A Join Operator for Property Graphs

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

Matteo Magnani
Uppsala University
Department of Information Technology
Uppsala, Sweden
matteo.magnani@it.uu.se

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

ABSTRACT

In the graph database literature the term "join" does not refer to an operator combining two graphs, but involves path traversal queries over a single graph. Current languages express binary joins through the combination of path traversal queries with graph creation operations. Such solution proves to be not efficient.

In this paper we introduce a binary graph join operator and a corresponding algorithm outperforming the solution proposed by query languages for either graphs (Cypher, SPARQL) and relational databases (SQL). This is achieved by using a specific graph data structure in secondary memory showing better performance than state of the art graph libraries (Boost Graph Library, SNAP) and database systems (Sparksee).

Keywords

Graph Database, Property Graph, Join, Partition Hash Join

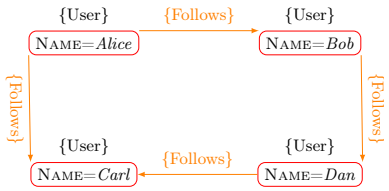
1. INTRODUCTION

Despite the term "join" appearing in the graph database literature, such operator cannot be used to combine two distinct graphs, as for table joins in the relational model. Such joins are *path joins* running over a single graph [1]: they are used for graph traversal queries [13] where vertices and edges are considered as relational tables [25, 16]. The result of such *path joins* cannot be directly used to combine values from different sources (e.g. join two distinct vertices appearing in different graphs alongside with their values), and hence supplementary graph operations are required. SPARQL allows to access multiple graph resources through *named graphs* and performs graph traversals one graph at a time through *path joins* [12, 3, 27]. At this point the `CONSTRUCT` clause is required if we want to finally combine the traversed paths from both graphs into a resulting graph. Similarly, Cypher's `CREATE` clause has to be used to generate new vertices and edges from graph patterns extracted through the `MATCH...WHERE` clause, and intemediate results are merged with `UNION ALL`. While current graph query languages allow to express our proposed graph join operator as a combination of the aforementioned operators, our study shows that our specialized graph join algorithm outperforms the evaluation of the graph join with existing graph and relational query languages.

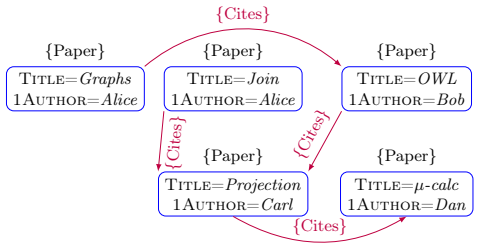
As for relational databases, they solve common graph queries efficiently, so graph database management systems rely either on relational database engines [1, 21, 11] or on column store databases [25, 7]. Moreover, relational databases already have efficient implementations for (equi) join algorithms [23]. We want to show that graph joins over the relational data model are not inefficient. Before all, let us see an example of a graph join query:

EXAMPLE 1. *Consider an on-line service such as* ResearchGate *(Figure 1a, or Academia.edu) where researchers can* follow *each others' work, and a* citation *graph (Figure 1b). Now we want to "return the paper graph where a paper cites another one iff. the* first author 1AUTHOR *of the first paper* follows *the* 1AUTHOR *of the second. (Figure 1c)". The* ResearchGate *graph does not contain any edge regarding the* references*, while the* Reference *graph does not contain any information pertaining to the follow relations. This demands a join between the two graphs: as a first step we join the vertices together as in the relational model (vertices are considered as tuples using* NAME = 1AUTH *as a vertex equi-join predicate, θ) and then combine the edges from both graphs. Accordingly to the query formulation, we establish an edge between two joined vertices only if the source has a paper citing the destination, and the user in the source follows the user in the destination.*

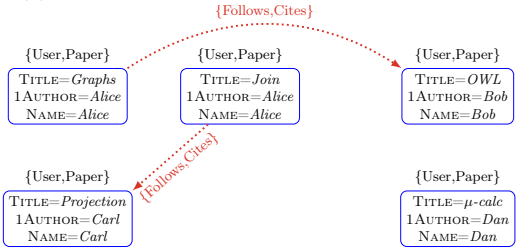
Let us now examine the graph join implementation within the relational model: vertices and edges are represented as two relational tables ([25], Figure 2a). In addition to the attributes within the vertices' and the edges' tables, we assume that each row (on both vertices and edges) has an attribute `id` enumerating vertices and edges. Concerning SQL interpretation of such graph join, we first join the vertices (see the records linked by θ lines in Figure 2a). Then the edges are computed through the join query provided in Figure 2b: the root and the leaves are the result of the θ join between the vertices, while the edges appear as the intermediate nodes. An adjacency list representation of a graph, as the one proposed in the current paper, reduces the joins within the relation solution to one (each vertex and edge is traversed only once), thus reducing the number of required operation to create the resulting graph. Other inefficiency



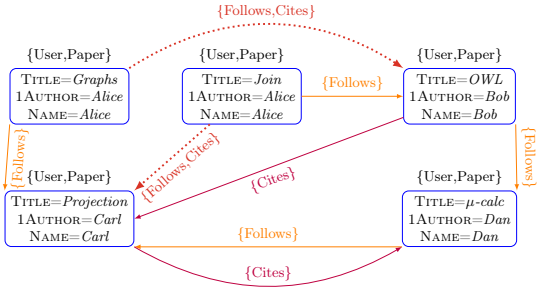
(a) ResearchGate graph, Follower relations.



(b) Reference graph, citation relations. Each paper has a first author.

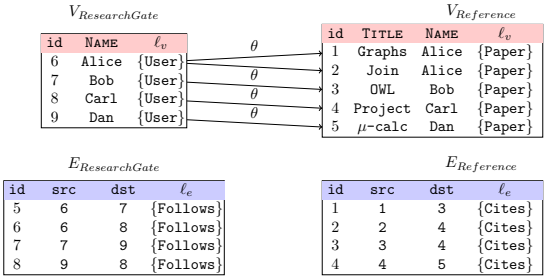


(c) First query: ResearchGate$\bowtie^{\wedge}_{\texttt{Name=1Author}}$Reference

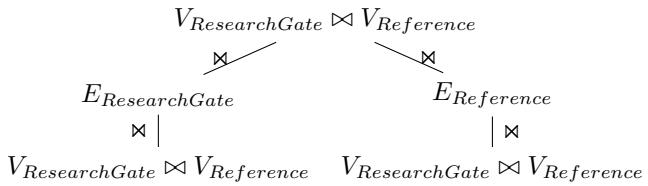


(d) Second query: ResearchGate$\bowtie^{\vee}_{\texttt{Name=1Author}}$Reference

Figure 1: Example of a Graph Database for an Enterprise. Dotted edges remark edges shared between the two different joins.



(a) Representing the operands' vertices and edges with tables. The θ join for the vertices only involves tables $V_{ResearchGate}$ and $V_{projects}$.



(b) SQL join query plan required to create edges for ResearchGate$\bowtie^{\wedge}_{\texttt{Name=1Author}}$Reference. The leaves acts as the edges' sources while the root as their destinations.

Figure 2: Graphically representing the relational join procedure required to evaluate the first query (Figure 1c).

considerations for graph query languages are provided in the Related Work section (Section 6.1).

Example 1 showed only one possible way to combine the operands' edges, but we can even return edges pertaining to both operands as in the following query: "*For each paper reveal both the direct and the indirect dependencies (either there is a direct paper citation, or one of the authors follows the other one in ResearchGate)*". The resulting graph (Figure 1d) has the same vertex set than the previous one, but they differ on the final edges. This implies that our graph join definition must be general enough to allow different edge combinations: we refer to those as **edge semantics**, "**es**" for shorthand. This paper provides two contributions:

- **Graph join operator** $\bowtie^{\mathbf{es}}_{\theta}$ (Section 3), combining both vertices (θ) and edges (**es**). A property graph model (Section 2) is used as a data model of choice.
- **Graph Conjunctive Equijoin Algorithm** (Section 4): vertex buckets ordered by hash value are created and the resulting graphs' edges and vertices are produced at the same time. Our solution outperforms the query evaluations in SPARQL, Cypher and SQL (Section 5.2). Since the aforementioned algorithm relies on an *ad hoc* secondary memory data structure, we tested it over different graph libraries (Boost, SNAP) and low level graph databases (Sparksee). Even in this case our solution provide better results with large graphs (Section 5.1).

2. PRELIMINARIES

We model the vertices' and edges' set as **multisets (of tuples)** S of elements s_i, where s_i unequivocally identifies the i-th occurrence of a tuple s in S. Each **tuple** associates to each attribute a value: it is a function $A \mapsto \mathcal{V} \cup \{\texttt{NULL}\}$ mapping each attribute in A to either a value in $\mathcal{V}$ or NULL (ε is the empty tuple). We slightly change the property graph definition in [16] in order to ease the join definition between vertices and edges as later on required by the graph join:

Definition 1 (Property Graph). *A **property graph** is a tuple $G = (V, E, \Sigma_v, \Sigma_e, A_v, A_e, \lambda, \ell_v, \ell_e)$ where (a) V is a multiset of nodes, (b) E is a multiset of edges, (c) Σ_v is a set of node labels, (d) Σ_e is a set of edge labels, (e) A_v is a set of node attributes, (f) A_e is a set of edge attributes,*

(g) $\lambda\colon E \to V \times V$ *is a function assigning node pairs to edges, (h)* $\ell_v\colon V \to \mathcal{P}(\Sigma_v)$ *is a function assigning a set of labels to nodes, and (i)* $\ell_e\colon E \to \mathcal{P}(\Sigma_e)$ *is a function assigning a set of labels to edges.*

This is the baseline for our graph database:

DEFINITION 2 (GRAPH DATABASE). *A **graph database** is a collection of n distinct property graphs* $\{G_1, \ldots, G_n\}$ *represented as a single property graph* $\mathcal{D}$ *with n distinct connected components. From now on we refer to each component simply as **graph**. Each graph is identified by two functions:* $\mathcal{V}\colon \{1, \ldots, n\} \mapsto \mathcal{P}(V)$ *determining the vertices* $\mathcal{V}(i)$ *of the i-th graph and* $\mathcal{E}\colon \{1, \ldots, n\} \mapsto \mathcal{P}(E)$ *determining the edges* $\mathcal{E}(i)$ *of the i-th graph.*

EXAMPLE 2. *Two edges* e_i *and* f_j *come from two distinct graphs, respectively* G_a *and* G_b*, within the same graph database* $\mathcal{D}$*. Edge* e_i *connects vertex* u_h *to* v_k *(*$\lambda(e_i) = (u_h, v_k)$*), while* f_j *connects* u'_h *to* v'_k *(*$\lambda(f_j) = (u'_h, v'_k)$*). Such edges store only the following values:*

$$e_i(\textsc{Time}) = 12{:}04, \quad f_j(\textsc{Day}) = Mon$$

and have the following labels:

$$\ell_e(e_i) = \{\text{ Follow }\}, \quad \ell_e(f_j) = \{\text{ FriendOf }\}$$

For the multiset θ-join, we need a function $\oplus$ combining two tuples for the relational join operator over multisets, where $r_i \oplus t_j$ is a valid multiset element $(r \oplus s)_{i \oplus j}$ and $i \oplus j$ maps each integer pair (i, j) to a single number.

DEFINITION 3 (θ-JOIN). *Given two (multiset) tables* R *and* S *over a set of attributes* A_1 *and* A_2*, the **θ-join** $R \bowtie_\theta S$ [4, 5] is defined as follows:*

$$\{r_i \oplus s_j \mid r_i \in R, s_j \in S, \theta(r_i, s_j), (r_i \oplus s_j)(A_1) = r_i, (r_i \oplus s_j)(A_2) = s_j\}$$

where $(t \oplus t')(A_i)$ *denotes the projection of the tuple* $t \oplus t'$ *over* A_i*. If* θ *is the always true predicate,* θ *can be omitted and, when also* $A_1 \cap A_2 = \emptyset$*, we have a cartesian product.*

If we define $\oplus$ as a linear function (that is for each function H, $H(e_i \oplus f_j) = H(e_i) \oplus H(f_j)$), the θ-join also induces the definition of ℓ_v, ℓ_e and λ for the joined tuples. As a consequence, $\oplus$ must be overloaded for each possible expected output from H (see Definition 7 in Appendix).

EXAMPLE 3. *By continuing the previous example, suppose that the edge* $e_i \oplus f_j$ *comes from a graph join where edges from* G_a *are joined to the ones in* G_b *in a resulting graph, where also vertices* $u_h \oplus u'_h$ *and* $v_k \oplus v'_k$ *appear. So:*

$$(e_i \oplus f_j)(\textsc{Time}) = 12{:}04, \quad (e_i \oplus f_j)(\textsc{Day}) = Mon$$

By $\oplus$*'s linearity, we have that the labels are merged:*

$$\begin{aligned}\ell_e(e_i \oplus f_j) &= \ell_e(e_i) \oplus \ell_e(f_j) = \{\text{Follow}\} \oplus \{\text{FriendOf}\} \\ &= \{\text{Follow}, \text{FriendOf}\}\end{aligned}$$

And the result's vertices are updated accordingly:

$$\begin{aligned}\lambda(e_i \oplus f_j) &= \lambda(e_i) \oplus \lambda(f_j) = (u_h, v_k) \oplus (u'_u, v'_k) \\ &= (u_h \oplus u'_h, v_k \oplus v'_k)\end{aligned}$$

Since all the relevant informations are stored in the graph database, we represent the graph as the set of the minimum information required for the join operation.

DEFINITION 4 (GRAPH). *The i-th **graph** of a graph database* $\mathcal{D}$ *is a tuple* $G_i = (\mathcal{V}(i), \mathcal{E}(i), A^i_v, A^i_e)$*, where* $\mathcal{V}(i)$ *is a multiset of vertices and* $\mathcal{E}(i)$ *is a multiset of edges. Furthermore,* A^i_v *is a set of attributes* $a \in A^i_v$ *s.t. there is at least one vertex* $v_j \in \mathcal{V}(i)$ *having* $v_i(a) \neq$ `NULL`*;* A^i_e *is a set of attributes* $a' \in A^i_e$ *s.t. there is at least one edge* $e_k \in \mathcal{E}(i)$ *having* $e_k(a') \neq$ `NULL`*.*

3. GRAPH JOINS

As we discussed in the introduction, our graph join is based on the combination of vertices and edges: $G_a \bowtie^{\mathbf{es}}_\theta G_b$ expresses the join of graph G_a with G_b where *(i)* we first use a relational θ-join among the vertices, and then *(ii)* we combine the edges using an appropriate user-determined edge semantics, **es**. This modularity is similar to the graph products defined in graph theory literature [14, 17], where instead of a join between vertices they have a cross product, and different semantics are expressed as different graph products. We now provide the graph join definition:

DEFINITION 5 (GRAPH θ-JOIN). *Given two graphs* $G_a = (V, E, A_v, A_e)$ *and* $G_b = (V', E', A'_v, A'_e)$*, a **graph θ-join** is defined as follows:*

$$G_a \bowtie^{\mathbf{es}}_\theta G_b = (V \bowtie_\theta V', E_{\mathbf{es}}, A_v \cup A'_v, A_e \cup A'_e)$$

where θ *is a binary predicate over the vertices and* $\bowtie_\theta$ *the* θ*-join (Definition 3) among the vertices, and* $E_{\mathbf{es}}$ *is a subset of all the possible edges linking the vertices in* $V \bowtie_\theta V'$ *expressed with the* **es** *semantics.*

Given that graph join returns a property graph like the graphs in input, property graphs are closed under the graph join operator via the definition of $\oplus$ for the multiset θ-join.

3.1 Two possible "es" edge semantics

The result of the join between two graphs, ResearchGate (Figure 1a) and References (Figure 1b), produces the same set of vertices regardless of the edge semantics of choice. On the other hand, edges among the resulting vertices change according to the edge semantics. In the first one (Figure 1c) we combine edges appearing in both graphs and linking vertices that appear combined in the resulting graph. We have a **Conjunctive Join**, that in graph theory is known as *Kronecker graph product* [26, 14]. In this case $E_{\mathbf{es}}$ is defined with the "$\wedge$" **es** semantics as an edge join $E_\wedge = E \bowtie_{\Theta_\wedge} E'$, where the $\Theta_\wedge$ predicate is the following one:

$$\Theta_\wedge(e_h, e'_k) = (e_h \in E \wedge e'_k \in E') \wedge \lambda(e_h \oplus e'_k) \in (V \bowtie_\theta V')^2 \quad (1)$$

We can also define a **disjunctive** semantics (Figure 1d), having "$\vee$" as **es**. In this case we want edges appearing either in the first or in the second operand. This means that two vertices, $u_h \oplus u'_h$ and $v_k \oplus v'_k$, could have a resulting edge $e_i \oplus \varepsilon'_j$ even if only $\lambda(e_i) = (u_h, v_k)$ appears in the first operand and ε' is a "fresh" empty edge $\lambda(\varepsilon'_j) = (u'_h, v'_k)$ not appearing in G_b such that $\lambda(e_a \oplus \varepsilon'_b) = (u_h \oplus u'_h, v_k \oplus v'_k)$. Consequently the disjunctive join can be represented as a *full outer join*, where the edges either match in the conjunctive

semantics, or appear in the two distinct graph operands:

$$E_{\vee} = E \rtimes\ltimes_{\Theta_{\wedge}} E' \tag{2}$$

4. ALGORITHM AND DATA STRUCTURE

We now outline our algorithm, GCEA, for θ equijoin predicates, involving an equivalence between attributes or a conjunction of such equivalences. This specific predicate choice was driven by the fact that the most performant and implemented relational database join is the equi-join [23]. Moreover we provide an implementation for conjunctive semantics, since this task is more prone to be optimized than the disjunctive one. Algorithm 1 for GCEA consists in three parts: *(i)* vertex partitioning (bucketing) through an hashing function (OperandPartitioning) *(ii)* graph serialization on secondary memory (SerializeOperand), and *(iii)* actual join algorithm over the graphs' buckets (PartitionHashJoin). Relational partition hash-join undergo the same phases, even if relational algorithms do not deal with outgoing edges (lines 31-35 and Section 6). We allow vertices with replicated values as in current graph databases implementations (such as Titan and Neo4J). Consequently *id*s enumerate the vertices within a single graph.

As a first step, the hashing function h is inferred from θ (line 2): if $\theta(u, v)$ is a binary predicate between distinct attributes from u and v, then h is defined as a linear combination of hash functions over the attributes of either u or v. When no h could be inferred from θ, then h is a constant function.

Algorithm 1 Graph Conjunctive EquiJoin Algorithm (GCEA)

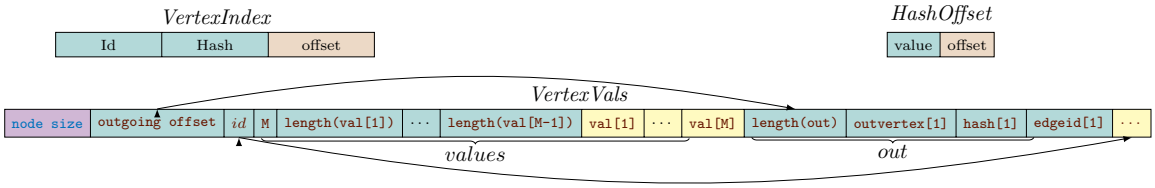
```
 1: procedure ConjunctiveJoin(G, G', θ)
 2:     hashFunction = generateHash(θ);
 3:     omap_1 = OperandPartitioning(G,hashFunction)
 4:     omap_2 = OperandPartitioning(G',hashFunction)
 5:     Ḡ_1 = SerializeOperand(G,omap_1)
 6:     Ḡ_2 = SerializeOperand(G',omap_2)
          return PartitionHashJoin(Ḡ_1, Ḡ_2, θ)
 7:
 8: procedure SerializeOperand(G,omap):
 9:     File VertexIndex = Open();
10:     VertexVals= Open(), HashOffset= Open();
11:     ulong offset = HashOffset = 0;
12:     for each h ∈Keys(omap) do        ▷ Ordered maps have ordered keys.
13:         HashOffset.Write({h,HashOffset});
14:         for each id ∈omap[h] do
15:             v = G.V[id];
16:             v.hash = h; v.offset = VertexVals;
17:             VertexIndex.Write({v.id, h, offset});
18:             ulong offsetNext = VA.Write(serialize(v));
19:             offset+=offsetNext; HashOffset+=offsetNext;
      return (VertexIndex,VertexVals,HashOffset,G.A_v,G.A_e)
20:
21: procedure PartitionHashJoin(G_1, G_2, θ):
22:     θ'(u, u') := θ(u, v) ∧ (u ⊕ u')(A_v) = u ∧ (u ⊕ u')(A'_v) = u';
23:     Θ'(e, e') := (e ⊕ e')(A_e) = e ∧ (e ⊕ e')(A'_e) = e'
24:     HI = IntersectHashes(HashOffset_1,HashOffset_2).iterator();
25:     File AdjFile = Open();
26:     while HI.hasNext() do
27:         h = HI.next();
28:         for each u ∈ VertexVals_1[h.offset_1], u' ∈ VertexVals_2[h.offset_2] do
29:             if θ'(u, u') then
30:                 AdjFile.Write(V={u ⊕ u'},)
31:                 HIout = IntersectHashes(out_V(u),out_V'(u')).iterator();
32:                 while HIout.hasNext() do
33:                     h_out = HIout.next();
34:                     for each edge e ∈ out_V(u)[h_out.offset_1], e' ∈
    out_V'(u')[h_out.offset_2] do
35:                         if θ'(e.outvertex, e'.outvertex) and Θ'(e, e') then
36:                             AdjFile.Write(E={e ⊕ e'})
```

OperandPartitioning performs a vertex bucketing in main memory: its outcome is an ordered map, where each vertex v is stored in a collection $omap[h(v)]$, where h is the aforementioned hashing function. For each operand G_i, the $omap_i$ construction takes at most $\sum_{j=0}^{|\mathcal{V}(i)|} \log(j)$ time, where $|\mathcal{V}(i)|$ is the multiset vertex size. Such time complexity is bounded by $|\mathcal{V}(i)| \leq \sum_{j=0}^{|\mathcal{V}(i)|} \log(j) < |\mathcal{V}(i)|^2$ where $|\mathcal{V}(i)| \gg 1$.

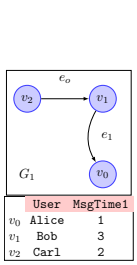
SerializeOperand stores the operand in secondary memory: both buckets (line 12) and vertices (line 14) are already sorted by hash value, and hence such data structures are accessed linearly. Figure 3c depicts a serialized representation of the graph in Figure 3b: all the labels and the edge values are not serialized but are still accessible through the original graph G via *id*. Buckets are represented by *HashOffset* providing both the bucket value and the pointer to the first vertex of the bucket stored in *VertexVals*. *VertexVals* stores vertices alongside with their adjacency list, where vertices are sorted by hash value and are represented by *id* and hash value. *VertexIndex* allows to find the vertices stored in *VertexVals* in constant time: each record is ordered by vertex *id*, has a constant size and contains the pointer to where the vertex data is stored in *VertexVals*. Even the outgoing edges are stored by the destination vertex's hash value. Given k_i the size of Keys(omap_i), this phase takes $3k_i + |G_i|$ time, where $2k_i$ is the omap visit cost, k_i is the omap serialization as *HashOffset* and $|G_i|$ is the time to serialize the graph as VA_i.

The last step performs the actual conjunctive join over the serialized graph (PartitionHashJoin): the data structure is accessed from secondary memory through memory mapping. Line 24 prepares the intersection: while performing a linear scan over the buckets, the HI iterator checks if both operands have a bucket with the same hash value (line 26), then the common hash value is extracted (line 27) and the two buckets accessed (line 28), then the composition $u \oplus u'$ between the vertices is performed (line 30). Next, differently from the relational join, the adjacent vertices for both operands are visited. Similarly to line 24, the hash-sorted edges induce a bucketing (line 31), and then we check if the destination vertices meet the join conditions alongside with the to-be-joined edges (line 35). Please note that, as stated out in Definition 5, edges are not filtered by θ predicate. Furthermore, the resulting graph is stored in a bulk graph where only the vertices *id* from the two graph operators appear as pairs. This last operation takes time $k_1 + k_2 + \sum_{h \in HI} \left(b_1^h \cdot b_2^h + out_1^h \cdot out_2^h\right)$ where b_i^h is the size of the h bucket for the i-th operand, while out_i^h is the outgoing vertices' size for all the vertices within the h bucket for the i-th operand.

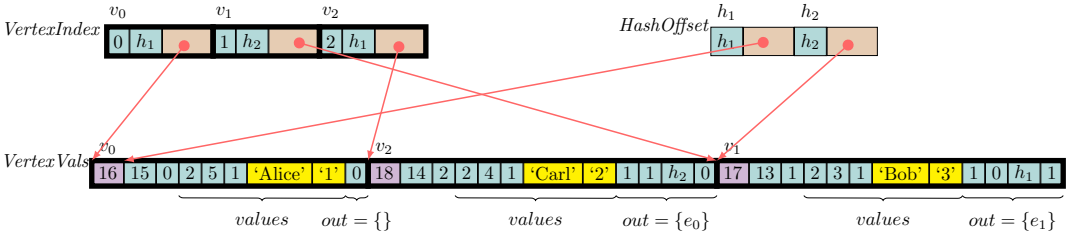
Such algorithm could be also extended to the disjunctive semantics as follows: all the edges discarded from the intersection in line 30 for $u \oplus u'$ should be considered, either if they come from the left operand or from the right one. Between all such edges, we consider only those e' that have a destination vertex ν which hash value appears in HI. Moreover it has to satisfy the binary predicate θ jointly with another vertex ν', coming from the opposite operand. Hence we establish (e.g.) an edge $(u \oplus u', \nu \oplus \nu')$ having the same values and attributes of e' and the same set of labels.



(a) Data structures used to implement the graph in secondary memory. Each data structure represents a different file.



(b) G_1



(c) Using the graph schema in Figure 3a for representing G_1 in secondary memory. v_0 and v_2 belong to a different bucket from v_1 only for illustrative purposes.

Figure 3: *Graph representation in secondary memory.*

5. EXPERIMENTAL EVALUATION

Through the following experiments we want to prove that *(i)* both hash buckets and memory mapping for the graph join operands provide better results for GCEA, *(ii)* which outperforms the query plans for other query languages (both graph and relational). For the first case we have to use graph libraries or graph databases where transactions and logging can be disabled, while for the second we choose state of the art graph databases implementing specific query languages.

In order to do so we choose the simplest graph representation that provides better performances for all the addressed languages: we choose a graph where only vertices contain values and where labels are stored in both vertices and edges. We created our data using the LiveJournal Graph [19] containing 4,847,571 unlabelled vertices and 68,993,773 unlabelled edges. Each vertex represents a user which is connected to each of its friends by an edge. Since no data values are given within the datasets, we enriched the graph using the guidelines of the LDBC Social Network Benchmark protocol [10], and hence associated to each user an IP address, an Organization and the year of employment[1]. For each experiment, the input data were obtained by starting a random walk from the same vertex but using a different seed for the graph traversal. New data sets were obtained incrementally by visiting each time a number of vertices that is a power of 10, from 10 to 10^6.

We performed our tests over a MacOsX with a 2.2 GHz Intel Core i7 processor and 16 GB of RAM at 1600 MHz, and an SSD Secondary Storage with an HFS file system. We evaluate the graph join using as operands two distinct sampled subgraphs with the same vertex size ($|V|$), where the θ predicate is the following one: $\theta(u,v) \stackrel{def}{=} u.Year1 = v.Year2 \wedge u.Organization1 = v.Organization2$. Such predicate does not perform a perfect 1-to-1 match with the graph vertices, thus allowing to test the algorithm with different multiplicities values. We tested the algorithm with the conjunctive semantics, having a subset of the operations of the disjunctive one.

[1]More informations regarding our proposed solutions are available at http://smartdata.cs.unibo.it/?page_id=798.

5.1 Evaluating Data Structures

We benchmark our solution with graph data models where database transactions either do not exist or can be disabled. We first consider two graph libraries accessing graphs in main memory; we tested the Boost Graph Library 1.60.0 with the most efficient configuration for graph traversals tasks, *vec* [24], and Snap 3.0 [20] considering the attributes only over the vertices (`TNodeNet<TAttr>`). Then we consider the Sparksee* graph database [9]: transactions were disabled in the configuration file, as well as logging, rollback and recovery facilities. Concerning the graph database management implementation, no assumptions can be made as it is closed source.

We implemented our graph join algorithm for all the aforementioned libraries. We used the standard graph library methods to store the graph in secondary memory (serialization or graph database storage) and extended the PartitionHashJoin by doing a preliminary vertex bucketing phase: buckets are not supported and vertices cannot be sorted by hash value.

Join Evaluation Time. In this case we evaluate two aspects: *(i)* the join algorithm running time and *(ii)* the time required to create the solution and store it in secondary memory.

Table 4a provides the cost of performing the sole join algorithm excluding the result storing time. All the competitors' graphs were joined through GCEA and vertices with the same hash were put in the same bucket in main memory. It must be emphasised that both Boost and SNAP operands were loaded in primary memory, while our operands were accessed in secondary memory through memory mapping. The table shows how all the other data structures had a worse performance due to the initial cost of the bucket creation and sorting. We must also remark that this result justifies the need of our data structure for the proposed algorithm.

Operands Size		GCEA running time, result creation excluded				GCEA result creation time			
Left ($\|V\|$)	Right ($\|V\|$)	Proposed	Boost	SNAP	Sparksee	Proposed	Boost	SNAP	Sparksee
10	10	0.19 ms	**0.09**×	0.23×	9.42×	**0.0010** ms	17.00×	36.40×	738.33×
100	100	0.18 ms	**0.85**×	1.72×	24.96×	**0.0023** ms	5.39×	17.04×	290.14×
1 000	1 000	**0.31** ms	5.68×	14.93×	88.42×	**0.0036** ms	7.72×	14.67×	215.65×
10 000	10 000	**1.90** ms	11.13×	26.83×	156.42×	**0.3706** ms	4.60×	7.61×	15.67×
100 000	100 000	**32.31** ms	8.73×	19.33×	81.05×	**39.3428** ms	4.20×	5.80×	11.70×
1 000 000	1 000 000	**332.60** ms	15.42×	33.15×	171.54×	**3 207.8738** ms	5.76×	12.29×	15.50×

(a) Performing GCEA when all operands are already loaded.

Size ($\|V\|$)	Proposed	Boost	SNAP	Sparksee
10	0.23 ms	**0.68**×	0.98×	7.73×
100	**0.50** ms	1.60×	5.22×	11.76×
1 000	**3.38** ms	1.68×	6.94×	13.47×
10 000	**34.26** ms	1.52×	7.25×	13.84×
100 000	**355.96** ms	1.47×	6.27×	14.73×
1 000 000	**3 518.47** ms	1.89×	6.10×	17.79×

(b) Graph operand creation+storing time.

Figure 4: Benchmarking results for the LiveJournal database over C++ graph libraries and low level databases.

Operands Size		Result		Join Time (C/C++) (ms)			Join Time (Java) (ms)	
Left ($\|V\|$)	Right ($\|V\|$)	Size ($\|V\|$)	Size ($\|E\|$)	Virtuoso	PostgreSQL	GCEA (C++)	Neo4J	GCEA (Java)
10	10	5	2	4.99	11.29	0.53	211.45	24.97
10^2	10^2	16	4	4.94	22.82	0.93	222.87	32.70
10^3	10^3	251	55	4.55	22.92	4.35	448.97	117.58
10^4	10^4	2 734	680	117 712.00	183.90	40.42	3 149.90	1 150.37
10^5	10^5	26 803	7 368	>4H	7 150.74	411.78	241 026.79	17 178.49
10^6	10^6	151 212	99 558	>4H	99 683.91	3 966.72	>4H	178 066.80

Figure 5: Graph Join Running Time. Each data management system is grouped by its graph query language implementation.

The same table provides the time required to store the results as an adjacent list in secondary memory using the default graph library representation (non-labelled vertices and edges, default serialization). In this case our solution always outperforms the other graph libraries and databases.

Operand creation time. We consider the graph creation time in main memory and the cost of storing it in secondary memory per operand (Table 4b). For both Boost and SNAP the default serialization methods are performed, while for Sparksee* we simply closed the database. In this case our solution outperforms all the competitors.

5.2 Join Execution Time

This last experiment compares the interpretation of query plans for both relational and graph databases with GCEA. The opponents' query plans are discussed in Section 6.1.

We used default configurations for both Neo4J and PostgreSQL, while we changed the cache buffers configurations for Virtuoso (as suggested in the configuration file) for 16GB of RAM. We kept the default multithreaded query execution plan. Cypher queries were sent using the Java API but the graph join operation was performed only in Cypher through the `execute` method of an `GraphDatabaseService` object. PostgreSQL queries were evaluated directly through the `psql` client and benchmarked using both `explain analyze` and `\timing` commands. Virtuoso was benchmarked through iODBC connection evoked in C using Redland RDF library: no HTTP connections were used and only the `librdf_model_query_execute` function was involved in the graph join operation. Neo4J graphs were fine tuned by indexing the attributes `Organization` and `Year` involved in the query and, since Cypher language does not allow to access to different graphs, both graph join operands were stored within the same graph. Virtuoso triples were not indexed, as a default set of indices are defined during the graph creation, and data is automatically indexed. All the aforementioned conditions do not degrade the query evaluations.

Table 5 represents the result of such benchmarks. The competitors' join time is made up only by the query evaluation time, while our proposed implementation considered the whole GCEA algorithm, and hence both the partitioning phase, the operands' serialization and the actual join execution were considered. As a result our solutions always outperform the competitors' query plans within their own language implementation.

6. RELATED WORK

At the time of writing, the only field where a binary graph operation is discussed is **Discrete Mathematics**. Such operations are defined over either finite graphs or finite graphs with cycles, and are named *graph products* [14]. Every graph product produces a graph whose vertex set is defined as a cartesian product between the vertices' sets producing pair of vertices, while the edge set changes accordingly to the different graph product definition. Consequently the Kroneker Graph Product [26] is defined as follows:

$$G \times G' = (V \times V', \{\ ((g,h),(g',h')) \in V \times V' \mid (g,g') \in E, (h,h') \in E'\ \})$$

while the *cartesian graph product* [18] is defined as follows:

$$G \square G' = (V \times V', \{\ ((g,h),(g',h')) \in V \times V' \mid (g = g', (h,h') \in E') \vee (h = h', (g,g') \in E)\ \})$$

Other graph products are *lexicographic product* and *strong product* [14, 17]. This definition has two issues: *(i)* the resulting vertex set is not made of single vertices but of pair of vertices, and hence *(ii)* those graph product definitions are

not tailored for graphs with embedded data (i.e. property graphs, triple stores).

6.1 (Graph) Query Languages

In this section we describe how a graph join is implemented for property graph databases and RDF triplestores. The reason is twofold: we want both to show that graph joins can be represented in different data representations, and to detail how our experiments in Section 5 were performed.

Property Graph and Cypher. Property graphs are, to the best of our knowledge, the more general graph data model representation because they consider both vertices and edges as multi-labelled tuples. It is necessary to compare the performances of our algorithm with query languages running on top of property graph databases because our physical model generalizes property graphs. Among the Property Graph databases we do not consider SQLGraph [25] because there is no existing implementation and, most importantly, the Gremlin query language allows only to perform graph traversal queries returning bag of values. We choose to perform our tests over Neo4J using Cypher as a query language, because Neo4J allows to extend the built-in query plans with ad hoc solutions [16], eventually allowing an implementation of our algorithm in a future.

Cypher uses a pipe query evaluation model allowing to refine queries in further steps. Regarding the implementation of the graph conjunctive join operator in Cypher, *ValueHashJoins* are performed between vertices coming from different graph operands, and hash values are either evaluated at run time, or depend on attributes' values indexings. This choice supports the experimental evidence of Cypher having a better scalability than SPARQL, where RDF graphs cannot be indexed by values (see the next paragraph). Once the Cypher query is transformed into a pipe-based query plan, most of the pipes' sources appear to be *NodeByLabelScan* and *AllNodeScan*: this means that all the graph's vertices (with a given label) are considered in the first steps of computation. As a result the query plan scans more data than it should to provide the final result. In our algorithm this drawback does not occur because we directly access the data per buckets on both graph operands, avoiding to consider any vertices' combinations that will not appear in the final result.

RDF triplestores and SPARQL. Triplestore systems store the graph informations as triplets, $(source, property, destination)$, where *source* and *destination* are two vertices, and *property* is the edge linking them. Such *property* could even appear as a source vertex whenever additional information is provided [8]. [8] shows that property graphs can be entirely mapped into RDF triplestore systems as follows:

Definition 6 (Property Graph over Triplestore). *Given a property graph $G = (V, E, A_v, A_e)$, each vertex $v_i \in V$ induces a set of triples (v_i, α, β) for each $\alpha \in A_v$ such that $v_i(\alpha) = \beta$ having $\beta \neq$ `NULL`. Each edge $e_j \in E$ induces a set of triples (s, e_j, d) such that $\lambda(e_j) = (s, d)$ and another set of triples (e_j, α', β') for each $\alpha' \in A_e$ such that $e_j(\alpha') = \beta'$ having $\beta' \neq$ `NULL`. Each property graph G is stored as a distinct named graph.*

This allows us to query each property graph with SPARQL query language, specifically targeted for triplestores, through their RDF representation. We took this query language into account for our benchmarks because it is well consolidated: a lot of research has been carried out [22] and efficient query plans have been implemented [15], even when multiple graphs are took in input. These results involve the interpretation and the execution of "optional joins" paths [2], thus allowing to check whether the graph conjunctive join conditions are not met for the outgoing edges. Such performances quickly degrade due to both the sparsity of the data representation requiring to perform more *path joins* than the ones required for the property graph model, and to the `CONSTRUCT` clauses are not included in the SPARQL algebra optimizations. However, `CONSTRUCT` is required for produce a graph as a final outcome of our graph join query. Moreover RDF triplestores as Virtuoso prefer to index triplets per patterns and do not allow triplet indexing by values. Within our tests, we also took into account that both input and output met the requirements of Definition 6.

7. CONCLUSIONS

This paper defines for the first time a graph join operator. A graph algorithm is proposed for the conjunctive semantics, outperforming the implementations on different languages. By comparing the execution of our algorithm with our graph data structure with other ones provided by graph libraries, we also show that our choice of sorting the vertices per hash value (required for the partition hash join) allows to have better performances during the overall execution of the join algorithm.

Some results have been omitted for lack of space. First, we could prove that our operator is both commutative and associative. This result could be proved both for the vertices' and edges' tuples, and for their labels. Such are relevant properties when such operators are used for data integration tasks. The bucketing approach allows even to implement the graph join through a map-reduce approach. Last, we could extend our algorithm to support $\leq$ predicates in θ over one attribute having partially ordered values [6]: since our data structure is already ordered by hash value, we could just use a monotone hashing function h w.r.t such attribute.

8. REFERENCES

[1] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 431–446, New York, NY, USA, 2016. ACM.

[2] M. Atre. Left Bit Right: For SPARQL Join Queries with OPTIONAL Patterns (Left-outer-joins). In *SIGMOD Conference*, pages 1793–1808. ACM, 2015.

[3] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "bit" loaded: A scalable lightweight join query processor for rdf data. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 41–50, New York, NY, USA, 2010. ACM.

[4] P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Database Systems - Concepts, Languages and Architectures.* McGraw-Hill, 1 edition, 1999.

[5] P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Basi di dati. Modelli e linguaggi di interrogazione.* McGraw-Hill, Milan, 3 edition, 2009.

[6] G. Bergami, M. Magnani, and D. Montesi. On joining graphs. *CoRR*, abs/1608.05594, 2016.

[7] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 121–132, New York, NY, USA, 2013. ACM.

[8] S. Das, J. Srinivasan, M. Perry, E. I. Chong, and J. Banerjee. A tale of two graphs: Property graphs as RDF in oracle. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pages 762–773, 2014.

[9] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey. Survey of graph database performance on the hpc scalable graph analysis benchmark. In *Proceedings of the 2010 International Conference on Web-age Information Management*, WAIM'10, pages 37–48, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 619–630, New York, NY, USA, 2015. ACM.

[11] O. Erling and I. Mikhailov. Virtuoso: Rdf support in a native rdbms. In R. D. Virgilio, F. Giunchiglia, and L. Tanca, editors, *Semantic Web Information Management*, pages 501–519. Springer, 2009.

[12] G. H. Fletcher and P. W. Beck. Scalable indexing of rdf graphs for efficient join processing. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pages 1513–1516, New York, NY, USA, 2009. ACM.

[13] J. Gao, J. Yu, H. Qiu, X. Jiang, T. Wang, and D. Yang. Holistic top-k simple shortest path join in graphs. *IEEE Trans. on Knowl. and Data Eng.*, 24(4):665–677, Apr. 2012.

[14] R. Hammack, W. Imrich, and S. Klavzar. *Handbook of Product Graphs, Second Edition*. CRC Press, Inc., Boca Raton, FL, USA, 2nd edition, 2011.

[15] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.

[16] J. Hölsch and M. Grossniklaus. An algebra and equivalences to transform graph patterns in neo4j. *Fifth International Workshop on Querying Graph Structured Data*, 2016.

[17] W. Imrich and S. Klavzar. *Product Graphs. Structure and Recognition*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2000.

[18] W. Imrich and I. Peterin. Recognizing cartesian products in linear time. *Discrete Mathematics*, 307(3-5):472–483, 2007.

[19] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[20] J. Leskovec and R. Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[21] M. Paradies, W. Lehner, and C. Bornhövd. Graphite: An extensible graph traversal framework for relational database management systems. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, SSDBM '15, pages 29:1–29:12, New York, NY, USA, 2015. ACM.

[22] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, Sept. 2009.

[23] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1961–1976, 2016.

[24] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[25] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1887–1901, New York, NY, USA, 2015. ACM.

[26] P. M. Weichsel. The kronecker product of graphs. *Proceedings of the American Mathematical Society*, 13(1):47–52, 1962.

[27] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: A fast and compact system for large scale rdf data. *Proc. VLDB Endow.*, 6(7):517–528, May 2013.

APPENDIX

DEFINITION 7 (CONCATENATION). *$\oplus : A \times A \mapsto A$ is a lazy evaluated* ***concatenation*** *function between two operands of type A returning an element of the same type, A. The concatenation function is a linear function such that, given any function H with $dom(H) = A$, $H(u \oplus v) = H(u) \oplus H(v)$. $\oplus$ is defined for the following A-s:*

- ***sets****: it performs the union of the two sets:* $S \oplus S' \stackrel{def}{=} S \cup S'$
- ***integers****: it returns the dovetail number associating to each pair of integers an unique integer:* $i \oplus j \stackrel{def}{=} \sum_{k=0}^{i+j} k + i$
- ***functions****: given a function $f : A \mapsto B$ and $g : C \mapsto D$, $f \oplus g$ is the function returning $f(x)$ if $x \in dom(A)$, and $g(x)$ if $x \in dom(C)$.* `NULL` *is returned otherwise. Such function concatenation are used when $\forall x \in A \cap C. f(x) = g(x)$.*
- ***pairs****: given two pairs (u, v) and (u', v'), then the pair concatenation is defined as the pairwise concatenation of each element, that is $(u, v) \oplus (u', v') \stackrel{def}{=} (u \oplus u', v \oplus v')$. Elements belonging to multisets are represented as pairs of elements and integers, and hence $s_i \oplus t_j \stackrel{def}{=} (s \oplus t)_{i \oplus j}$.*