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

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

## ABSTRACT

Graph matching is one of the most important applications in graph databases. There are many works have claimed that they can be an order or even orders of magnitude faster than the existed industrial graph databases. However, there are still two big gaps that hinder these algorithms from being widely adopted in the real-world scenarios: (1) the complexity of real-world queries and the simplicity of graphs that the existing algorithms can handle; and (2) the huge memory and high-quality network requirements for the existing algorithms to query on large graphs and the limitation of resource budget.

To resolve the above problems, we design and implement a practical property graph matching algorithm that can execute efficiently in an out-of-core enviroment. The innovation of our algorithm includes (1) a novel decomposition method of the graph query that can handle both complex graph patterns and property constraints specified by WHERE clauses; (2) an optimized compression representation of the intermediates results and the corresponding join algorithm that do not need to fully decompress it untill the final results; (3) a set of effective data structures for disk-based graph indexes and join process. The evaluation results show that ???.

**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

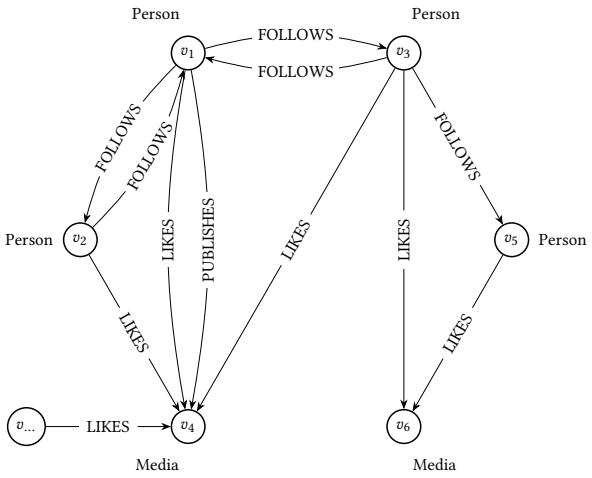
Graph matching is one of the most important applications of graph databases. It is widely used in many different fields, such as Twitter's recommendation systems [8, 28], electronic computer-aided design [21], and protein-protein interaction (PPI) networks [20]. Nowadays, users of an industrial graph database such as Neo4j[1] can easily model and manipulate their data as property graphs and expressing their queries via the Cypher [5] query language. However, although it's convenient, **the matching process of these industrial graph databases are notoriously time and resource consuming**. As a result, many novel subgraph matching algorithms have been proposed [4, 9, 22, 24, 27, 29], which usually claim an order or even orders of magnitude of speedup. However, **there are still two gaps that hinder these algorithms from being widely adopted in the real-world scenarios**: the first is the gap between the complexity of real-world queries and the simplicity of graphs that the existing algorithms can handle; while the second gap is the huge memory and high-quality network requirements for the existing algorithms to query on large graphs and the limitation of resource budget.

As for the first gap, the property graph model is the de facto graph model for graph databases, these graphs are directed labeled (both vertices and edges may have labels) multigraphs (two vertices may be connected by more than one edges) [23], users of an industrial graph database can also provide extra searching conditions via, for example, the WHERE clause [1]. However, **current studies only focus on simple graphs and extra searching conditions are not supported** [9, 12, 19, 22, 29]. This limitation is not an engineering problem that can be solved by adding more if/else in the code. It is actually a serious algorithm problem because the optimizations of existing algorithms rely heavily on the equivalence of vertices [9, 22], but in a property graph matching problem the condition of partitioning vertices that attached with labels and condition filters into equivalent groups are much more complex. For example, Figure 1b shows an pattern graph under the property graph model which can be commonly found in recommendation systems of a social network, here we represent a labeled directed multigraph and the extra searching condition $C$. (In fact, the pattern graph can also contain undirected edges, which means one could ignore the direction of some edges in the data graph.) In contrast, Figure 1c shows a simple unlabeled undirected pattern graph with similar topology. This two kinds of data model are essentially different, e.g., $u_1$ and $u_4$ are topologically equivalent in Figure 1c, however this equivalence does not hold in Figure 1b because of the labels, directions of edges, and extra searching conditions. More details of the problem caused by these equivalent conditions will be discussed in Section 3.

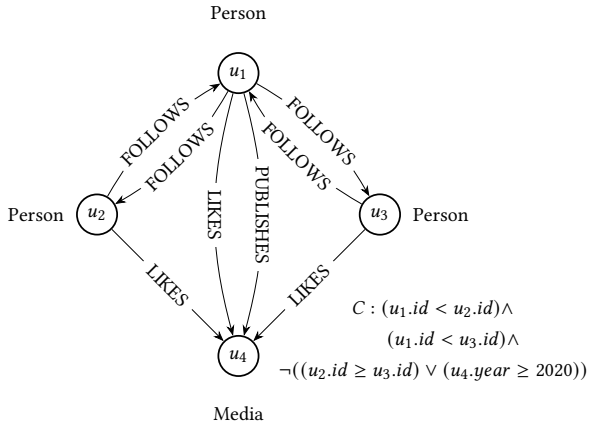
For the second gap, due to the well-known locality problem of graph exploration, current researches focus on using distributed systems to handle large graphs. However, **keeping all the data in memory is extremely uneconomic for graph matching problem** because (1) The memory needs to not only hold all the graph data but also the intermediate results of a graph matching algorithm, which can usually be dozen times larger than the original graph (Table 1). And these systems usually require tens or even


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

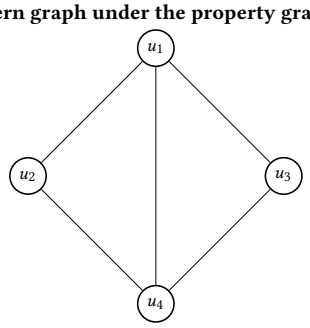
[1] https://neo4j.com




**(a) A part of a huge social network data graph.**




**(b) A pattern graph under the property graph model.**




**(c) A pattern graph under the simple graph model.**

hundreds of GB of memory, which is rarely seen on commercial machines; (2) The locality problem of graph exploration heavily effects the scalability of these distributed algorithms, so that they require high-quality network and still have only sub-sub-linear scalability [4]; (3) The startup overhead of distributed algorithms also reduces the response time, and becomes a bottleneck for small graphs. For example, to match on CiteSeer, it takes Fractal [4] ??? to boot the system, while executing the algorithm itself only takes ???. Moreover, even though distributed computing resources are available from commercial cloud providers, it is still a hard problem to partition graphs across the cluster nodes, optimizing and debugging distributed algorithms [14, 18]. Not to mention that, it is also financially expensive for ordinary users to purchase and maintain a cluster.

Therefore, in order to match up with the need of real-world graph databases, **we must provide an efficient and scalable disk-based solution that can match property graphs**.

## Challenges & Contributions

As discussed in above, it is especially valuable to design an real-world property graph matching algorithm that can both handle complex graphs and graph queries and does not require all the data to be held in memory (i.e., execute fast even in an out-of-core environment). However, there are many challenges towards this object.

As a summary, there are two kinds of approaches exist to match subgraphs, differ on whether intermediate results are materialized or not: The first kind of approaches are based on Ullmann's backtracking tree-search method [9, 12, 17, 30], which does not materialized intermediate results and is usually adopted by earlier in-memory algorithms. However, **this strategy is not very suitable for a graph database**: On one hand, it may incur considerable disk reads since the data vertices are scattered among the pages in disk. On the other, the evaluation of new queries cannot be boosted by previous queries, because this method doesn't generate reusable partial results. The second kind is join-based [15, 22, 27], which is also adopted by this paper. It works logically as follows: (1) Decompose the pattern graph into a series of smaller subgraphs, e.g., edges, and match these subgraphs by filtering the data graph, (2) materialize the partial results which can be used for afterward queries without repetitive computation, (3) and the final result is obtained by joining them together. However, in order to design an efficient disk-based property graph matching system, **challenges have to be faced in each of these steps**.

*Filter on Data Graph.* During the first filtering stage, two challenges exist. **One is that the result of the filtering process should be as small as possible**. Therefore, a simple edge-based join (the pattern graph is decomposed into a series of edges) method is inefficient, as matching an edge cannot make use of the graph structure information, which will lead to numerous useless partial results. In

contrast, a more complex decomposing method requires a properly designed graph storage format that allow efficient indices to be preserved and updated for faster filtering. **The other challenge is reducing I/O, essentially the random access problem**, which is the curse for performance improving of all out-of-core systems. **It is desirable that the filtering process scan at most once of the data graph stored on disk**, hence avoid unnecessary reads and cause no random accesses.

We adopt two techniques to address the first challenge: First, **pattern graph is decomposed into a series of stars (e.g., Figure 2) rather than edges**, which is similar to the STwig structure [29]. It contains a root vertex and the neighbor vertices of the root. Distinct from STwig, **our star structure support property graph features**, e.g., multigraph, edge labels, etc, whereas STwig can only match undirected simple graph. More over, **our decomposition algorithm will keep more structural information of the pattern graph** (see Section 3 for more details), which can reduce the size of partial results by up to ???% as our experiments show. Second, **our algorithm will push the searching conditions provided by users down to the star matching phase, rather than keep them until we get the partial results**. Specifically, it automatically rewrite the WHERE clause and extract useful information by applying Boolean algebra, then apply these extra information to filter out useless partial results. For example, in Figure 2, $C_1$, $C_2$ and $C_3$ was extracted from the original $C$. Experiments shows that this technique will reduce ???% to ???% of the intermediate results.

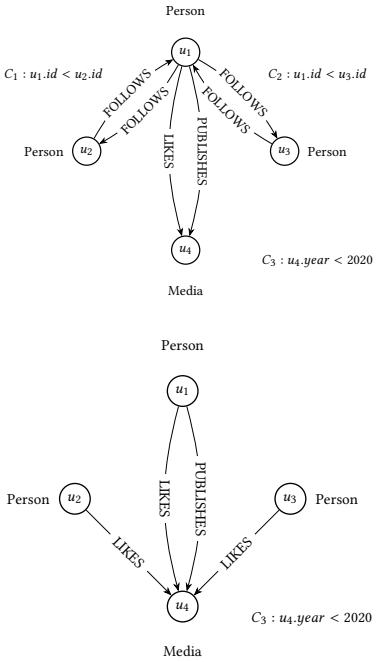



**Figure 2: The stars decomposed from the pattern graph in Figure 1b.**

To address the second challenge, **we design a compact disk format of the data graph that can match stars efficiently. With the help of a simple label-to-vertex index, we can quickly locate the candidate pages and only these pages will be swapped into memory**. To avoid unnecessary page swap even further, **we group the pattern stars with same root label together, so these stars will be matched in a single sequential read**. That is to say, every page of the data graph will be read sequentially at most once. Experiments shows that, for a billion-node graph, we can match a star in less than ???.

| Data set | $\|V\|$ | $\|E\|$ | \|Partial Results\| |
|---|---|---|---|
| CiteSeer | 3312 | 4591 | ??? |
| MiCo | | | |
| US Patents | | | |
| YouTube | | | |

**Table 1: The size of partial results on different data sets. (TODO)**

*Compression of Intermediate Results.* The intermediate results produced during the graph matching processes is a double-edges sword. Even though it can be used as caches to boost afterward matching queries, **the size of the partial results grows exponentially when the size of the pattern graph grows, which heavily affect the I/O cost**. For example there are $\binom{n}{3} = O(n^3)$ triangles in a complete graph with $n$ vertices, which is far more larger than the original graph ($O(n^2)$ edges). Table 1 gives more practical examples. To address this challenge, we adopt and extend a novel lossless graph compression algorithm called VCBC [22]. The idea of this algorithm is straightforward: By grouping the matching results with the same vertex cover $V_C$ in the data graph, the vertices in $V_C$ will be stored only once and other vertices will be stored as separate sets without repetition (We call it SuperRow because after doing Cartesian production, it is equivalent to a sequence of relationship rows.) VCBC has a significant compression ratio of several orders of magnitude [22].

However, challenge still have to be faced to implement VCBC efficiently for an out-of-core property graph matching system, since **VCBC was originally designed for unlabeled undirected simple graphs** like the graph in Figure 1c [22]. As we have stated before, the vertices in a simple graph are isotropic, e.g., all the neighbors connected to the root are equivalent, so they can be stored together in the compressed intermediate result and the disk-write is sequential. But **things are different for a property graph**. In Figure 2, for example, $u_2$ and $u_4$ have different labels, and their connections to the root $u_1$ are also distinctive. So the matching vertices of them can not be stored in the same place, and it is unknown where to store these matching results ahead of time.

To address this problem, a naive approach may use buffers in core to store the partial matching vertices, and then copy them to disk after finishing the graph scanning. However, the naive method will encounter numerous data movement, and the buffer may also exceed the size of the core because of the existence of vertices with high degree, e.g., popular social media usually have billions of viewers. Therefore, **we extend the original VCBC to make it support property graphs by proposing a novel space allocation method, which avoid the unnecessary data movement and**

**will write the compressed data in a sequential style**. Specifically, we use the statistical information of the data graph to get the upper bound of the matching result, and calculate the locations to store matched vertices based on these information. By using memory-mapped file, the data will be appended to the pages on disk, which avoid the random disk access problem. Experiments shows that our extended VCBC algorithm still keep a very high compression ratio by up to ???, and the run time to write the compressed data for a billion node graph is just ???.

*Join on Compressed Data.* Since the intermediate results are now compressed, a new challenge arises: How to join on these compressed data efficiently? Firstly, **we still have to face the space allocation problem because the join result is also compressed**. We adopt a similar technique as the one used in the last section to get the upper bounds of the matched result sets, and use these information to make the space allocation, so the result sets can be appended sequentially afterward. Secondly, as the basic operation unit is set of vertices, which is distinct from binary join, **the join on compressed data must calculate set intersection efficiently**. To address this challenge, we propose an efficient merge intersection algorithm, which avoids the random access problem and will run in $O(m+n)$ time without the need of sorting. Specifically, the vertices are sorted ahead of time in the data graph, and the order will always hold in the intermediate results without further sorting. So the interaction can be achieved efficiently on disk by two sequential reads and one sequential write, without the need of complex indices and random I/Os. As experiments show, our join algorithm only takes ???% of the total run time, and the memory consumption is below ???.

*Summary.* We have implemented the prototype of this out-of-core property graph matching framework, and run extensive experiments on real graphs with different scales. The results demonstrate that our solution outperforms the state-of-the-art methods 4. Most excitingly of all, compared to distributed solutions, we can solve the same problem on a single PC in a reasonable time with only ???% of the CPUs, and ???% of the memory.

The contributions of our work are summarized as follows:

- We propose an efficient our-of-core property graph matching solution, which can well handle graphs ranging from small to billion node graphs.
- We propose a new method to decompose pattern graphs and user provided searching conditions, which will help to filter out useless matches in an early stage.
- We design a simple and effective disk format to store complex property graphs, which can explore the data graph efficiently and is simple enough to be ported to existing databases.
- We adopt a novel graph compression algorithm and extend it to support property graph, which reduces the size of intermediate results significantly.
- We propose an efficient join method for compressed data, without the need of complex indices, we can get the results by sequential disk accesses.

The rest of the paper is organized as follows: we introduce preliminaries and related works in Section 2. Section 3 elaborates the design of our solution. We show the experiment results in Section 4. And finally, we conclude the work in Section 6.

## 2 BACKGROUND

This section introduces the formal definition of property graph matching.

A *property graph* is a **directed vertex-labeled edge-labeled multigraph 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., $\sigma$ in $G$, because similar techniques can be used as processing the labels $\lambda$. Thus, the property graph $G$ can be denoted by $(V(G), E(G), \rho_G, \lambda_G)$. Please note that the total function $\rho_G$ 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, e.g., there are 2 different edges between $u_1$ and $u_4$ in Figure 1b. 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 $\forall e \in E(G), \rho_G(e) = (u, v) \implies u \in V_C \vee v \in V_C$.*

Definition 3 (Subgraph). *A property graph $F$ is called a subgraph of a property graph $G$, written $F \subseteq G$, if $V(F) \subseteq V(G)$, $E(F) \subseteq E(G)$, $\rho_F$ is a restriction of $\rho_G$, and $\lambda_F$ is a restriction of $\lambda_G$.*

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 $\theta$ is fixed. However, due to automorphism, where an *automorphism* of a graph is an isomorphism of the graph to itself, the bijection $\theta$ may not be unique.

Definition 5 (Property Graph Matching). *Given a data property graph $D$, a pattern property graph $P$ and a searching condition $\psi : PG \rightarrow B$ 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 $\psi$ in their definition of graph matching [12, 15, 22, 27]. And they adopt a loosely related technique called *symmetry-breaking* [7], which ensures there is a unique bijection $\theta : V(P) \rightarrow V(F)$ 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 $\psi$ to filter out unnecessary matchings not only symmetry-breaking conditions. Thus, the searching condition we 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 FRAMEWORK OF OUR APPROACH

In this section, we propose our out-of-core property graph matching framework.

### 3.1 Overview

Figure 3 shows the main execution workflow of our framework. Basically, the framework contains three parts: 1. Filter on data graph; 2. Compression of intermediate results; 3. Join on compressed data. We adopt a join-based method instead of tree-search method to match property graph in an out-of-core environment for two reasons: Firstly, join-based method is more suitable for real-world graph databases, the pattern graph is decomposed into smaller parts and matched separately and these partial results can be used as caches for queries afterward; Secondly, a tree-search method will result in considerable disk I/O because the data vertices are scattered among the disk file.

In order to demonstrate the workflow clearly, consider the Cypher query in Figure 4 which corresponds to the pattern graph in Figure 1b (For simplicity, the properties are ignored and use the ID of vertices instead in the WHERE clause).

**A graph matching query consists two parts: the pattern graph description part (the MATCH clause) and the constraint specification part (the WHERE clause)**. It is also possible to

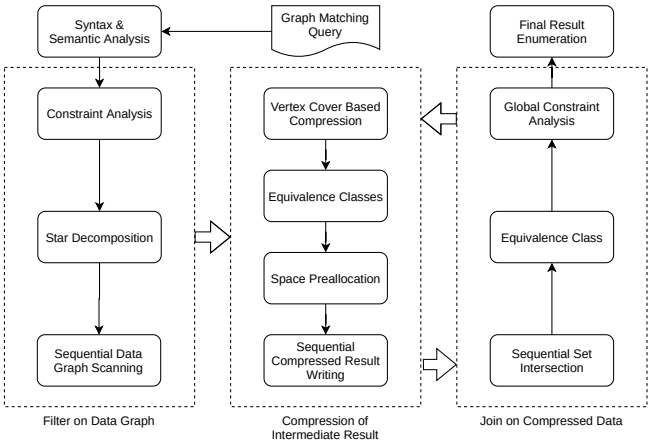



**Figure 3: Main Execution Workflow.**

```
MATCH (u1:Person)-[:FOLLOWS]->(u2:Person)-[:FOLLOWS]->(u1),
      (u1)-[:FOLLOWS]->(u3:Person)-[:FOLLOWS]->(u1),
      (u1)-[:PUBLISHES]->(u4:Media), (u1)-[:LIKES]->(u4),
      (u2)-[:LIKES]->(u4)<-[:LIKES]-(u3)
WHERE id(u1) < id(u2) AND id(u1) < id(u3) AND
      NOT (id(u2) >= id(u3) OR id(u4) >= 2020)
```

**Figure 4: Graph matching query of the pattern graph in Figure 1b.**

support other graph query language if the parser is changed respectively. The syntax and semantic analyzer would analyze the graph matching query, check the graph and types, and then generate a valid abstract syntax tree (AST). Readers interested in these subjects could refer to the textbook [6] and we would not discuss it further in this paper. In the following sections, we will focus on the three main stages of our property graph matching system (as shown in Figure 3).

Section 3.2 describes the first filtering stage, in which we try to reduce not only the output size of the intermediate results, but also the input size by reading only necessary parts from the huge data graph file. This is achieved by **a properly decomposition** of both the constraint specification part (Section 3.2.1) and the graph description part (Section 3.2.2) of the query, which is distinct from previous works that our method will hold all structural information that can be used to minimize the disk I/O. We also design an efficient disk-based **graph index** (Section 3.2.3), such that only one sequential read of the origianl graph is needed to match all the decomposed atomic query and the preserved structural information can be used to skip unnecessary parts of the graph.

In Section 3.3, we demonstrate the data structure that we use to express the intermediate results of graph matching **with an extremly high compression ratio**. Our compression algorithm is based on VCBC [22], and we will give a brief review of the VCBC algorithm in Section 3.3.1. To reduce the I/O cost even further, in Section 3.3.2, we study the *NeighborInfo equivalence classes* in the pattern graphs and use these equivalence classes to avoid blindly matching result writing. To put theory into practice, one more challenge is how to write this compressed expression to file without

introducing random disk I/O. To solve this problem, we propose a space pre-allocation method in Section 3.3.3.

Finally, Section 3.4 provides how we join the compressed intermediate results to obtain the final matching results. To reduce the I/O cost, our algorithm is elegantly designed such that the set intersection operation could be performed sequentially in linear time (Section 3.4.1). The space allocation problem still occurred during the join phase, in Section 3.4.2 we will address this problem, and we will show that the equivalence classes can still be applied here to reduce I/O cost further. At last, in Section 3.4.3, we will describe how to boost the join operation by the user-provided searching conditions.

## 3.2 Filter on Data Graph

This section describes how we filter on the data graph efficiently by analyzing the user provided searching constraint, decomposing the pattern graph into stars, and scanning the data graph sequentially.

*3.2.1 Constraint Analysis.* 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 4, 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 $\psi$ to denote abstract predicate for simplicity in this section: $\psi(u)$ defines a constraint $\psi$ 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 4 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 5, 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

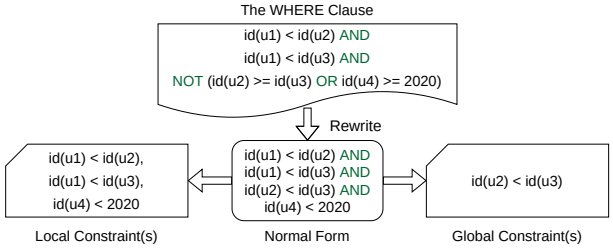



**Figure 5: Constraint Analysis.**

$\psi(u_1, u_2)$. For example, in Figure 1b "`id(u4) < 2020`" sets a vertex constraint on $u_4$, "`id(u1) < id(u2)`" and "`id(u1) < id(u3)`" are edge constraints, 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, however, it still contains many useful information that can be used during the join phase, and we will discuss it later in Section 3.4.

**Algorithm 1:** 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 $\psi_1$ and $\psi_2$, where $\psi_1$ and $\psi_2$ can be used to check the matching results independently because there is no side effects in constraints, so we could safely split $\psi$ into $\psi_1$ and $\psi_2$. **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 1 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 "$\leq$",

"≠", hence many constraints are naturally local constraints. And the De Morgan's law enables us to convert the OR (∨) operator into AND (∧):

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

So Algorithm 1 will always keep the semantics of the original user provided constraint. For example, the third predicate of the AND operator in Figure 4 would be rewritten to

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

by applying De Morgan's law. And the WHERE clause of Figure 4 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 2:** 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 2. 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. For example, we would obtain the pattern graph in Figure 6 after doing constraint analysis for the query in Figure 4. After that, We could then decompose it into stars (Section 3.2.2). 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 (Section 3.2.3).

*3.2.2 Star Decomposition.* **To design a join based graph matching algorithm, the first thing is to decide what is the basic join unit.** The simplest join unit is edge, however, it will result in huge amount of useless intermediate results if we use edge as basic unit. Some authors may adopt more complex structure such as frequent subgraphs as their basic unit, however, super-linear indices are inevitable for these algorithms [29], and this problem would be more severe for out-of-core environment. Based on these observations, **we make a balance by adopting star as our basic matching unit**, where a star $S_k$ is a complete bipartite graph $K_{1,k}$ with one root and $k$ leaves. For one thing, stars can be matched

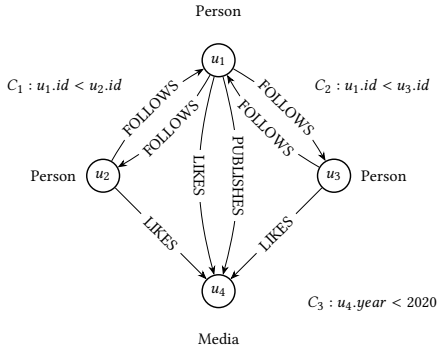



**Figure 6: The pattern graph with local constraints pushed down.**

sequentially without complex indices in a out-of-core environment, which we will discuss more details in the next section. And for another, stars can hold enough information, both structural information of the pattern graph and the additional searching constraints provided by user's WHERE clause, to reduce the size of intermediate results. In this section, we will show how to keep these information as much as possible by decompose the pattern graph into stars properly.

**Algorithm 3:** Star Decomposition

**input :** The pattern graph $p$ with vertex/edge constraints pushed down

**output :** A sequence of stars with a specific order

```
1  function DecomposeStars(p)
2      results ← [ ];
3      p′ ← p;
4      candidates ← {u| max_{u∈Vertices(p)} f(u)};
5      while candidates ≠ ∅ do
6          root ← Peek(candidates);
7          candidates ← candidates \ {root};
8          candidates ← candidates ∪ {leaf |
             leaf is adjacent to root in p′};
9          RemoveVertex(p′, root);
10         candidates ← candidates \ {u | u ∈ p′ ∧ deg u = 0};
11         results ← results ∪ { Star(p, root) };
12     return results
```

As many authors have stated before, **the matching order has a significant impact on the performance of a graph matching query** [9, 15, 29]. In our framework, matching order is determined by the stars' root order. We conclude that the root order affects the overall performance in two aspects: First, different matching orders result in different join operations which have different performance. Second, the size of the star matching result is determined by the root selection. Hence, we provide two rules respectively to guide the star decomposition algorithm (Algorithm 3):

- The root should be selected in the leaves of the previous star except for the first star.
- We prefer to select the vertex $u$ such that $f(u)$ is the maximum among the candidates, where $f(u) = \frac{\deg \times nc}{freq}$ is a heuristic function on the degree deg of $u$, number of constraints $nc$ and the frequency $freq$ of vertices with the same label of $u$ in the data graph.

The first rule indicates that the roots would form a connected vertex cover of the original pattern graph step by step. **By doing so, the join operation will always occurred between two adjacent stars**, and we could use a simple index to boost the join operation (Section 3.4). In contrast, for example, if the two stars are totally separated without any common vertices, then the join result of these stars would be the Cartesian product of respective star matching results, where most of the rows in the Cartesian product would be invalid. The Second rule considers both the information of the pattern graph and the distribution of data graph. We prefer to select root that contains more structural constraints (deg) and user provided constraints ($nc$), and prefer root with a low label frequency in the data graph. **By doing so the matching result of the star could be as small as possible which reduces the I/O cost and alleviates the burden of the later join operation.**

Previous work STwig [29] and TwinTwig [15] also use stars as their basic matching units. However, unlike these previous work which are designed for simple graphs, our stars fully support the real-world property graph model, and most importantly, we will show that the star decomposition algorithm of previous work has a lot of room to be optimized.

**Both STwig and TwinTwig are designed for undirected simple graph, which has a limited expressiveness and is not suitable for many real-world applications.** In contrast, we adopt the property graph model which is widely used in industrial graph databases. Figure 2 shows the decomposition results of the patter graph in Figure 1b, both vertices and edges are labeled and multiple edges may exist between a leaf and the root vertex. The vertex constraint $C_4$ and edge constraints $C_1$, $C_2$ are also shown in the graph.

**Apart from the graph model differences, we found that existing query decomposition algorithms still have a a lot of room to be improved.** As we discussed before, the result of the data graph scanning process should be as small as possible to avoid unnecessary I/O cost. The leaves of a star graph can be viewed as a bunch of constraints on the root vertex, which help to eliminate useless mappings of the root. That is to say: *more leaves, more better*. However, **previous work ignore the benefits of leaves and leave a lot of room to be optimized**. Compared to the decomposition result of previous work in Figure 2, our algorithm always keeps all the leaves in the pattern graph, while, the stars of previous work drop many useful leaves. In some cases, e.g., triangle graph, they even generate stars with only two vertices which reduce to edge decomposition. Apparently, an edge will result in much more unnecessary matchings than a star.

The key of the optimization of our algorithm is the use of $p'$: We first copy the pattern graph $p$ and obtain $p'$. Then we run the vertex cover finding algorithm on $p'$, vertices are removed one by one in $p'$ while $p$ is always unchanged. Finally, we use the original $p$ to induce stars. So the resulting stars will always hold as much leaves as possible, which improves the performance of data graph scanning significantly. Experiments show that this simple optimization can reduce the size of matching results to ???% of previous work.

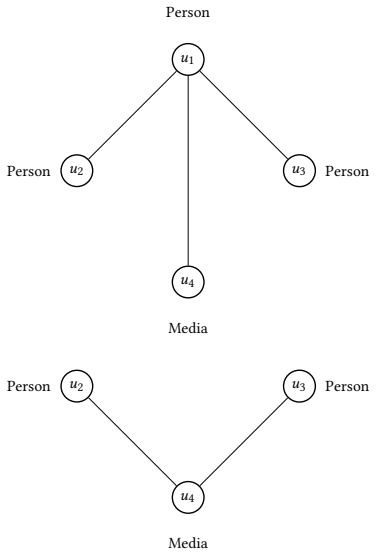



**Figure 7: The stars generated by previous algorithms.**

*3.2.3 Data Graph Scanning.* Now, we are ready to elaborate how we match stars efficiently in an out-of-core environment. First, we will propose our stars matching algorithm by introducing two iterators. And then we'll show how to implement the two iterators by designing a novel disk format to store the data format, such that the star matching process is I/O efficient, where **the huge data graph will be read sequentially without random access, and read at most once**.

*Stars Matching.* Random disk access problem is always a challenging problem for out-of-core systems, especially for the out-of-core graph matching problem where graphs are notorious for their poor locality. However, in this section, we will show that we could avoid random disk accesses by matching stars using two simple iterators. Furthermore, our techniques ensure that we would only read the necessary part from the huge data graph file, and these necessary parts would be only read sequentially once. That is to say, **we could match all the stars by scanning only the necessary part of the data graph file sequentially once**.

Consider the first star $star_1$ in Figure 2, the root vertex is $u_1$ with the label "Person", and it has three leaves $u_2$, $u_3$ and $u_4$. To match such a star in a data graph, we should find a star whose root also contains the label "Person" and it has at least 3 neighbors that could match $u_2$, $u_3$ and $u_4$ respectively. Then it is straightforward to obtain an naive method to match $star_1$ in a data graph $d$: Iterate through VertexIter($d$, *Person*) and check every neighbors of these vertices whether they could match $u_2$, $u_3$ and $u_4$, where VertexIter($d, l$) travels through all the vertices with the label $l$ in the data graph $d$.

For example, in Figure 1a, $d[v_1]$, the induced graph of $d$ on $v_1$, is such a star that could match $star_1$.

However, this naive star matching method has a great drawback for vertices with a high degree. Consider the second star $star_2$ in Figure 2, the root vertex is $u_4$ with the label "Media". And we suppose that in Figure 1a, $v_4$ is such a popular media that millions or even billions of persons have left their likes. The naive method have to visit all the neighbors of $v_4$ but most of them are useless because there is no vertex with label "Comment" in $star_2$. To solve this problem, we introduce the second iterator NeighborIter$(v, l)$, which visits the neighbors with the label $l$ of a given data vertex $v$. By iterating through NeighborIter$(v_4, Person)$, we could skip useless neighbors and only load the necessary ones.

Definition 6 (VertexIter and NeighborIter). *In a data graph $d$, given a vertex label $l$, VertexIter$(d, l)$ is a iterator that would visit all the vertices with the label $l$ in $d$ sequentially. Given a data vertex $v$ and a vertex label $l$, NeighborIter$(v, l)$ is a iterator that would iterate through the neighbors with label $l$ of $v$ sequentially.*

Now we will answer the question: How to match the neighbors with the help of NeighborIter? Consider $star_1$ in Figure 2, it is easy to find the symmetry in this star. We say that $u_2$ and $u_3$ are equivalent because they have the same vertex label, their connections to the root are the same, and the edge constraints $C_1$ and $C_2$ are also equivalent (Remember that an edge constraint is a function, that in this case $u_2$ and $u_3$ are just free variables). For simplicity, we introduce the definition NeighborInfo as follows:

Definition 7 (NeighborInfo). *A NeighborInfo of a neighbor vertex $n$ is a structure that stores:*

*(1) the label of the neighbor $n$,*
*(2) the vertex constraint of the neighbor $n$,*
*(3) the edges between the root vertex $v$ and the neighbor $n$ (the direction of the edges may be $v \rightarrow n$, $v \leftarrow n$ or $v$—$n$ because a user could ignore the direction of some edges in the pattern graph),*
*(4) and the edge constraints of the edges.*

In a star graph, **the leaves with the same NeighborInfo form a equivalence class, and there is no need to blindly permute all possible mappings for the vertices in the same NeighborInfo equivalence class**. Suppose that there are $m$ leaves with the same NeighborInfo in a star, for each leaf there are $n$ vertices that could be matched in the data graph. We have to use $m!n$ rows to store the matching result if we blindly permute the mappings for the leaves. In contrast, only $n$ rows are required if we match group these leaves together which reduces I/O cost dramatically. **So instead of matching each neighbors directly, we group the leaves of the star pattern graph by their NeighborInfo during the static analyzing phase, and then use the NeighborInfo to match equivalent vertices at the same time.** For each neighbor in the NeighborIter, we check whether it would be matched by, firstly, checking the local constraints of the neighbor, and then checking the edges between the root and the neighbor.

By far, there is only one challenge left: How to scan the huge data graph file only once? The size of the data graph file is proportional to the number of edges in the data graph. Existing in-memory algorithms would visit the vertices back and forth to explore the data graph, naively adopt these algorithms to the out-of-core environment would result in numerous swap in/outs, which is very time consuming and I/O inefficient. An algorithm that only scan the data graph file sequentially once is desirable, and we propose a positive answer by providing Algorithm 4. Consider the stars in Figure 2, there are two stars that the labels of the roots are different. VertexIter$(d, Person)$ and VertexIter$(d, Media)$ would be used to visit the vertices in the data graph $d$. If the data vertices with the same label are stored together in disk, then the VertexIter could load only the necessary part from disk and visit the vertices sequentially. Generally speaking, if there are $n$ stars generate by Algorithm 3, and the size of the labels of the roots is $m$, we have $m \leq n$ since multiple stars may have the same label of root. **Then we could use $m$ different VertexIter to scan the data graph file by grouping same-label-of-root stars together, and every iterator would only load a distinct part of the data graph sequentially** (There is a special case that some stars may isomorphic, and there is no need to do duplicate star matching, Section 3.3.2 would discuss it in detail). **Macroscopically, only the necessary part of the data graph would be loaded to match a pattern graph, and these parts would be read sequentially only once with the help of VertexIter and NeighborIter.**

**Algorithm 4:** Stars Matching

**input :** The data graph $d$ and a list of *stars* with the same root label
**output:** A list of compressed star matching *results* corresponding to the *stars*

```
1  function MatchStars(d, stars)
2      results ← InitializeResults(stars);
3      foreach v ∈VertexIter(d, stars[0].root.vlabel) do
4          candidates ←
             {star|star.root.VertexConstraint(v)
             ∧star.root.DegreeConstraint(v) };
5          nlabel_stars ← groups the candidates by leaves'
             label;
6          foreach (nlabel, stars) ∈ nlabel_stars do
7              foreach n ∈NeighborIter(v, nlabel) do
8                  MatchNeighbor(v, n, stars, results);
9      FinalizeResults(results);
10     return results;
11 function MatchNeighbor(v, n, stars, results)
12     foreach star ∈ stars do
13         foreach info ∈ star.NeighborInfos(n.vlabel) do
14             if MatchNeighborInfo(v, n, info) then
15                 AppendResult(results[star], n);
```

*Data Graph Disk Format.* To implement the two iterators efficiently, we design a compact disk format to store the data graph (Figure 8). The disk format is a little bit like to the conventional adjacency list format, that is, we store a vertex and it's neighbors together.

Specifically, in order to implement the VertexIter efficiently, **we should locate data vertices efficiently given a vertex label and visit these vertices without random disk access**. So **we**

**store vertices with same vertex label adjacently, and adopt a simple label-to-VertexIter index to boost the searching**. The num_bytes field would be used to calculate the position of the next data vertex, such that the VertexIter can visit the data vertices on disk sequentially to reduce I/O cost. Apart from that, we also store in-degree and out-degree which can be used as early filters during the star matching process.

For the NeighborIter, there is one thing to notice that we adopt the property graph model, all edges are directed and multi-edges may exist between a pair of vertices. To store these information, a naive method is to store in-edges and out-edges for each data vertex. However, this naive method will be very inefficient when matching property graphs. Consider the pattern graph in Figure 1b, suppose that in a data graph $v_1$ matches $u_1$ and we want to check whether $v_2$, a neighbor of $v_1$, matches $u_2$, then we have to scan all the in-edges to find if there is an edge from $v_2$ to $v_1$ with label "FOLLOWS", and all the out-edges to check if there is an edge from $v_1$ to $v_2$ with the same label. Real-world graphs usually have a high skewness, for example, a celebrity in a social network may have millions of followers. **These "celebrities" will become bottleneck if we adopt the naive method to scan all the adjacent edges.** To solve this problem, we propose a novel schema to store the neighbors, **all the edges connected to a neighbor are stored together with the neighbor, so the neighbor can be checked in place without scanning all the in-edges/out-edges of the root vertex**. And we also use a simple label-to-NeighborIter index to boost the searching of neighbors given the label of neighbor. The position of the next neighbor could be calulated by the num_n_to_v and num_v_to_n fields. And we could use NeighborIter to travel through the neighbors sequentially without random disk access.

The vertices are sorted by their ID in the data graph, which is very helpful for the join operation (Section 3.4). Given a series of vertices and a series of edges with arbitrary order, the data graph file could be generated in sorting time. And for simplicity, we ignore the key-value properties of a property graph and focus on the labels. In fact, these properties can be added to our disk format by simply add the corresponding fields. And the simplicity of the VertexIter and NeighborIter also makes it possible to implement them on other platform, e.g., industry-tested relational database, which we will study in future work to make dynamic property graph transactions and analysis easier.

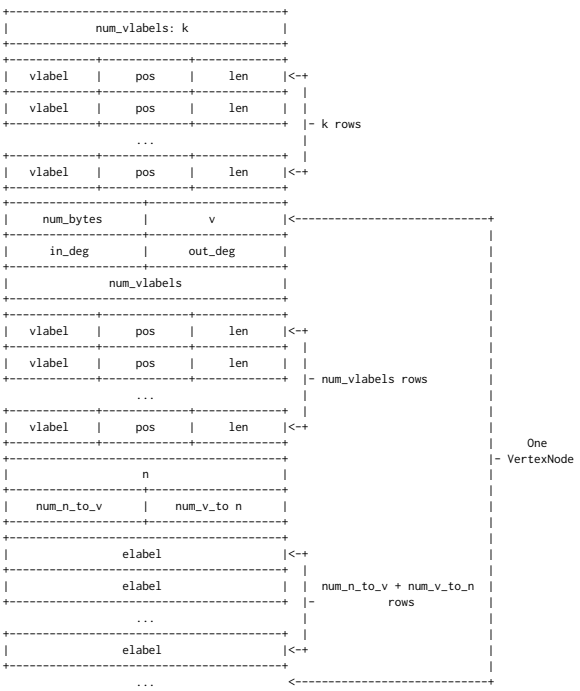



**Figure 8: Data graph disk format. (TODO: Update this figure.)**

## 3.3 Compression of Intermediate Result

In this section, we describe our efforts to reduce the I/O cost of materializing intermediate results by compression. Since both the star matching results and the join results are compressed, to make it clearly, we'll focus on the compression of star matching results in this section, and leave the compression of join results in Section 3.4. Firstly, we'll give a brief review of the theoretical VCBC algorithm in Section 3.3.1, and then provide our contributions to make it practical for out-of-core environment in the following subsections.

*3.3.1 Vertex Cover Based Compression.* Recall that in Definition 5, the property graph matching problem is to report the set $\mathcal{I}$. If we specify an order on the vertices of the pattern graph $P$, then $\mathcal{I}$ could be represented by a sequence of rows with each row containing the vertices in $D$ that match the vertices in $P$ with the order we specified. The space to store $\mathcal{I}$ is proportional to the number of matching graphs if we store these rows naively. Unfortunately, the size of matching result is super-linear with respect to the size of the data graph. Thus, a compression algorithm with a significant compression ratio is required to reduce the I/O cost for materializing the intermediate matching results. The *vertex cover based compression* (VCBC) [22] algorithm is such an attempt to resolve the output crisis of graph matching using output compression.

The authors of VCBC define the *helve* of an instance of $P$, where the instance $F$ is a subgraph of the data graph $D$ such that $F \cong P$.

Definition 8 (Helve). *Let $V_C = \{u_1, u_2, \ldots, u_k\}$ be a vertex cover of $P$. Let $F$ be an instance of $P$. The helve of $F$ is the vertored images of $V_C$ under the instance-bijection $f_F$:*

$$\mathcal{H}_{V_C}(F) = (f_F(u_1), f_F(u_2), \ldots, f_F(u_k))$$

*It is also denoted as $\mathcal{H}(F)$ if $V_C$ is obvious in the context. Similarly, the helves of an instance set $\mathcal{I}$ is defined as*

$$\mathcal{H}(\mathcal{I}) = \{\mathcal{H}(F) | F \in \mathcal{I}\}$$

Let $h_1, h_2, \ldots, h_l$ be the $l$ helves in $\mathcal{H}(\mathcal{I})$, the compression of $\mathcal{I}$ could then be obtained in the following steps:

(1) Define the *conditional instance set* $\mathcal{I}|h_i$ of $h_i$ as

$$\mathcal{I}|h_i = \{F | \mathcal{H}(F) = h_i\}$$

(2) For $\mathcal{I}|h_i$, identify the *conditional image set* $\mathrm{Img}_P(u|h_i)$ for each vertex $u \in V(P)$:

$$\mathrm{Img}_P(u|h_i) = \{f_F(u) | F \in (\mathcal{I}|h_i)\}$$

(3) Compress $\mathcal{I}|h_i$ with the concatenation of the conditional images $\text{Img}_P(u|h_1)$ for all vertices in $V(P)$, and we name $\text{code}(\mathcal{I}|h_i)$ as SuperRow in this paper:

$$\text{code}(\mathcal{I}|h_i) = \{\text{Img}(u|h_i)|u \in V(P)\}$$

(4) Obtain the VCBC $\text{code}(\mathcal{I})$ by concatenation:

$$\text{code}(\mathcal{I}) = \{\text{code}(\mathcal{I}|h_i)|i \in [1, l]\}$$

For a star graph, the root is the vertex cover. For example, in Figure 2, $u_1$ and $u_4$ are the roots and the vertex cover.

The decompression process works as follows to restore $\mathcal{I}$ from $\text{code}(\mathcal{I})$:

(1) For each $\text{code}(\mathcal{I}|h_i)$, let $S$ be the Cartesian product over the image sets:

$$S = \prod_{u \in V(P)} \text{Img}(u|h_i)$$

(2) Let $\mathcal{I}'|h_i$ be the set of tuples in $S$ without duplicated vertices that are validated by the searching condition $\psi$
(3) Finally, $\mathcal{I}' = \bigcup_{i \in [1,l]} (\mathcal{I}'|h_i)$ is the decompressed matching result.

VCBC is lossless such that $\mathcal{I}' = \mathcal{I}$ [22]. The data compression ratio of VCBC $\rho(\mathcal{I})$ could be calculated by

$$\rho(\mathcal{I}) = \frac{|\mathcal{I}|}{|\text{code}(\mathcal{I})|}$$

In fact, the compression ratio could have several orders of magnitude since VCBC postpone the resource consuming Cartesian product to the decompression stage.

There is one thing to note that, the VCBC algorithm only introduce a theoretical method to compress the matching results. **For a real-world out-of-core environment, many practical challenges have to be solved**, for example, how to design the disk format to store the VCBC compressed data efficiently? How to write the compressed data sequentially to disk? And how to decompress the VCBC data file in a sequential manner? These challenges would be solved in the following two subsections.

*3.3.2 NeighborInfo Equivalence Classes.* VCBC save the space by postponing the Cartesian product, however, this is not the only way to save space. In this subsection, we discuss how to reduce the size of matching results further by finding equivalence classes in the pattern graph, and finally introduce the disk format of our compressed data.

In Section 3.2.3, we introduce the concept NeighborInfo. In a star graph $S$, suppose that the leaves $n_1, n_2, \ldots, n_k$ have the same NeighborInfo, then their conditional image set would also be the same $\text{Img}_S(n_1|h_i) = \text{Img}_S(n_2|h_i) = \cdots = \text{Img}_S(n_k|h_i)$ because of symmetry. These leaves are equivalent under the relation "has the same NeighborInfo", so we could store the image set only once. This technique is of great importance for "celebrity" rooted stars, otherwise we have to store the numerous followers multiple times.

For some pattern graph, e.g., complete undirected graph with all the vertices and edges share the same label, the stars decomposed from it would always be the same. Formally, we say that these stars are equivalent under the relation "has the same Characteristic", where the Characteristic is defined as follows:

Definition 9 (Characteristic). *A Characteristic of a star $S$ is a tuple $(\psi, l, N)$, where:*

*(1) $\psi$ is the vertex constraint for the root of $S$.*
*(2) $l$ is the label of the root of $S$.*
*(3) $N = \{\text{NeighborInfo}(n)|n \in S.leaves\}$ is a set of NeighborInfos of the leaves in $S$.*

Apparently, isomorphic stars have the same Characteristic. These equivalent stars could be matched once for all, and the matching results could be stored only once.

**The previous NeighborInfo and Characteristic are two basic equivalence classes that can be used to avoid duplication in the star matching results.** In Section 3.4 we will show that these equivalence classes can also be used to reduce I/O cost for joined results. Before that, we will discuss how to design the disk format for the compressed data. Figure 9 shows the disk format of our compressed matching result. Theoretically, the VCBC compressed data is a sequence of SuperRows, where each SuperRow contains a list of conditional image sets $\text{Img}_P(u|h_i)$ for $u \in V(P)$. However, **because of the NeighborInfo equivalence class, we don't have to store all the conditional image sets**. To address this problem, we group the vertices by their NeighborInfo and store the conditional image set for each NeighborInfo instead. For each conditional image set, we store the starting position `pos` and the size of the set `len`, and the elements are stored at the end of each SuperRow.

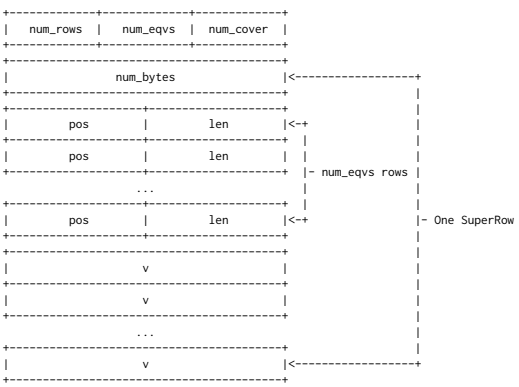



**Figure 9: Disk format for compressed matching result.**

*3.3.3 Space Allocation.* In Section 3.2.3 we showed that during the star matching process, the data graph file would only be read sequentially at most once without random disk access. Now in this subsection, **we will show that the matching result of stars would also be written sequentially with new data appended to the result file**.

For a star graph $S$, the vertex cover is the root, so the first image set for each SuperRow always contains only one element and this is the helve $h_i$. However, for the leaves in the star, the length of their image sets could only be known after all the neighbors of the helve are scanned in the data graph. That is to say, $|\text{Img}_S(n|h_i)|$ would be known only if $\text{Img}_S(n|h_i)$ is already written to file for $n \in S.leaves$.

To address this dilemma, a naive approach may allocate buffers in core to store the temporary data and then write these buffers to file. **However, this naive approach is very inefficient for the copy, and most of all, these temporary data could even be larger than the size of the main memory because real-world graph are skewed and "celebrities" may have millions or even billions of followers.** Thus, it is desirable to write the image sets direct to disk file. For this purpose, **we propose a space allocation algorithm to allocate space for image sets ahead of time**. Specifically, for each vertex in data graph, we store its neighbors together with the vertex and the neighbors are grouped by the vertex label (Section 3.2.3). For each group, the number of vertices are stored in the data graph file and this number is the upper bound of the corresponding image set, because these vertices are filtered and then write to the image set. Based on this observation, we could allocate enough space to store the image sets ahead of time. And with the help of NeighborIter, the neighbors are checked sequentially and then write to the proper place in the result file. **The operating system would keep at most `num_eqvs` pages to write the image sets, and the elements could be written sequentially to each of these pages.**

## 3.4 Join on Compressed Data

Theoretically [22], the join operation is performed as in Algorithm 5. However, **there are many challenges have to be conquered to implement it in an out-of-core environment efficiently**, i.e., How to implement the set intersection operation such that the I/O cost could be minimized? How to write the result of the join operation in an sequential manner without frequent random disk access? And how to apply the user-provided searching conditions to boost the join operation? In this section, we'll answer these problems in detail.

**Algorithm 5:** Join algorithm in theory.

**input :** A helve $h$ for an instance of the pattern graph $p$, a sequence of stars $\mathcal{P} = \{p_1, p_2, \ldots, p_\lambda\}$ decomposed from $p$, for each star $p_i \in \mathcal{P}$, projections $h_i$ on $p_i$ and conditional code $\text{code}(\mathcal{I}_{p_i}|h_i)$.

**output :** $\text{code}(\mathcal{I}_p|h)$.

```
1 function Join(h, p, P, {code(I_{p_i}|h_i) | i ∈ [1, λ]})
2     foreach u ∈ V(p) do
3         Img'_p(u|h) ← ⋂_{i∈[1,λ]∧u∈V(p_i)} Img_{p_i}(u|h_i);
4     code(I_p|h) ← apply the compression algorithm in
        Section 3.3.1;
5     return code(I_p|h);
```

*3.4.1 Sequential Set Intersection.* The authors of VCBC do not draw much attention on the set intersection operation in Algorithm 5 because they assume that these operations are performed in memory [22]. Given two image sets $s_1$, $s_2$ with $|s_1| \geq |s_2|$, a straightforward approach to compute $s_1 \cap s_2$ is to load these sets into memory and convert $s_1$ into hash table, then iterate through $s_2$ and check existence in the hash table $s_1$. **However, as illustrate in Algorithm 5, the set intersection operation is the innermost operations in the loop, and it could become the bottleneck if not handled properly.** Consider the social media network, the trending media could easily attract millions or even billions of viewers, to join on such trending media rooted stars, billions of viewers have to be loaded into memory and converted into hash table to perform the set intersection, which could easily eat up the memory for a PC and have poor locality because of the hash table. **A sequential approach with linear time performance is desirable.**

To address this challenge, we provide an out-of-core sequential approach. Suppose that if both $s_1$ and $s_2$ are sorted, then $s_1 \cap s_2$ could be obtained by a merge operation, the elements in $s_1$ and $s_2$ could be scanned sequentially in disk and there is no need to load them entirely into memory and there is no overhead to create the hash table. However, $s_1$ and $s_2$ must be sorted ahead of time otherwise the time consuming sorting operation could lead to disaster. Now let's check the data graph scanning phase in Section 3.2.3. The elements in $\text{Img}_{p_i}(u|h_i)$ are in fact written by filtering the NeighborIter sequentially. If the NeighborIter visits the vertices in a sorted order, then every $\text{Img}_{p_i}(u|h_i)$ is also sorted consequently. In fact, this is a byproduct since our data graph file is elegantly designed such that the vertices are sorted by their Id, so the order always preserves. **Thus we could implement our sequential out-of-core set intersection for free.**

*3.4.2 Space Allocation & NeighborInfo Equivalence Classes.* As in Section 3.3.3, the space allocation problem still occurs during the join phase since the join result is also compressed. For $u \in V(p)$, we would write $\text{code}_p(u|h) = \text{code}_{p_1}(u|h_1) \cap \text{code}_{p_2}(u|h_2)$ to the intermediate result file as illustrate in Algorithm 5. However, the size of $\text{code}_p(u|h)$ is not known until the set intersection operation is done. To address this problem and make the out-of-core sequential intersection discussed in the previous subsection doable, we estimate the upper bound for $|\text{code}_p(u|h)|$ similar to the approach in Section 3.3.3. This time, the upper bound is estimated as

$$\min |\text{code}_{p_1}(u|h_1)|, |\text{code}_{p_2}(u|h_2)|$$

since the cardinality of interaction could not be greater than the cardinality of the smallest set. Thus, the space to store the image sets could be allocated before the set intersection operation, and **the join result could then be written to these specific positions sequentially without random disk access or redundant memory buffer**.

During the star matching phase, the NeighborInfo equivalence class makes it possible to combine multiple leaves together and store the image set only once, which saves more space and reduces the I/O cost further. Now we will show that **NeighborInfo equivalence classes also exist in the join phase**. Consider the two stars $p_1$ and $p_2$ in Figure 2, $u_2$ and $u_3$ belong to the same NeighborInfo equivalence class in both of the stars respectively, i.e., for a helve $h$ and its projections $h_1$, $h_2$, we have

$$\text{Img}_{p_1}(u_2|h_1) = \text{Img}_{p_1}(u_3|h_1)$$
$$\text{Img}_{p_2}(u_2|h_2) = \text{Img}_{p_2}(u_3|h_2)$$

Consequently,

$$\text{Img}_{p_1}(u_2|h_1) \cap \text{Img}_{p_2}(u_2|h_2) = \text{Img}_{p_1}(u_3|h_1) \cap \text{Img}_{p_2}(u_3|h_2)$$

which means the equivalence class also hold in the join phase. Formally, if $u_1, u_2, \ldots, u_k \in V(p_i)$ belong to the same equivalence class $C_i$ in $p_i$, and $u_1, u_2, \ldots, u_k \in V(p_j)$ belong to the same equivalence class $C_j$ in $p_j$, then we have

$$\text{Img}_{p_i \cup p_j}(C|h) = \text{Img}_{p_i}(C_i|h_i) \cap \text{Img}_{p_j}(C_j|h_j)$$

where $p = p_i \cup p_j$ is the composition of $p_i$ and $p_j$, i.e., $V(p) = V(p_i) \cup V(p_j), E(p) = E(p_i) \cup E(p_j)$. And we could use $\text{Img}_{p_i \cup p_j}(C|h)$ to represent the image set of $u_1, u_2, \ldots, u_k$, **which reduces the I/O cost to write the join results**.

*3.4.3 Global Constraint Analysis.* Constraints can be applied only if enough information could be obtain, so we left the global constraints in Section 3.2. Now in the subsection, we are ready to study the global constraints.

Consider the Cypher query in Figure 4, if the WHERE clause is replaced by:

```
WHERE id(u1) < id(u2) OR id(u4) < id(u2)
```

Then in this case, we could not simplify it anymore because the two predicates are connected by the "OR" operator not the "AND", and the De Morgan's law is not helpful. This is a global constraint and it cannot be used to filter the vertices during the star matching process. Suppose that we want to match the first star $p_1$ in Figure 2, and $v_{10}$ in the data graph is the helve, i.e., $v_{10}$ matches the root $u_1$, and we want to check whether the neighbor of $v_{10}$, say $v_5$, could match $u_2$ or $u_3$. Then the first predicate `id(u1) < id(u2)` would certainly fail ($10 > 5$) but we cannot simply say that $v_5$ cannot match $u_2$ because the second predicate `id(u4) < id(u2)` may be true. The point here is that we do not have enough information about $u_4$ and we could not use this global constraint to boost the star matching process.

**This dilemma could be resolved in the join phase if only we have got enough information.** Consider also the case in Figure 2, we got the original pattern graph from the two stars after the join process, and $u_1, u_4$ is the vertex cover. That is to say, a helve $h$ is a vector of two vertices in the data graph that matches $u_1$ and $u_4$ respectively, for example, $h = (v_{10}, v_4)$. Now, the global constraint `id(u1) < id(u2) OR id(u4) < id(u2)` is able to filter the image set of $u_2$ since we have already known that $u_1 \mapsto v_{10}$ and $u_4 \mapsto v_4$, only $u_2$ is the free variable. Formally, a global constraint $\psi(u_1, u_2, \ldots, u_k, u_{k+1})$ is a user defined function on a set of vertices $\{u_1, u_2, \ldots, u_k, u_{k+1}\}$. If $u_1, u_2, \ldots, u_k$ are the roots of the stars, then these vertices are the vertex cover for the new joined graph and the image sets of these vertices form the helve, and $u_{k+1}$ is the only free variable in $\psi$. **So we could use $\psi$ to filter on the image set of $u_{k+1}$, which helps reduce the size of the join result and avoid useless mappings in an early phase.**

However, there are some global constraints that cannot be applied during the join phase, for example, the `id(u2) < id(u3)` we left in Section 3.2.1. As we discussed just before, the constraint have to relate to a vertex cover such that it could be used boost the star matching or join process. This constraint is unique because there are no edges between $u_2$ and $u_3$. **From a practical point of view, it makes sense to provide extra constraints on connected vertices because these connections have concrete meanings, and if a user provide a self defined relationship irrelevant to the topology of the pattern graph, she or he is highly likely to expect it works as deduplicator to remove duplicate results because of the symmetry in the pattern graph**, and this is the case in our example. In fact, as is shown in Figure 6, there is no difference between $u_2$ and $u_3$ in the pattern graph. In the previous subsection we have shown that $u_2$ and $u_3$ share a common image set in the join result, so in fact there is no need to apply the `id(u2) < id(u3)` constraint in the join phase. This kind of constraint could just be used during the decomposition process to obtain the final results.

# 4 EXPERIMENTS

## 4.1 Experiment Environment

*Hardware.*

*Data Set.*

*Queries.*

## 4.2 Comparison with Existing Work

*4.2.1 Supported Features.* Table 2.

*4.2.2 Overall Performance.*

- Compare the runtime with others to show that we have a good overall performance.
- Compare the runtime with different memory limits where many others fail to run to show that we don't have to consume a lot of memory resources.

## 4.3 Study of the Data Graph Scanning

*4.3.1 Study of Data Graph Storage Method.*

- Compare with the conventional CSR/CSC method that stores in-edges and out-edges separately to show that our method is efficient for property graph matching problem.
- Compare with the naive method that do not use the index for VertexIter to show that the index is effective.
- Compare with the naive method that do not use the index for NeighborIter to show that the index is effective.

*4.3.2 Study of Star Matching.*

- Compare with Neo4j and naive method (match and filter on the final results) to show that our local constraint pushdown method is effective.
- Compare with STwig's star decomposition algorithm to show that our algorithm can reduce the size of useless matching results, and also show that this optimization can propagate to the join phase that the CPU and I/O cost of join is also reduced.
- Redirect the matching result to a sink and plot the I/O bandwidth over time to show that the data graph is read sequentially.

## 4.4 Study of Compression

- Compare with naive method that do not compress at all to show that the compression ratio is significant.
- Compare with naive method that do not consider equivalence classes (star matching & join) to show that the equivalence classes can reduce the size of matching results.

| Algorithm | Environment | Direction | Vertex Label | Edge Label | Multigraph | WHERE Clause |
|---|---|---|---|---|---|---|
| opgm | Out-of-core | ✓ | ✓ | ✓ | ✓ | ✓ |
| Neo4j | DBMS | ✓ | ✓ | ✓ | ✓ | ✓ |
| STwig | Distributed | | ✓ | | | |

**Table 2: Comparisons of supported features (TODO).**

- Plot the I/O bandwidth (star matching & join) over time to show that random disk access is avoided.

### 4.5 Study of Join

- Compare with PostgreSQL to show that the join on compressed data is more efficient.
- Compare with hash table set intersection algorithm to show that our join algorithm is more efficient, and it is better to find some data set where the image set is so large that the hash table set interaction is not runnable.
- Compare with naive method that do not use index to show that our index is simple but effective.
- Compare with naive method that do not use global constraint to show that the global constraint pushdown algorithm is effective.
- Redirect the join results to sink and plot the disk bandwidth over time to show that the read of the join operation is sequential.

## 5 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 [30], many subgraph matching algorithms have been proposed using different join order, filter rules, and neighborhood indices [3, 9, 10, 17, 25]. 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 [15, 16, 24, 27, 29]. 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. [29] introduce a star-like basic matching unit called STwig, and implement their subgraph matching algorithm on top of the Trinity [26] memory cloud. Lai et al. [15] propose TwinTwig join using MapReduce, where a TwinTwig is either a single edge or two incident edges of a vertex. The SEED [16] 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* [12, 15, 16, 22, 27, 27, 30]. CBF [22] 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, 9, 24, 25, 29]. $\text{Turbo}_{\text{ISO}}$ [9], 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, 11, 13, 14]. However, all these algorithms only deal with triangulation, a special case of the graph matching problem. Recently, DualSim [12] 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.

# 6 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] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1433–1445. https://doi.org/10.1145/3183713.3190657

[6] Daniel P Friedman, Mitchell Wand, and Christopher Thomas Haynes. 2001. *Essentials of programming languages*. MIT press.

[7] 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

[8] 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

[9] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. 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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6, 1 (2009), 29–123. https://doi.org/10.1080/15427951.2009.10129177

[19] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *PVLDB* 12, 11 (2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

[20] Tijana Milenkovic and Natasa Pržulj. 2008. Uncovering Biological Network Function via Graphlet Degree Signatures. *Cancer Informatics* 6, 1 (2008), 0–0.

[21] Miles Ohlrich, Carl Ebeling, Eka Ginting, and Lisa Sather. 1993. SubGemini: Identifying SubCircuits using a Fast Subgraph Isomorphism Algorithm. In *Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993.*, Alfred E. Dunlop (Ed.). ACM Press, 31–37. https://doi.org/10.1145/157485.164556

[22] 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

[23] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *PVLDB* 11, 4 (2017), 420–431. http://www.vldb.org/pvldb/vol11/p420-sahu.pdf

[24] 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

[25] 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

[26] 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/

[27] 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

[28] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy J. Lin. 2016. GraphJet: Real-Time Content Recommendations at Twitter. *PVLDB* 9, 13 (2016), 1281–1292. https://doi.org/10.14778/3007263.3007267

[29] 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

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