao, Xiaolei Li, Dong Xin, and Jiawei Han. 2011. Graph Cube: On Warehousing and OLAP Multidimensional Networks. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. ACM, New York, NY, USA, 853–864.

A QUERY BENCHMARKS

Listing 1: Graph Nesting in PostgreSQL's SQL dialect. Two distinct tables are created for both vertices and edges.

```
-- 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. We use 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))
  }
}
```

Listing 3: Graph Nesting in ArangoDB using AQL. All the fields marked with an underscore represent externally indexed structures.

```
-- 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.

```
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)
```