Out-of-Core Property Graph Matching for Real-World Graph Databases

Yanxuan Cui
Tsinghua University
Being, China
cuiyx18@mails.tsinghua.edu.cn

ABSTRACT

PVLDB Reference Format:
Yanxuan Cui. Out-of-Core Property Graph Matching for Real-World Graph Databases. PVLDB, 14(1): XXX-XXX, 2021.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at http://vldb.org/pvldb/format_vol14.html.

1 INTRODUCTION

2 BACKGROUND

This section introduces the formal definition of property graphs, and then discusses the property graph matching problem.

2.1 Property Graph Model

A *property graph* is a directed vertex-labeled edge-labeled multi-graph with self-edges, and key-value properties are stored on vertices and edges. We now provide the formal definition of a property graph.

Definition 1 (Property Graph [1]). *A property graph G is a tuple $(V, E, \rho, \lambda, \sigma)$, where:*

1. *V is a finite set of vertices.*
2. *E is a finite set of edges such that V and E have no elements in common.*
3. *$\rho : E \to (V \times V)$ is a total function. Intuitively, $\rho(e) = (v_1, v_2)$ indicates that e is a directed edge from v_1 to v_2.*
4. *$\lambda : (V \cup E) \to L$ is a total function where L is a set of labels. Intuitively, if $v \in V, \rho(v) = l$ (respectively, $e \in E, \rho(e) = l$), then l is the label of vertex v (respectively, edge e).*
5. *$\sigma : (V \cup E) \times Prop \to Val$ is a partial function with $Prop$ a finite set of properties and Val a set of values. Intuitively, if $v \in V, p \in Prop, \sigma(v, p) = s$ (respectively, $e \in E, p \in Prop, \sigma(e, p) = s$), then s is the value of property p for vertex v (respectively, edge e) in the property graph G.*

For simplicity, in this paper, we do not discuss the properties i.e., σ in G, because similar techniques can be used as processing the labels λ. Thus, the property graph G can be denoted by $(V(G), E(G), \rho_G, \lambda_G)$. Please note that the total function ρ_G is necessary, in general, we cannot identify an edge simply by the starting and ending vertices such as (u_1, u_2) as can be done in the simple graph model, because multiple edges may appear between the two vertices. However, we may use the (u_1, u_2) notation if all we care about is that there exist at least one edge between u_1 and u_2.

Definition 2 (Vertex Cover). *A vertex cover V_c of a property graph G is a subset of $V(G)$ such that $\forall e \in E(G), \rho_G(e) = (u, v) \implies u \in V_C \vee v \in V_C$.*

2.2 Property Graph Matching Problem

Definition 3 (Subgraph). *A property graph F is called a subgraph of a property graph G, written $F \subseteq G$, if $V(F) \subseteq V(G)$, $E(F) \subseteq E(G)$, ρ_F is a restriction of ρ_G, and λ_F is a restriction of λ_G.*

Let G be any property graph, and let $S \subseteq V(G)$, then the *induced subgraph* $G[S]$ is the graph whose vertex set is S and whose edge set consists of all of the edges in $E(G)$ that have both endpoints in S.

Definition 4 (Property Graph Isomorphism). *Two property graphs G and H are isomorphic, written $G \cong H$, if there exists bijections $\theta : V(G) \to V(H)$ and $\phi : E(G) \to E(H)$ such that $\rho_G(e) = (u, v)$ if and only if $\rho_H(\phi(e)) = (\theta(u), \theta(v))$, $\lambda_G(v) = \lambda_H(\theta(v))$ for all $v \in V(G)$ and $\lambda_G(e) = \lambda_H(\phi(e))$ for all $e \in E(G)$; Such a pair of mappings is called an isomorphism between G and H.*

The bijection $\theta : V(G) \to V(H)$ is the key in the definition of property graph isomorphism, because the bijection $\phi : E(G) \to E(H)$ is straightforward if θ is fixed. However, due to automorphism, where an *automorphism* of a graph is an isomorphism of the graph to itself, the bijection θ may not be unique.

Definition 5 (Property Graph Matching). *Given a data property graph D, a pattern property graph P and a searching condition $\psi : PG \to B$ with PG the set of property graph and B the set of Boolean values, the property graph matching problem is to report the set $\mathcal{I} = \{F | F \subseteq D, F \cong P, \psi(F) = true\}$.*

Authors of previous works usually omit the searching condition ψ in their definition of graph matching [10, 13, 17, 21]. And they adopt a loosely related technique called *symmetry-breaking* [5], which ensures there is a unique bijection $\theta : V(P) \to V(F)$ by providing a partial order on $V(P)$ after exploiting the automorphism of P. However, as we have stated before, the WHERE clause is a ubiquitous part of the query language of a graph database. Users of a real-world graph database usually provide their self-defined searching condition ψ to filter out unnecessary matchings not only symmetry-breaking conditions. Thus, the searching condition we

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150–8097.
doi:XX.XX/XXX.XX

defined here can be viewed as a super set of symmetry-breaking. We add the searching condition in the definition because it is actually a part of the property graph matching problem, and we also found that it can be decomposed and pushed down to lower phase to boost the evaluation of graph matching (Section 3).

A property graph is always directed. However, in some cases such as friendship, there is no need to pay attention to the directions of the edges. In order to support this kind of relationship, a naive approach is to add a duplicate edge in opposite direction for each edge in the data graph. More elegantly, we allow the pattern graph P to contain undirected edges. Users can simply ignore the direction by providing undirected edges in P like in industrial graph databases such as Neo4j.

3 PROPERTY GRAPH MATCHING FRAMEWORK

4 AN I/O EFFICIENT VERTEX-CENTRIC STORAGE METHOD FOR PROPERTY GRAPH

The random access problem is a well known hard problem for out-of-core systems, especially for the graph related problem, which is notorious for its poor locality [12]. The traditional way to store graphs on disk is the double list method: Stores the in-edges and out-edges separately for each vertex, via the compressed sparse column (CSC) and the compressed sparse row (CSR) format [16]. However, we find that this storage method have limitations to solve the real-world property graph matching problem: *Random disk accesses are unavoidable when in/out-edges are stored separately.*

To solve the random disk access problem, in this section, we propose a novel property graph storage engine from a vertex-centric point of view. Noticeably, only part of the huge billion node data graph would be read when solving a property graph matching problem, with the help of a few easy to implement indices. And all the disk reads are sequential. Moreover, by using the property graph matching engine that will be discussed in the next section, we would be sure that the huge data graph would be read at most once.

4.1 Scan the Data Graph Sequentially

The property graph matching problem requires the complete connection information between vertices, however, the conventional double list storage method break up the completeness and results in random disk accesses.

Consider the simple patten graph in Figure 1, which simply finds friendship pairs in a data graph. If two vertices in the data graph, say v_i and v_j, are supposed to match the pattern, we must be sure that v_i and v_j follows each other at the same time. This kind of neighbor checking is also the most essentially building block of a property graph matching engine.

In the data graph, v_1 has m in-edges and n out-edges, and suppose that they are stored separately in the traditional way. In order to determine whether v_1 could match u_1, one has to scan the in-edges (or equally, the out-edges) of v_1 and then check whether the visited neighbor is in the out-edges (or in-edges) list. This scan and check method would significantly slow down the graph matching process, because the checking process results in random disk read. For real world power-law graphs, where the celebrities or trending topics have a huge amount of followers, the in/out-edges lists stored on disk have to be swapped in and out frequently during the scan and check process as a result of the random disk access pattern. And it would be more complicated if there are more than two edges between u_1 and u_2 in the pattern graph.

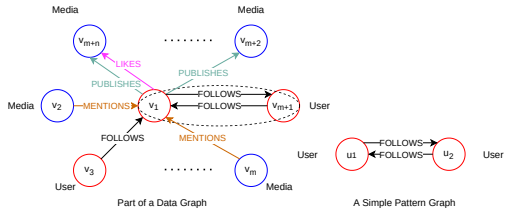

Figure 1: Part of a social network data graph where the center v_1 is a user, and a simple pattern graph.

The source of the problem is that the in/out-edges are stored separately in the conventional method, whereas the property matching process need to retrieve both in- and out- connection information to determine whether a neighbor vertex could be matched. Based on this observation, we propose a vertex-centric property graph storage method that always keeps the necessary connection information together for graph matching. Take Figure 2 as an example, which shows the logical storage structure for the celebrity in Figure 1. Real-world property graphs are directed labeled graphs, instead of splitting the edges into in/out-edges, we treat all the neighbors equally and store the connection information together with the neighbor vertex. The specific edge information (direction, types) could be obtained by the position of the stored edge labels, i.e., the upper labels means the direction is pointed to the root and the lower labels means the opposite. For example, the ":FOLLOWS" at the upper right corner of v_3 means v_3 follows v_1, the two ":FOLLOWS" besides v_{m+1} indicate that v_1 and v_{m+1} follow each other. And multiple edges are ubiquitous in property graphs, we support that by store the edge labels in a sequence, as is shown in the figure where v_1 publishes a social media v_{m+n} and also likes it.

By storing the neighbors in our vertex-centric approach, all the necessary information are now stored locally together with the vertices, and the neighborhood checking process of a property graph matching engine could be accomplished efficiently within a sequential disk scan. For the simple pattern in Figure 1, if we suppose that v_1 matches u_1 and want to check whether the neighbors of v_1 would match u_2, we could just scan the neighbors and check the labels sequentially and find that v_{m+1} would match. No random disk access appears during the whole process, and there is no need to use complex indices to check the edge labels.

4.2 Simple Indices to Reduce I/Os

Despite of the fact that the size of the whole data graph could be gigantic, we may only care a fraction of the graph by specify the labels in our query for a concrete graph matching problem. There are two basic operations for a graph database to solve a graph matching problem: 1. Given a data vertex, retrieve the neighbors

of it with a specific label; 2. Given a vertex label, retrieve the data vertices with the label. In the following paragraphs, we'll show how to make the two operations efficiently by adding a few simple but efficient indices to the vertex-centric storage engine, which reduces the searching space and I/O significantly.

Consider again the pattern graph in Figure 1, where we only care about the users in the social network, no matter how much social media a user have published or viewed we could just ignore all of them in our specific problem. A straightforward idea is to group the neighbor vertices with the same label together when storing the vertices. However, if the neighbor vertices were stored in the traditional in/out-edge double lists method, we have to group them twice and then still face the information insufficient problem as we discussed in the previous subsection. As for our vertex-centric storage method, the index could be added to the storage engine easily as is shown in Figure 2, where we simply store the key-value pairs that maps the vertex labels to the starting/ending position of the neighbors on disk. Since it is used to locate only the neighbors of a specific vertex, we refer to this kind of index as *local index*. With the help of local index, we could skip all the useless neighbor vertices and only scan the necessary ones.

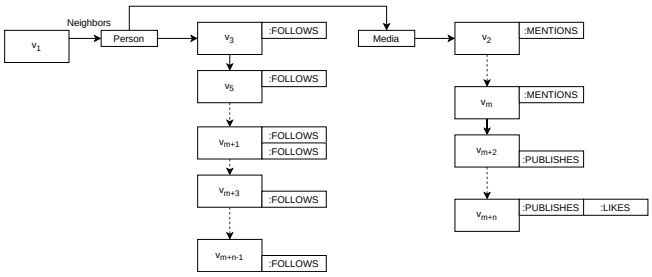

Figure 2: The vertex-centric storage structure of the celebrity v_1 in Figure 1.

From a higher perspective, in order to solve a property graph matching problem, one need to firstly select a starting vertex in the data graph, regardless of the concrete graph matching algorithm whether it is tree based or join based. There is no need to scan all the vertices in a billion node data graph if we only care about vertices with the specific labels. Just like the local index we discussed above, we could add a global index which contains key-value pairs that maps the labels to the corresponding position on disk (Figure 3).

In summery, for a property graph matching problem, we could quickly jump to the domain of interest with the help of the global index, and then scan and check only the necessary neighbors with the help of our local index. After jump to the correct position, all the disk reads are sequential.

4.3 Remarks on the Implementation of the Storage Method

For applications that need to perform the two basic operations (visiting vertices and visiting the neighbors), we define two iterators as the interface to visit the data graph:

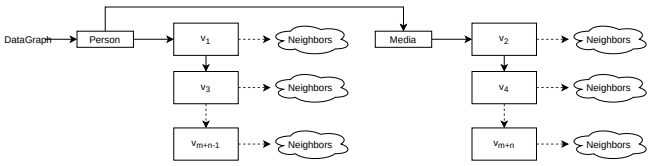

Figure 3: The storage structure overview of the data graph in Figure 1.

(1) VertexIter: Given a vertex label, it iterates through the vertices with the specified label ordered by the ID of vertices;
(2) NeighborIter: Given a data vertex and a label, it visits the neighbors with the label, the neighbors are also sorted by the IDs.

In Section 5 we'll show that the sorting constraint could boost the matching of property graphs.

These iterators could be implemented efficiently by using our vertex-centric storage model. And we also provide a compact disk format implementing the vertex-centric storage model in Figure 4. Vertices are sorted by their IDs, and we store in/out-degrees as early filters when scanning the vertices. Edge labels are stored as integers here, however, bitmap could also be used for higher performance. The VertexIter just searches the global index and then visits the disk data sequentially; The NeighborIter scans the neighbors with the help of the local index.

Please note that the vertex-centric storage model is not restricted to this disk format. In fact, as long as a storage engine could implement the two iterators efficiently, it could implement the vertex-centric storage model well. For example, the sorted vertices could be replaced with B-tree to make the insertion/deletion operation easier for dynamic graphs. It is also possible to implement the vertex-centric storage model in memory as buffer cache for existing graph databases to achieve better locality. Moreover, we are now working on implementing the vertex-centric storage model on top of relational database to embrace the power of the half-century-year-old mature technology.

5 A PRACTICAL PROPERTY GRAPH MATCHINIG ENGINE

Real world billion-node property graph can easily eat up hundreds of gigabytes, apart from that, even more spaces are required to store the intermediate results, which makes it financially impossible to solve the realistic property graph matching problem totally in memory. However, challenges have to be faced when developing an out-of-core property graph matching engine, because of the infamous random disk access problem.

In this section, we present a very interesting contribution based on the vertex-centric storage model that we discussed before. We avoid random disk accesses when scanning the huge data graph file, and avoid the intermediate results explosion problem by adopting an impressive compression algorithm. Moreover, we designed an efficient pipeline join method for compressed data, and developed a series of optimizations for real world property graph matching problems.

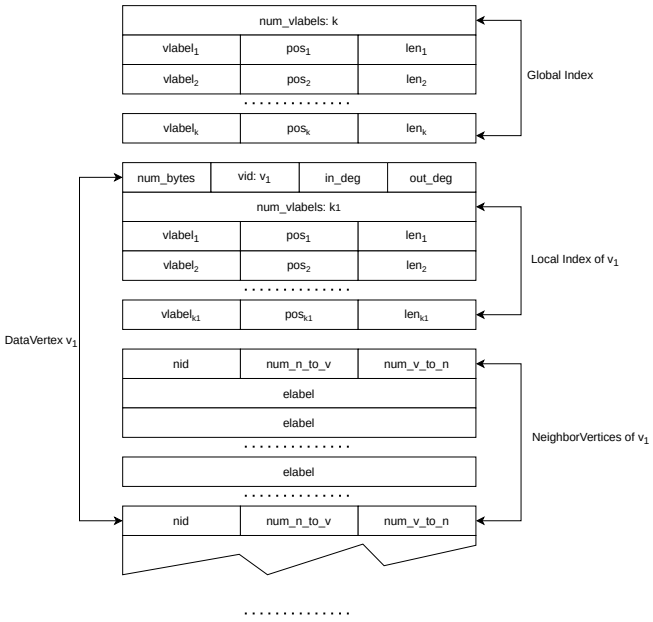

Figure 4: An I/O efficient property graph storage method that can boost the graph matching process.

5.1 Using Stars to Sequentially Scan the Huge Graph at Most Once

Because of the intrinsic poor locality of graphs, tree-based algorithms would incur incredible random disk accesses when jumping between the vertices scattered among the disk. Therefore, a join based matching algorithm is more suitable for real world problems. However, to *choose a proper join unit that can avoid random disk accesses and minimize the intermediate results as well* is still a hard problem. Perhaps the most intuitive way is to decompose the original pattern graph into a series of edges. However, lots of useless intermediate results would be generated by doing so. Consider the diamond pattern graph in Figure 5, which is a ubiquitous pattern in recommendation systems [6]. There are 8 edges in this pattern graph. Many intermediate results would be generated if they were matched in Figure 1, however, they are all pointless since there is no such a graph that could match the original diamond pattern. To solve this problem, more complex structures such as frequent subgraphs, multi-hop edges could be used, however, as Sun et al. have stated before, these methods require complex index that has super-linear space and time complexity [22], and are not very suitable for solving real world property graph matching problems.

Recall the vertex-centric property graph storage model that we discussed previously, which provides two efficient iterators that can scan the vertices (via VertexIter) and neighbors (via NeighborIter) sequentially (Section 4.3), if the matching process only requires neighborhood link information, random disk accesses could then be avoided. Based on this observation, we make a balance by using stars as our basic matching unit. As is shown in Figure 5, a star graph contains a root vertex and some neighbors connected to the root. The star pattern can then be matched within a sequential disk scan based on our vertex-centric storage model: 1. Select the domain of interest using the global index, 2. iterate through the relevant vertices by the VertexIter, 3. and for each visited vertex, use the NeighborIter to check the neighbors to determine whether the star would be matched. Besides, a star contains far more structural information then an edge, which means the matching results of a star have a predictable smaller size. Moreover, we made further contributions to keep the matching results even smaller (Section 5.2 and Section 5.4).

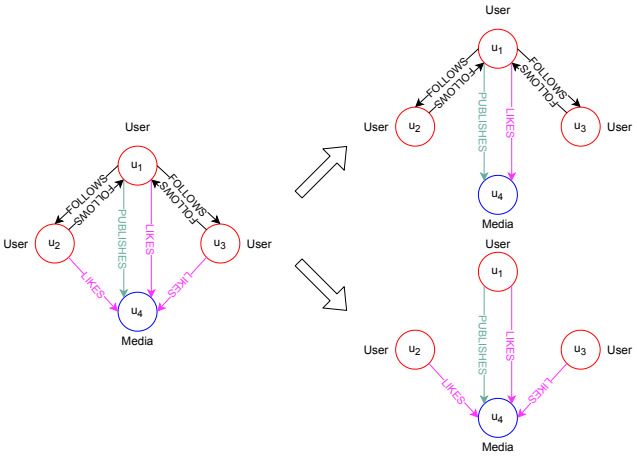

Figure 5: A pattern for recommendation system of social network and the decomposition of stars.

Some authors also adopt star-like structures as their basic matching unit [13, 22], however, our contribution takes further steps in three different ways:

(1) In order to make a practical graph matching engine, we adopt the property graph model (Section 2) rather than the simple graph model ubiquitous among academical paper. A simple graph can be viewed as a special case of a property graph, which ignores the labels, multi-edges, or even the direction of edges. However, real world applications of simple graphs are very limited because of the information they dropped out. It is not easy to make a simple graph matching algorithm to solve the property graph matching problem. For one thing, it is a hard engineering problem, because the traditional underlying graph storage method is not suitable for property graphs (Section 4). For another, many existing work rely on the perfect isotropic properties of a simple graph to operate and optimize their algorithms [7, 17, 22].

(2) We developed a novel pattern decomposition algorithm that is able to preserve as much matching information as possible, whereas the existing decomposition methods may lose information and result in gigantic useless intermediate results. Consider the pattern graph in Figure 6, suppose that u_1, u_2 and u_3 are selected as the roots, existing decomposition method would result in three stars with 3, 2 and 1 neighbor(s) by consecutively selecting and removing vertices from the original pattern. However, many useful matching information are lost by doing so, e.g., the third "star" is just an edge, which would generate enormous unnecessary matching results whereas every edge in the data

graph would match it but only a part of them could match the original pattern graph. In contrast, our approach (Algorithm 1) would keep all the neighborhood matching information as is shown in the bottom of Figure 6, which could then reduce unnecessary intermediate results significantly (Section 6). Like previous work [22], we also use a heuristic function to select a join order, which is defined as $f(u) = \frac{\deg(u)+|\psi(u)|}{\text{freq}(u.label)}$, i.e., we prefer to select vertex with bigger degree (more early filters) and less label frequency (smaller intermediate matching results) first. $|\psi(u)|$ is the number of local constraints of u, which will be discussed further in Section 5.4. The root candidate set R is used to select a connected vertex cover, which could then be joined efficiently (Section 5.3). The key feature of the algorithm is to remove selected vertex in a copy p' of the pattern and always keeps the original useful information in p, and thus, the intermediate results of our star could be much smaller.

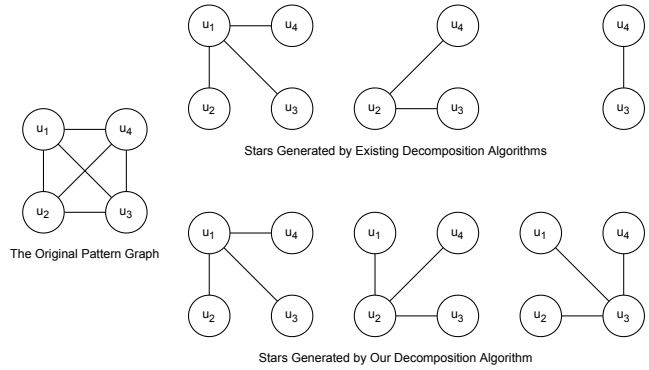

Figure 6: Stars decomposed from the same pattern graph using different algorithms.

(3) As the real-world billion node graphs are so large that it is preferable to scan it only once in a streaming style when solving a property graph matching problem. However, it is not a simple task because multiple stars has to be matched in a single sequential scan, the context switch cost and the intermediate result write cost must be minimized. For the context switch cost, it is strongly coupled with the underlying storage method of the data graph. Thanks to the elegant design of our vertex-centric storage model, which is able to match stars in a sequential run given a root label, we could group the stars with the same root label together and match them at the same time when iterating through VertexIter. For the matching result write cost, we developed a compression algorithm for star's matching results that could be wrote sequentially (Section 5.2). As a result, we could scan the huge data graph only once and all the I/Os are sequential.

5.2 Compressed Star Matching Results Wrote Sequentially

As our experiment shows that a small graph with 10^5 edges could easily results in 10^{10} rows of matching results (Section 6). Even though we could use stars and auxiliary optimizations to drop

Algorithm 1: Star Decomposition

input : The pattern graph p
output: A sequence of stars with a specific order

function `DecomposeStars(p)`
- $results \leftarrow \emptyset$;
- $p' \leftarrow p$;
- $R \leftarrow \{\max_{u \in V(p)} f(u)\}$;
- **while** $R \neq \emptyset$ **do**
  - $root \leftarrow \max_{u \in R} f(u)$;
  - $R \leftarrow R \setminus \{root\}$;
  - $R \leftarrow R \cup \{leaf \mid leaf \text{ is adjacent to } root \text{ in } p'\}$;
  - `RemoveVertex(`p', $root$`)`;
  - $R \leftarrow R \setminus \{u \mid u \in p' \wedge \deg u = 0\}$;
  - $results \leftarrow results \cup \{$ `Star(`p, $root$`)` $\}$;
- **return** $results$

out useless matching results as soon as possible, the intermediate results could still be very large. Figure 7 illustrates this phenomenon that a small graph with only 6 vertices could result in 12 rows (48 vertices) of matching results. In the table we could find that u_1 and u_4 always match the same vertices v_2 and v_1, whereas the matching results of u_2 and u_3 are permutations of v_3, v_4, v_5, v_6. *The key of the matching result explosion problem is the explosive permutation.* In order to address this problem, we avoid the permutation by postpone the Cartesian production when matching stars, which is similar to VCBC [17] but we focus on the compression of property star's matching results for out-of-core systems.

Consider Figure 7, there is a symmetry with u_2 and u_3. We say that they have the same NeighborInfo or they form a NeighborInfo equivalence class, as they have the same label and same connections to the root u_4, and we can be sure that the matching results of u_2 and u_3 are always same. The NeighborInfo of u_1 is different from u_2 because u_1 has more edges connected to the root. By iterating through the neighbors of v_1, we can find the image set for each vertex in the pattern. Instead of permuting the matching vertices, we compress the matching result by just writing down the image sets of each NeighborInfo equivalence class as is shown in the right bottom corner in Figure 7. And Figure 8 gives a straightforward disk format to store the compressed star matching results. The final results can be retrieved by doing Cartesian product on the image sets and keeping the unique vertices. We called the compressed data as *SuperRow* since one SuperRow could generate enormous tuples by Cartesian production.

However, there are still two challenges to be faced in practice: 1. In real-world property graphs, as a celebrity vertex could have millions of neighbors, it could become a bottleneck if we have to scan the neighbors multiple times when matching a star; 2. The SuperRows should be written sequentially to reduce the I/O cost. If we want to scan the neighbors only once, we should be able to append the neighbor vertex to the corresponding image set, however, the variable-length image sets make it hard to address these problems. To solve this dilemma, for each SuperRow, we preallocate enough space based on the statistical information in the data graph, i.e., the size of neighbors with the NeighborInfo's

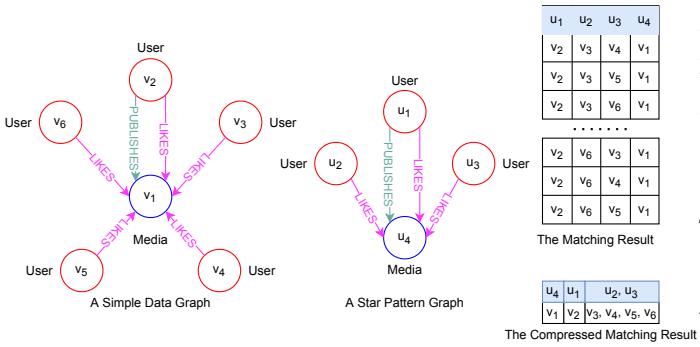

Figure 7: A small graph could results in enormous matching results.

label. Thus the vertices could be scanned only once and wrote to the corresponding image set sequentially.

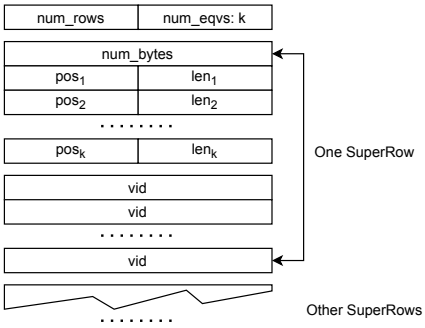

Figure 8: The disk format of our compressed star matching results.

5.3 Pipeline Join on Compressed Data

By far, we've got all the compressed matching results for stars and it's time to join them to obtain the final answer. A simple and straightforward method is the binary join, however, intermediate join results have to be materialized by doing so. Even though the compression ratio of star matching result is very impressive, the joined result could expand significantly because the permutation among roots is unavoidable [22]. To address this problem, we propose a indexed pipeline join algorithm on compressed data:

Consider the SuperRow that we discussed before, whose columns are image set for NeighborInfo equivalence class except the first column, which is the matching result of the root. Thus the first column of a SuperRow contains only a single vertex, which is suitable for the key of a index. In fact, the index is generated during the star matching process, for each SuperRow we append the root id and the position of it to the index file. With this index, we are able to locate to the corresponding SuperRows efficiently during the join process.

The basic structure of our pipeline join is a series of nested loops. However, unlike the traditional join problem, we join on image sets rather than single elements, which means set intersection is the most computation intensive operation. Consider the social media network, a trending media could easily attract millions or even billions of viewers, to join on such trending media rooted stars, we must calculate the set intersection of such enormous viewers. A conventional hash join method could easily eat up the memory of a PC and have poor locality. To address this problem, we provide an out-of-core sequential approach by merging on the image sets. Therefore the image sets should be sorted otherwise the sorting operation could be another bottleneck. In fact, with the elegant design of our vertex-centric property graph storage method, the vertices are already sorted in the data graph, and we can implement our sequential out-of-core set intersection for free.

5.4 Optimizations

In this section we discuss a series of optimizations for the property graph matching engine.

5.4.1 Predicate Push Down. The WHERE clause of a graph matching query specified a constraint or searching condition on the matching results. The constraint are expressed in the form of predicates, e.g., $=, \neq, >, \geq, <, \leq$. And the Boolean operator AND ($\wedge$), OR ($\vee$), NOT ($\neg$) can be used to combine multiple predicates into a new one. For example, in Figure **??**, there are three predicates concatenated by AND. Formally, the constraint is a function $\psi : PG \rightarrow B$ with PG the set of pattern graph and B the set of Boolean values. We will also use ψ to denote abstract predicate for simplicity in this section: $\psi(u)$ defines a constraint ψ on vertex u, e.g., "`id(u4) >= 2020`" defines a vertex constraint on u_4 where the ID of the matching vertex of u_4 must great than or equal to 2020; and $\psi(u_1, u_2)$ defines a constraint on vertex u_1 and vertex u_2, e.g., "`id(u1) < id(u2)`" defines a constraint on u_1 and u_2 that the ID of the matching of u_1 must be less than that of u_2.

Previous work usually ignore the constraint specification part of a graph matching query. If someone wants to query a pattern with a specific searching condition, she or he has to match the pattern graph first and then filter on the matching results, which leaves a lot of room for improvement because the user provided searching conditions could filter out many unnecessary partial results in an early phase.

However, it is still challenging to make use of the constraint provided by user's WHERE clause. The pattern graph and the constraint are logically two different things, we have to obtain enough information in order to use the constraint as early filter during the graph matching phase. For example, the constraint in Figure **??** include all the vertices in the pattern graph, only when the pattern graph is already matched could we got enough information to apply the constraint, which makes the constraint filter nearly useless.

To address this problem, as shown in Figure 9, we dive into the syntax tree of the graph matching query and decompose the searching condition into smaller parts which require only what we could got during the graph matching phase. Specifically, we decompose the searching condition into three parts: *vertex constraints*, *edge constraints* and *global constraint*. A vertex constraint is a function $\psi(u)$ mapping vertex u to Boolean values, and an edge constraint sets a constraint on edge (u_1, u_2) by a function $\psi(u_1, u_2)$. For example, in Figure 5 "`id(u4) < 2020`" sets a vertex constraint on u_4, "`id(u1) < id(u2)`" and "`id(u1) < id(u3)`" are edge constraints,

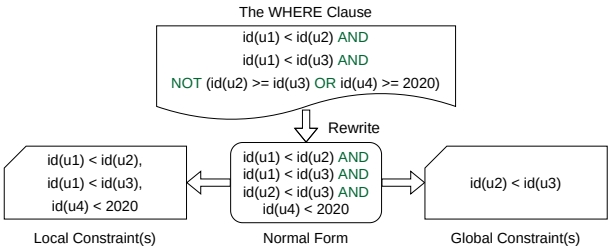

Figure 9: Constraint Analysis.

while "`id(u2) < id(u3)`" is not because there is no edge between u_2 and u_3. The *vertex constraints* and *edge constraints* are *local constraints* that only require local information that can be obtained during the graph matching phase. So they could then be pushed down to the data graph scanning phase to short-circuit useless matching results. A global constraint $\psi(u_1, u_2, \dots)$ sets a constraint on a series of vertices $u_1, u_2, \dots$, the information is insufficient during the data graph scanning phase.

Algorithm 2: Constraint Rewriting

input : *expr*: the abstract syntax tree of the WHERE clause
output: A set of simplified constraints connected by the AND ($\wedge$) operator

```
1  function ConstraintRewrite(expr)
2      match expr do
3          case ¬¬e do
4              return ConstraintRewrite(e)
5          case ¬(e1 ∨ e2) do
6              return ConstraintRewrite(¬e1) ∪
                ConstraintRewrite(¬e2)
7          case e1 ∧ e2 do
8              return ConstraintRewrite(e1) ∪
                ConstraintRewrite(e2)
9          case e do
10             return { Simplify(e) }
```

Logically, the AND ($\wedge$) operator create a new constraint $\psi = \psi_1 \wedge \psi_2$ by combining two constraints ψ_1 and ψ_2, where ψ_1 and ψ_2 can be used to check the matching results independently because there is no side effects in constraints, so we could safely split ψ into ψ_1 and ψ_2. Because local constraints are the earliest constraint filters, we should extract as much as possible. In order to make the constraint filters more efficient and extract more local constraints: Firstly, we optimize the AST by classic methods such as compile-time calculation, handle special cases such as "`WHERE false`". Then, we apply Algorithm 2 to analyze the syntax tree and rewrite it into *normal form*, where a normal form is a list of simplified constraints connected by the AND operator. In fact, the constraints are mostly specified by binary operators such as "$\leq$", "$\neq$", hence many constraints are naturally local constraints. And the De Morgan's law enables us to convert the OR ($\vee$) operator into AND ($\wedge$):

$$\neg(\psi_1 \vee \psi_2) = \neg\psi_1 \wedge \neg\psi_2 \tag{1}$$

So Algorithm 2 will always keep the semantics of the original user provided constraint. For example, the third predicate of the AND operator in Figure **??** would be rewritten to

```
id(u2) < id(u3) AND id(u4) < 2020
```

by applying De Morgan's law. And the WHERE clause of Figure **??** would be rewritten to the normal form:

```
WHERE id(u1) < id(u2) AND id(u1) < id(u3)
AND id(u2) < id(u3) AND id(u4) < 2020
```

Algorithm 3: Constraint Pushdown

input : The normal form of constraints *exprs* and the user described pattern graph p
output: The vertex constraints and edge constraints are pushed down to p and the global constraints will be returned

```
1  function ConstraintPushdown(exprs, p)
2      globals ← [ ];
3      foreach expr ∈ exprs do
4          match expr do
5              case ψ(u) do
6                  AddVertexConstraint(p, ψ(u))
7              case ψ(u1, u2) do
8                  if (u1, u2) ∈ Edges(p) then
9                      AddEdgeConstraint(p, u1, u2,
                        ψ(u1, u2))
10             case e do
11                 globals ← globals ∪ {e};
12     return globals
```

The normal form is then used to extract useful information to be pushed down to the pattern graph as in Algorithm 3. For each constraint in the normal form, we check if it is local constraint and then push it down to the corresponding vertex or edge. After that, We could then decompose it into stars. Our framework contains a JIT compiler that is able to emit callable closures based on the AST, and the local constraints can then be used to serve as early filters in the data graph scanning process to short-circuit unnecessary matchings as soon as possible.

5.4.2 Star Isomorphism. Consider Figure **??**, we generate three stars from the original pattern, and these stars are isomorphic with each other. Therefore the matching results of these stars are always the same, there is no need to match these stars again and again. Though the general graph isomorphism problem is NP complete, the isomorphism of stars are easier to check. We say that our stars in Figure **??** belong to the same Characteristic equivalence class, where Characteristic is a structure that stores the root and neighbors in a predefined order. By grouping isomorphic stars into the same Characteristic equivalence class, we could avoid unnecessary computation.

6 EXPERIMENTS

7 RELATED WORK

Despite the fact that general property graph matching problem is seldom discussed in previous works, simple graph matching has been widely studied. We survey these relevant work in this section.

In-memory Methods

Most of the early work assumes that the data graph and indices are fit in the main memory of a single machine. Sparked by Ullmann's backtracking algorithm [23], many subgraph matching algorithms have been proposed using different join order, filter rules, and neighborhood indices [3, 7, 8, 15, 19]. These algorithms usually use a DFS-style tree-based graph exploration to search the matchings without materializing intermediate results. However, these single machine in-memory algorithms are no longer suitable for nowadays billion-node graphs.

To address the scalability problem of single machine in-memory algorithms, many distributed subgraph matching algorithms have been proposed [13, 14, 18, 21, 22]. Because the vertices of the data graph are scattered among machines, these algorithms usually match smaller patterns and get the final result by join operation. For example, Sun et al. [22] introduce a star-like basic matching unit called STwig, and implement their subgraph matching algorithm on top of the Trinity [20] memory cloud. Lai et al. [13] propose TwinTwig join using MapReduce, where a TwinTwig is either a single edge or two incident edges of a vertex. The SEED [14] algorithm use both star and clique as the join units, and use clique compression technique to further improve the performance. However, these distributed algorithms still suffer from severe memory crisis, because the size of partial results grow exponentially with respect to the size of the date graph. Moreover, they must be transferred to other machines before join, which is the most expensive operation in a parallel system such as MapReduce.

Besides, the optimization of a subgraph matching algorithm relies heavily on the underlying graph model:

Unlabeled undirected simple graph is perhaps the simplest graph model, which can be viewed as a special case of property graph with all the vertices and edges have the same label and have no multi-edges. Some authors distinguish this kind of graphs from others and designate the matching problem of this kind of graph as *subgraph listing* [10, 13, 14, 17, 21, 21, 23]. CBF [17] is the state-of-the-art subgraph listing algorithm, which decompose the pattern graph into a several basic structures called *crystals*, and match these basic units with partial results compressed by the VCBC algorithm. However, it is unable to support general property graph because CBF relies on clique listing to match crystals, which implies the equivalence of vertices in a clique (complete graph) and is not the case of property graph model because of labels and direction of edges.

Another widely studied graph model is vertex-labeled undirected simple graph [4, 7, 18, 19, 22]. $\text{Turbo}_{\text{ISO}}$ [7], for example, is turbocharged by the concept of *neighborhood equivalence class* (NEC). It outperforms other competitors by safely avoid the permutation of all possible vertices in the same NEC. A NEC is a set of vertices in the pattern graph, where every vertex has the same label and the same set of neighbors. However, things become more complex and make it not suitable for the property graph model. Because one has to check the labels of vertices, labels of edges, directions of edges in order to test the isomorphism of a property graph, and the real-world multigraphs make life even harder.

Out-of-core Methods

Many out-of-core triangle enumeration algorithms have been proposed [2, 9, 11, 12]. However, all these algorithms only deal with triangulation, a special case of the graph matching problem. Recently, DualSim [10] take a further step and is able to match general unlabeled undirected graphs. To avoid the materialization of intermediate results, it fixes the data vertices by fixing a set of disk pages and then find all matchings in these pages. Apparently, every page of the data graph must be swapped in/out many times in order to get the final result, which lead to severe I/O cost. In contrast, our approach will load the pages sequentially at most once, and we can also use the compressed partial results to boost afterward queries.

8 CONCLUSION

REFERENCES

[1] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40. https://doi.org/10.1145/3104031

[2] Shumo Chu and James Cheng. 2011. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, Chid Apté, Joydeep Ghosh, and Padhraic Smyth (Eds.). ACM, 672–680. https://doi.org/10.1145/2020408.2020513

[3] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 26, 10 (2004), 1367–1372. https://doi.org/10.1109/TPAMI.2004.75

[4] Vinícius Vitor dos Santos Dias, Carlos H. C. Teixeira, Dorgival O. Guedes, Wagner Meira Jr., and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1357–1374. https://doi.org/10.1145/3299869.3319875

[5] Joshua A. Grochow and Manolis Kellis. 2007. Network Motif Discovery Using Subgraph Enumeration and Symmetry-Breaking. In *Research in Computational Molecular Biology, 11th Annual International Conference, RECOMB 2007, Oakland, CA, USA, April 21-25, 2007, Proceedings (Lecture Notes in Computer Science)*, Terence P. Speed and Haiyan Huang (Eds.), Vol. 4453. Springer, 92–106. https://doi.org/10.1007/978-3-540-71681-5_7

[6] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy J. Lin. 2014. Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *PVLDB* 7, 13 (2014), 1379–1380. https://doi.org/10.14778/2733004.2733010

[7] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. $\text{Turbo}_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 337–348. https://doi.org/10.1145/2463676.2465300

[8] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 405–418. https://doi.org/10.1145/1376616.1376660

[9] Xiaocheng Hu, Yufei Tao, and Chin-Wan Chung. 2013. Massive graph triangulation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 325–336. https://doi.org/10.1145/2463676.2463704

[10] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, Jeong-Hoon Lee, Seongyun Ko, and Moath H. A. Jarrah. 2016. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1231–1245. https://doi.org/10.1145/2882903.2915209

[11] Jinha Kim, Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, and Hwanjo Yu. 2014. OPT: a new framework for overlapped and parallel triangulation in large-scale graphs. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer

Özsu (Eds.). ACM, 637–648. https://doi.org/10.1145/2588555.2588563

[12] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[13] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable Subgraph Enumeration in MapReduce. *PVLDB* 8, 10 (2015), 974–985. https://doi.org/10.14778/2794367.2794368

[14] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2016. Scalable Distributed Subgraph Enumeration. *PVLDB* 10, 3 (2016), 217–228. https://doi.org/10.14778/3021924.3021937

[15] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *PVLDB* 6, 2 (2012), 133–144. https://doi.org/10.14778/2535568.2448946

[16] Roger A. Pearce, Maya Gokhale, and Nancy M. Amato. 2010. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2010, New Orleans, LA, USA, November 13-19, 2010*. IEEE, 1–11. https://doi.org/10.1109/SC.2010.34

[17] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph Matching: on Compression and Computation. *PVLDB* 11, 2 (2017), 176–188. https://doi.org/10.14778/3149193.3149198

[18] Marco Serafini, Gianmarco De Francisci Morales, and Georgos Siganos. 2017. QFrag: distributed graph search via subgraph isomorphism. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 214–228. https://doi.org/10.1145/3127479.3131625

[19] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB* 1, 1 (2008), 364–375. https://doi.org/10.14778/1453856.1453899

[20] Bin Shao, Haixun Wang, and Yatao Li. 2012. *The Trinity Graph Engine*. Technical Report MSR-TR-2012-30. https://www.microsoft.com/en-us/research/publication/the-trinity-graph-engine/

[21] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 625–636. https://doi.org/10.1145/2588555.2588557

[22] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *PVLDB* 5, 9 (2012), 788–799. https://doi.org/10.14778/2311906.2311907

[23] Julian R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (1976), 31–42. https://doi.org/10.1145/321921.321925