Anti-Vertex for Neighborhood Constraints in Subgraph Queries

Kasra Jamshidi
kjamshid@cs.sfu.ca
School of Computing Science
Simon Fraser University
British Columbia, Canada

Mugilan Mariappan
mmariapp@cs.sfu.ca
School of Computing Science
Simon Fraser University
British Columbia, Canada

Keval Vora
keval@cs.sfu.ca
School of Computing Science
Simon Fraser University
British Columbia, Canada

ABSTRACT

This paper focuses on subgraph queries where constraints are present in the neighborhood of the explored subgraphs. We describe *anti-vertex*, a declarative construct that indicates absence of a vertex, i.e., the resulting subgraph should not have a vertex in its specified neighborhood that matches the anti-vertex. We formalize the semantics of anti-vertex to benefit from automatic reasoning and optimization, and to enable standardized implementation across query languages and runtimes. The semantics are defined for various matching semantics that are commonly employed in subgraph querying (isomorphism, homomorphism, and no-repeated-edge matching) and for the widely adopted property graph model. We illustrate several examples where anti-vertices can be employed to help familiarize with the anti-vertex concept. We further showcase how anti-vertex support can be added in existing graph query languages by developing prototype extensions of Cypher language. Finally, we study how anti-vertices interact with the symmetry breaking technique in subgraph matching frameworks so that their meaning remains consistent with the expected outcome of constrained neighborhoods to connected vertices.

ACM Reference Format:
Kasra Jamshidi, Mugilan Mariappan, and Keval Vora. 2022. Anti-Vertex for Neighborhood Constraints in Subgraph Queries. In *Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES & NDA'22), June 12, 2022, Philadelphia, P A, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3534540.3534690

1 INTRODUCTION

Subgraph queries are an important class of queries supported by graph databases. These queries find instances of query graphs in an input graph (also called data graph). Query graphs are well-defined structures that constrain the vertices and edges in matching subgraphs, and how they should be connected to each other. However, queries often require constraints on the *neighborhood* of the subgraphs, which are difficult to express in existing models for graph queries.

In this paper, we describe *anti-vertex*, a special kind of vertex that constrains how subgraphs are connected to the rest of the data graph. Specifically, an anti-vertex indicates absence of a vertex, i.e., there should be no vertex in the data graph that matches the anti-vertex in the neighborhood of specified matched vertices.

The declarative nature of the anti-vertex construct simplifies expressing the constraint on subgraph neighborhoods for several applications. For example, identifying whether a clique is maximal simply requires connecting an anti-vertex to all other vertices in the clique query graph. In this case, the anti-vertex will ensure that vertices in each matched subgraph do not have a common neighbor, thereby guaranteeing maximality of the explored cliques. To help develop the intuition for anti-vertex, as its first contribution this paper identifies several examples where anti-vertices can be employed.

The strength of any construct lies in its semantics being well-defined, as it not only enables a standard implementation across different languages and runtimes, but also allows automatic reasoning and optimization for efficient query evaluation. The second contribution is formalizing the semantics of anti-vertex. We observe that isomorphism, homomorphism and no-repeated-edge matching semantics [5] are commonly employed in graph querying solutions [4, 17, 21, 23, 41]. We formalize the anti-vertex under all three matching semantics, and generalize for the widely used property graph model [5]. The semantics are carefully developed to ensure the meaning of anti-vertex remains consistent with the matching semantics, i.e., it varies across the different matching semantics.

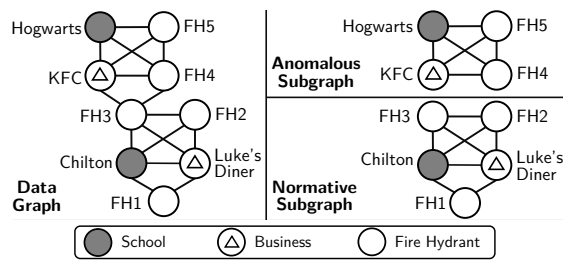
The third contribution of this paper is studying the interaction of anti-vertex construct with graph query languages and subgraph matching frameworks. This includes two components:

- The anti-vertex construct can be employed in graph query languages such as Cypher [17], GSQL [12], and GQL [11]. This paper develops prototype extensions of Cypher's pattern matching syntax in order to demonstrate how anti-vertices can be incorporated in Cypher without impacting its existing capabilities. The anti-vertex enhancement in Cypher allows declaratively writing the absence of vertex in the subgraph neighborhoods, which is difficult to express otherwise.
- Subgraph matching frameworks [36, 42] often avoid generating duplicate subgraphs by employing symmetry breaking [18]. Since anti-vertex imposes structural constraint in the query graph, we further study how anti-vertices should participate in symmetry breaking in order to ensure subgraph results that are consistent with the intuition of constrained neighborhoods.

Finally, the paper discusses a potential future extension to anti-vertex that enforces constraints on neighborhoods in terms of absence of paths and absence of patterns instead of vertices alone. We envision the final semantics of anti-vertices would naturally capture such extensions, making graph query languages more expressive and powerful.

GRADES & NDA'22, June 12, 2022, Philadelphia, PA, USA

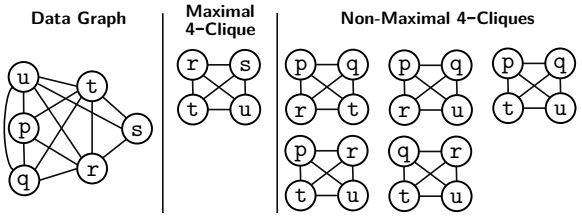
ACM ISBN 978-1-4503-9384-3/22/06…$15.00
https://doi.org/10.1145/3534540.3534690




```
MATCH (a:SCHOOL)--(b:BUSINESS),
      (a)--(c:FIRE_HYDRANT)--(b),
      (a)--(d:FIRE_HYDRANT)--(b)
WHERE NOT EXISTS {
  MATCH (a:SCHOOL)--(b:BUSINESS),
        (a)--(c:FIRE_HYDRANT)--(b),
        (a)--(d:FIRE_HYDRANT)--(b),
        (a)--(e:FIRE_HYDRANT)--(b)
}
RETURN a, b, c, d
```

```
MATCH (a:SCHOOL)--(b:BUSINESS),
      (a)--(c:FIRE_HYDRANT)--(b),
      (a)--(d:FIRE_HYDRANT)--(b)
WHERE NOT EXISTS {
   MATCH (a)--(e:FIRE_HYDRANT)--(b)
   WHERE e<>c
     AND e<>d
}
RETURN a, b, c, d
```

Figure 1: Anomaly detection use case. The anomalous subgraph of interest is the one where the school and the business are connected with two fire hydrants and not three. Cypher queries to find anomalous subgraphs shown on right.




```
MATCH (a)--(b), (a)--(c), (a)--(d),
      (b)--(c), (b)--(d), (c)--(d)
WHERE NOT EXISTS{
  MATCH (a)--(b), (a)--(c), (a)--(d),
        (b)--(c), (b)--(d), (c)--(d),
        (a)--(e), (b)--(e),
        (c)--(e), (d)--(e)
}
RETURN a, b, c, d
```

```
MATCH (a)--(b), (a)--(c), (a)--(d),
      (b)--(c), (b)--(d), (c)--(d)
WHERE NOT EXISTS{
  MATCH (a)--(e), (b)--(e),
        (c)--(e), (d)--(e)
  WHERE e<>a AND e<>b
    AND e<>c AND e<>d
}
RETURN a, b, c, d
```

Figure 2: Maximal cliques use case. Only clique r-s-t-u is a maximal size-4 clique since all other size-4 cliques are part of larger size-5 clique p-q-r-t-u. Cypher queries to find maximal 4-cliques shown on right.

2 USE CASES

In this section, we present motivating use cases that constrain the subgraphs of interest based on their neighborhoods. We also demonstrate how such constrained queries can be currently expressed in the Cypher language to argue the need for an easily expressible construct that indicates absence of vertices.

Example 2.1. Consider the use cases below.

(1) *Anomaly Detection.* Identifying anomalies in graph data [33] is crucial across various domains. Certain anomalies are identified as subgraphs that have missing vertices from a reference (normative or non-anomalous) subgraph [13]. Figure 1 shows an example of a city planning scenario. One of the planning requirements is if there is a school and a business close to each other, then there must be at least three fire hydrants nearby that are useful for both locations. For the graph shown in Figure 1, the subgraph with Chilton and Luke's Diner satisfies the allocation requirement because of fire hydrants FH1, FH2 and FH3. However, the subgraph with Hogwarts and KFC is anomalous since there are only two fire hydrants FH4 and FH5.
Finding subgraphs with exactly two fire hydrants is not straightforward. There are two obvious approaches to writing a Cypher query for this problem, both shown in Figure 1. In the first query, the problem is reformulated as matching subgraphs with two fire hydrants that are not part of subgraphs containing three fire hydrants. Expressing the absence of the third fire hydrant in such an indirect fashion causes tedious repetition and increase in query sizes. This not only makes it challenging to read and manage those queries (e.g., incrementally adjust to add new constraints), but also makes the process of writing complex queries (e.g., with multiple constraints) error-prone. The second query incurs less repetition in the subquery than the first approach, but requires users to specify additional constraints against the outer query to achieve the desired semantics. Determining the correct constraints to ensure the subquery does not filter too many or too few subgraphs is difficult in larger queries, as users must visualize how their query will be matched against complex graph structures. Both approaches lead to larger, less declarative queries involving error-prone subqueries. Instead, *the absence of another fire hydrant neighbor* can be directly expressed using an anti-vertex.

(2) *Maximal Cliques.* Finding and enumerating maximal cliques is a popular graph mining problem, with applications in social network analysis, financial analysis, security and biology [8]. In Figure 2, the clique r-s-t-u is maximal, but the other cliques are not maximal since they all can be extended into the larger clique p-q-r-t-u by adding a vertex. While cliques of a certain size can be easily expressed as a Cypher query, the maximality requirement can again be expressed in two ways. The first query in Figure 2 reformulates the problem as finding cliques of size k that are not contained inside cliques of size $k+1$, while the second query finds vertices adjacent but not equal to all k previously matched vertices. Since the maximality constraint simply limits the vertices in cliques to not have a common neighboring vertex, *the absence of this common neighboring vertex* can be directly expressed using an anti-vertex.

(3) *Approximate Subgraph Matching.* Approximate subgraph matching often allows optional and forbidden vertices and edges [43] to provide a loose subgraph template for which subgraphs are matched. As subgraphs get matched for approximate templates, identifying which subgraphs result due to the vertices being optional requires adding a constraint for the vertex to be absent. Such a constraint indicating *absence of vertices* can be achieved using anti-vertices.

(4) *Contrasting Quasi-Cliques.* Recent research [1] on mining for multigraphs argues the strength of finding collection of vertices that are dense in one graph but less connected in a second

graph. An interesting sub-case is mining contrasting quasi-cliques where the sparser subgraph is fully imposed on a subset of vertices, i.e., remaining vertices are not connected to the subgraph. Here, *the isolated vertices from the rest of the subgraph* can be represented using anti-vertices.

The above examples showcase the need for easily expressing absence of vertex connections in the neighborhood of explored subgraphs. A well-defined declarative construct would also enable thorough reasoning about correctness as well as methodical exploration of useful optimizations.

3 ANTI-VERTEX: CONCEPT AND SEMANTICS

This section develops the concept of the anti-vertex. Section 3.1 illustrates the semantics of anti-vertices by means of sample use cases. Then, Section 3.2 establishes our data model and terminology, and Section 3.3 formally defines the semantics of anti-vertices.

3.1 What is an Anti-Vertex?

An anti-vertex is a vertex in the query graph that indicates absence of a vertex in the resulting subgraph. Anti-vertices allow users to express constraints on the neighborhoods of subgraph vertices declaratively, simply by describing which vertices are undesirable.

To easily visualize an anti-vertex in the query graph, anti-vertices are pictorially represented as vertices with dashed borders as opposed to solid borders used for regular vertices. We demonstrate how anti-vertices can be employed in our use cases to help develop the intuition.

Example 3.1. Anti-vertices simplify the expression of the anomaly detection and maximal cliques use cases from Example 2.1.

(1) *Anomaly Detection.* To find anomalous subgraphs with two fire hydrants, a normative subgraph can easily be turned into a query graph by marking a fire hydrant as an anti-vertex. Figure 3a shows the query graph containing an anti-vertex that exactly returns the anomalous subgraph containing FH4, FH5 without returning the normative subgraph involving FH1, FH2, FH3. The anti-vertex (indicated by the dotted border) requires that any school and business matched by the query do not have a third fire hydrant in their neighborhood. In this case, the anti-vertex provides a declarative way to express a constraint on the shared neighborhood of nearby schools and businesses.
(2) *Maximal Cliques.* A maximal clique of size k can be directly expressed using an anti-vertex by expressing a clique of size $k+1$ and marking one of the vertices as an anti-vertex. As shown in Figure 3b, the query graph contains an anti-vertex connected to all the vertices of a 4-clique. This eliminates all the non-maximal 4-cliques from the result set, while still returning the maximal r-s-t-u clique.

As demonstrated in the above examples, the anti-vertex is a declarative construct that allows users to express constraints simply in terms of which results are not desired. Definitions are established next, before formal discussion of the anti-vertex semantics.

(a) Anomaly detection. Third fire hydrant marked as anti-vertex. **(b) Maximal 4-cliques. Fifth vertex in the clique marked as anti-vertex.**

Figure 3: Use cases with anti-vertex.

3.2 Preliminaries

Graph Model. When storing and processing graph-structured data it is often convenient to consider not only the structural information encoded by connections between vertices, but also the myriad information associated with each vertex and edge. Two popular models for rich graph data are the Resource Description Framework (RDF) [10] and property graphs [5]. Property graphs can model more complex structures than RDF by allowing edges and vertices to be associated with arbitrary key-value pairs called properties. As a result, property graphs have gained widespread adoption by both commercial and academic graph databases [4, 11, 17, 41].

We develop our graph model based on the property graph model. Though we define a similar graph model, this paper uses simple undirected graphs in order to ease exposition, since accounting for edge directions and properties is trivial but involves cumbersome notation. Descriptions of how definitions and semantics translate to property graphs are provided where it is not obvious.

Let $\mathcal{I} \subseteq \mathbb{Z}$ be a set of integer identifiers, and let $\mathcal{L}$ and $\mathcal{T}$ be sets of vertex labels and edge types, respectively. An *undirected graph* is a tuple $G = \langle V, E, \lambda, \tau \rangle$ where:

- $V \subseteq \mathcal{I}$ are the vertices of G.
- $E \subset V \times V$ are the edges of G, consisting of unordered pairs of distinct vertices.
- $\lambda : V \rightarrow 2^{\mathcal{L}}$ maps each vertex to a (possibly empty) set of labels.
- $\tau : E \rightarrow \mathcal{T}$ maps each edge to an edge type.

Let $G = \langle V_G, E_G, \lambda_G, \tau_G \rangle$ and $H = \langle V_H, E_H, \lambda_H, \tau_H \rangle$ be graphs. For $v \in V_G,\ e \in E_H,\ u \in V_H$, the (e,u)-*neighborhood* of v is the set of vertices in V_G which contain the labels of u and are adjacent to v via edges like e. Formally, we write the (e,u)-neighborhood of v as

$$\mathcal{N}(v,e,u) = \{v' \in V_G : (v,v') \in E_G \wedge \tau_H(e) = \tau_G((v,v')) \wedge \lambda_H(u) \subseteq \lambda_G(v')\}$$

Next, we clearly separate anti-vertices in our model. A *query graph* is a graph where a proper subset of vertices $V^- \subset V$ are distinguished as *anti-vertices*. Currently we assume two anti-vertices cannot form both endpoints of an edge, but we envision the semantics will evolve to a point where this assumption is not enforced [1]. For convenience, we introduce notation to partition the edges and

[1] Since anti-vertex simply captures absence of vertices, we envision that anti-vertices can be used to model paths of absent vertices (e.g., querying vertices separated by two hops), which will require edges between two anti-vertices. However, the semantics of anti-vertices with such extended capability need to be thoroughly defined, which is left for future work (see Section 8).

vertices of the query graph into the parts with and without anti-vertices:

$$V^+ = V \setminus V^-$$
$$E^+ = \{(u,v) \in E : u, v \in V^+\}$$
$$E^- = E \setminus E^+$$

Throughout the paper, we will refer to $G = \langle V_G, E_G, \lambda_G, \tau_G \rangle$ as the data graph and $P = \langle V_P, E_P, \lambda_P, \tau_P \rangle$ as the query graph.

Finally, we define property graph based on the definition in [17], while including anti-vertices. A *property graph* is a data graph or query graph where $E \subseteq \mathcal{I} \setminus V$, and there are additional functions $\rho : E \to V \times V$ and $\pi : V \cup E \to 2^{\mathcal{K} \times \mathcal{V}}$, where $\mathcal{K}$ and $\mathcal{V}$ are sets of property keys and values, respectively. ρ maps each edge to an ordered pair of endpoints, so that a pair of vertices can have multiple edges between them. The definition of the (e,u)-neighborhood of a vertex v generalizes easily to property graphs by using ρ to obtain all edges involving v and by considering the properties of e and u in addition to their type and labels.

Subgraph Matching. A *match* is a mapping $m : V_P^+ \cup E_P^+ \to V_G \cup E_G$ from the vertices and edges of P to the vertices and edges of G which satisfies two kinds of constraints: (a) structural constraints indicating how edges and vertices are connected; and (b) non-structural constraints regarding vertex labels and edge types of P. The non-structural constraints are defined as:

$$\forall v \in V_P^+, \lambda_P(v) \subseteq \lambda_G(m(v))$$
$$\wedge\ \forall e \in E_P^+, \tau_P(e) = \tau_G(m(e))$$

The structural constraints, on the other hand, are dependent on the underlying matching semantics. We consider three matching semantics defined by [5]: *isomorphism* (used in [21, 23]), *homomorphism* (used in [4, 41]) and *no-repeated-edge* (used in [17]).

- *Homomorphism*: Any mapping from P to G that preserves edge relationships is a homomorphism. Formally, a match m is a homomorphism if it satisfies:

$$\forall (u,v) \in E_P^+, (m(u), m(v)) = m((u,v))$$

- *No-Repeated-Edge*: In addition to preserving edge relationships, m must be an injective mapping with respect to the edges of G in no-repeated-edge semantics. While the same vertex in G may appear multiple times as part of the match, m never maps two different pattern edges to the same data edge. A match m satisfies no-repeated-edge semantics if:

$$\forall (u,v) \in E_P^+, (m(u), m(v)) = m((u,v))$$
$$\wedge\ \forall e_1, e_2 \in E_P^+, m(e_1) = m(e_2) \implies e_1 = e_2$$

- *Isomorphism*: Isomorphism semantics require that m is injective with respect to both vertices and edges, so that every data vertex/edge in the range of m is mapped from exactly one query vertex/edge in the domain. In other words, every vertex and edge matched by m is distinct. Formally, a match m is an isomorphism if:

$$\forall (u,v) \in E_P^+, (m(u), m(v)) = m((u,v))$$
$$\wedge\ \forall e_1, e_2 \in E_P^+, m(e_1) = m(e_2) \implies e_1 = e_2$$
$$\wedge\ \forall v_1, v_2 \in V_P^+, m(v_1) = m(v_2) \implies v_1 = v_2$$

We abuse the function notation to allow sets as well, e.g., for a vertex or edge subset S, $m(S) = \{m(x) : x \in S\}$.

These definitions can be easily extended to property graphs by also requiring that properties on vertices and edges be satisfied by the mapping, in a similar way to vertex labels, and using ρ to obtain edge endpoints.

3.3 Formal Semantics

Anti-vertices encode an additional requirement on matches, namely that an anti-vertex should not be possible to match.

Suppose m is a match for all edges and standard vertices of a query graph P. Let $C : V_P^- \to 2^{V_G}$ be a function which returns the set of data vertices that can be mapped to by m from a query anti-vertex. The match m is valid only if $\forall \overline{u} \in V_P^-, C(\overline{u}) = \emptyset$. Hence, an anti-vertex will invalidate a match if there are vertices in the data graph which can be mapped to it.

The definition of C depends on the underlying subgraph matching semantics (homomorphism, no-repeated-edge, isomorphism), as described next. To ensure the semantics of anti-vertices is consistent with the matching semantics, we define C by adhering to the requirements on m which allow or disallow multiple different pattern vertices/edges to be mapped to the same vertices/edges.

Let $P = \langle V_P, E_P, \lambda_P, \tau_P \rangle$ be a query graph, $G = \langle V_G, E_G, \lambda_G, \tau_G \rangle$ be the data graph, and m be a match for P in G.

Homomorphism. In homomorphism semantics, m only needs to satisfy the vertex labels and edge types, and preserve edge relationships. Hence, any data vertex with the correct edges and labels can fulfill the anti-vertex requirement and invalidate the match, including previously mapped vertices. C is defined as follows.

$$C(\overline{u}) = \bigcap_{v:(\overline{u},v) \in E_P} \mathcal{N}(m(v), (\overline{u}, v), \overline{u})$$

No-Repeated-Edge. No-repeated-edge semantics requires that m provide an injective mapping from edges in P to edges in G. Hence, for anti-vertices, the data edges that are already mapped by m cannot invalidate the match, but vertices from m can satisfy the anti-vertex requirement (i.e., allow repeated vertices). C is defined as follows.

$$C(\overline{u}) = \bigcap_{v:(\overline{u},v) \in E_P} \mathcal{N}(m(v), (\overline{u}, v), \overline{u}) \setminus m(\mathcal{N}(v, (\overline{u}, v), \overline{u}))$$

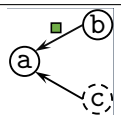
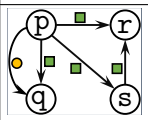
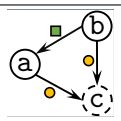
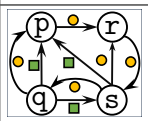
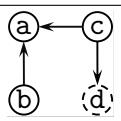
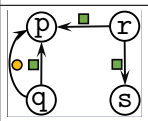
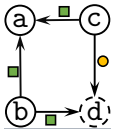
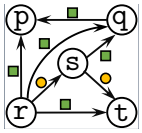
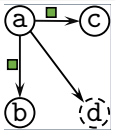
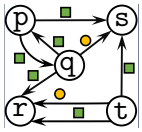
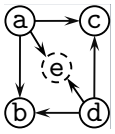
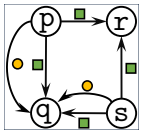
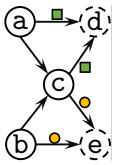
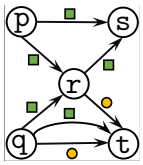
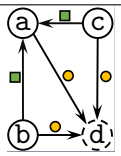
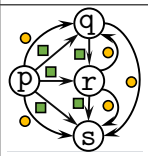
Isomorphism. In isomorphism semantics, m must be injective with respect to both edges and vertices, with no repetition. Hence, data vertices already mapped to by m (and implicitly, the edges incident on those vertices) cannot fulfill the anti-vertex requirement to invalidate the match. C is defined as follows.

$$C(\overline{u}) = \bigcap_{v:(\overline{u},v) \in E_P} \mathcal{N}(m(v), (\overline{u}, v), \overline{u}) \setminus m(V_P)$$

3.4 Generalization for Property Graphs

The above semantics naturally generalize to property graphs. In a property graph, each anti-vertex $\overline{u}$ can be incident on multiple edges with directions. Thus, to transform the previous definitions of $C(\overline{u})$ to fit property graphs, it suffices to intersect the $(e, \overline{u})$-neighborhoods of v for every $e \in E_P$ where $\rho(e) = (\overline{u}, v)$ or $\rho(e) = (v, \overline{u})$, and perform the same set differences.

Table 1: Examples with anti-vertices.

No.	Query Graph	Data Graph	Description (w.r.t. Isomorphism)	Subgraph Results		Notes
				Isomorphism	No-Repeated-Edge	
Q1			Find a,b such that i) b FOLLOWs a, and ii) a is not FOLLOWed/LIKEd by another PERSON	a b s p q p	a b s p	(q,p) is not a match in no-repeated-edge semantics
Q2			Find a,b such that i) b FOLLOWs a, and ii) a and b do not LIKE another common PERSON	a b p q s q	a b p q s q	(p,s) is not a match since p,s both LIKE r (s,q) is a match since only q LIKEs p
Q3			Find a,b,c such that i) b and c FOLLOW/LIKE a, and ii) c does not FOLLOW/LIKE another PERSON	a b c p r q	a b c p q q	(p,q,r) is not a match since r FOLLOWs s
Q4			Find a,b,c such that i) b and c FOLLOW a, and ii) b does not FOLLOW d, and iii) c does not LIKE d	a b c p q r p r q q s r	a b c p q r p r q q s r	(q,r,s) is not a match since r FOLLOWs t and s LIKEs t
Q5			Find a,b,c such that i) a FOLLOWs b and c, and ii) a does not FOLLOW/LIKE another PERSON	a b c p q s p s q t s r t r s	a b c p q s p s q	(t,s,r) and (t,r,s) are not matches with no-repeated-edge semantics since t LIKEs and FOLLOWs r
Q6			Find a,b,c,d such that i) a and d FOLLOW/LIKE b and c, and ii) a and d do not FOLLOW/LIKE another common PERSON	a b c d p q r s p r q s s q r p s r q p	a b c d - - - -	No matches with no-repeated-edge semantics since p and s LIKE and FOLLOW q
Q7			Find a,b,c such that i) a and b FOLLOW/LIKE c, and ii) a and c do not FOLLOW another common PERSON , and iii) b and c do not LIKE another common PERSON	a b c p r s r p s q r t r q t q p r	a b c p r s r p s q r t r q t q p r q q t	(p,q,r) is not a match since p and r both FOLLOW s, and q and r both LIKE t
Q8			Find a,b,c such that i) b and c FOLLOW a, and ii) the three do not LIKE a common PERSON	a b c s p r s r p	a b c s p r s r p	(r,p,q) is not a match since r,p,q LIKE s. (s,p,r) is a match since r,p LIKE q but s does not LIKE q

Let G and P be property graphs. The semantics with isomorphism in property graphs can be expressed by defining C as follows.

$$C(\overline{u}) = \bigcap_{\substack{e \in E_P: \\ \rho(e)=(\overline{u},v)}} \mathcal{N}(m(v), e, \overline{u}) \cap \bigcap_{\substack{e \in E_P: \\ \rho(e)=(v,\overline{u})}} \mathcal{N}(m(v), e, \overline{u}) \setminus m(V_P)$$

Similarly, semantics of anti-vertices with homomorphism and no-repeated-edge semantics in property graphs can be defined by translating the definitions from Section 3.3.

4 MORE EXAMPLES

Table 1 shows various subgraph queries where anti-vertices are used in different ways. The data graphs capture social network information where vertices represent people and edges represent LIKES and FOLLOWS relationships. The query graphs contain anti-vertices, and their textual description is provided to help familiarize with the concept by demonstrating how anti-vertices are perceived for social network analysis. For isomorphism and no-repeated-edge matching semantics, the resulting mappings between query vertices and data vertices are shown in relational format.

```
                pattern  ::= pattern° | a = pattern°
               pattern°  ::= node_pattern
                           | node_pattern rel_pattern pattern°
▷                          | node_pattern rel_pattern pattern+
▷                          | pattern+ rel_pattern pattern°
▷             pattern+   ::= anti_node_pattern
▷                          | anti_node_pattern rel_pattern pattern°
          node_pattern   ::= ( a? label_list? map? )
▷  anti_node_pattern   ::= ( ! a? label_list? map? )

   rel_pattern   ::= -[ a? type_list? len? ]->
                   | <-[ a? type_list? len? ]-
                   | -[ a? type_list? len? ]-
   label_list    ::= : l | : l label_list
   map           ::= { prop_list }
   prop_list     ::= k : expr | k : expr, prop_list
   type_list     ::= : t | type_list | t
   len           ::= * | *d | *d1 . . | *. . d2 | *d1 . . d2
                     d, d1, d2 ∈ ℕ
```

Figure 4: Syntax of Cypher patterns with anti-vertex. Enhancements to the original grammar are marked with ▷.

5 ANTI-VERTEX IN CYPHER

The anti-vertex construct can be incorporated in existing graph query languages to express the complex anti-vertex queries as easily as standard subgraph queries. Recent graph query languages [4, 11, 17, 41] integrate declarative "ASCII-art" pattern expressions with familiar SQL constructs. In particular, Cypher [17] is a popular graph query language, used in both academic and commercial graph databases including Neo4j [32], Amazon Neptune [2], and GraphFlow [26]. In this section we develop prototype extensions to Cypher's pattern matching syntax to support anti-vertices.

Cypher pattern matching syntax allows node patterns, relationship patterns, and path patterns. Since anti-vertices express absence of neighbors, they will be best expressed using relationship patterns and path patterns. However, the anti-vertex semantics developed in this paper do not consider the case where two anti-vertices are connected via an edge (recall the assumption in Section 3.2). This leaves the semantics of fixed/variable-length path fragments containing anti-vertices ambiguous. While such semantics are left for future work (Section 8), we envision the grammar to be able to support arbitrary path patterns with anti-vertices.

Hence, we develop two prototype extensions to Cypher's pattern matching syntax: one that allows arbitrary path patterns with anti-vertices (presented in Section 5.1), and one that limits the grammar to the anti-vertex semantics defined in this paper (presented in Section 5.2). Allowing arbitrary path patterns requires fewer changes to the original grammar, and thus easier implementation and validation in existing query engines, at the cost of some ambiguity regarding the semantics of anti-vertices in fixed/variable-length path patterns. Meanwhile, keeping the grammar limited requires more complex changes to the original grammar, but has no ambiguity, and provides flexibility for future work to define the semantics of path patterns involving an anti-vertex consistently (i.e., handle paths with one or both endpoints being an anti-vertex consistently).

5.1 Grammar with Arbitrary Path Patterns

Following the same notation as Cypher, Figure 4 shows an extended pattern matching grammar that supports anti-vertices (extensions added for anti-vertex support are marked with ▷). This syntax only applies within `MATCH` clauses of Cypher; the remainder of Cypher's syntax is unaffected.

An anti-vertex is defined by the anti_node_pattern construct, which is identical to node_pattern from the original Cypher grammar, but marked with a ! symbol [2]. This construct only appears in pattern+ either by itself or accompanied by rel_pattern. We compose these fragments with Cypher's original pattern definition to allow as much programmer flexibility as possible.

Intuitively, the syntax allows an anti-vertex to be present:

(1) at the beginning of the pattern:

```
(!a)--
```

(2) in the middle of the pattern:

```
--(!a)--
```

(3) at the end of the pattern:

```
--(!a)
```

Since we utilize rel_pattern as defined in the original grammar, fragments of path patterns with anti-vertices can be expressed in this grammar. For example, the following are allowed:

```
(a)-[*3]-(!b)

(!a)--(b)

()-[*2]-(!a)-[*2]-()
```

The pattern+ separates the anti_node_pattern from pattern°, which disallows anti-vertices at both endpoints of a relationship.

5.2 Grammar without Arbitrary Path Patterns

Here we limit the syntax to only express anti-vertices where semantics are well-defined in this paper. Figure 5 shows the extended pattern matching grammar for this case.

The simple_rel_pattern is added to ensure path fragments containing anti-vertices are not length-based (fixed or variable). simple_rel_pattern is simply rel_pattern without a len parameter. The anti_node_pattern construct only appears in pre_pattern and post_pattern, accompanied by simple_rel_pattern. Intuitively, pre_pattern represents the syntax fragment

```
(!a)--
```

and post_pattern represents the syntax fragments

```
--(!a)    and    --(!a)--
```

These fragments are composed with Cypher's original pattern definition to allow as much programmer flexibility as possible.

A pattern either begins with an anti-vertex (through pre_pattern) or a standard vertex (through pattern°). Anti-vertices in the middle or at the end of a pattern are supported through mutual recursion between pattern° and post_pattern. Two anti-vertices will never form both endpoints of a relationship because pre_pattern never occurs directly before post_pattern.

[2] The ! symbol typically denotes *not* operator in programming languages, which fits the meaning for anti-vertex (data vertex *not* present).

```
▷          pattern ::= pattern+ | a = pattern+
▷         pattern+ ::= pattern°| pre_pattern
▷      pre_pattern ::= anti_node_pattern simple_rel_pattern pattern°
          pattern° ::= node_pattern
                     | node_pattern rel_pattern pattern°
▷                    | node_pattern simple_rel_pattern post_pattern
▷     post_pattern ::= anti_node_pattern
▷                    | anti_node_pattern simple_rel_pattern pattern°
▷ anti_node_pattern ::= ( ! a? label_list? map? )
▷ simple_rel_pattern ::= -[ a? type_list? ]->
▷                    | <-[ a? type_list? ]- | -[ a? type_list? ]-

node_pattern ::= ( a? label_list? map? )
rel_pattern  ::= -[ a? type_list? len? ]->
               | <-[ a? type_list? len? ]-
               | -[ a? type_list? len? ]-
label_list   ::= : l | : l label_list
map          ::= { prop_list }
prop_list    ::= k : expr | k : expr, prop_list
type_list    ::= : t | type_list | t
len          ::= * | *d | *d1. . | * . . d2
               | *d1 . . d2
                    d, d1, d2 ∈ ℕ
```

Figure 5: Syntax of Cypher patterns with anti-vertex, without arbitrary path patterns. Enhancements marked with ▷.

While fixed/variable length path fragments with anti-vertex are disallowed, regular fixed/variable length path fragments containing node_pattern can still be expressed (same as defined in original Cypher grammar). For example, the following are allowed:

```
(a)-[*2]-()--(!b)
(!a)--(b)
(a)--(!b)--(c)
```

5.3 Examples with Cypher

We revisit the use cases from Section 2 to demonstrate how they can be easily expressed using the extended Cypher grammar.

Figure 6 and Figure 7 show the example Cypher queries in Figure 1 and Figure 2 rewritten declaratively with this syntax.

Example 5.1. Consider the anomaly detection and maximal cliques use cases from Example 2.1.

(1) *Anomaly Detection.* The Cypher query with anti-vertex for anomaly detection is shown in Figure 6. Instead of a long subquery which repeats most of the initial `MATCH` clause, or one that must explicitly specify the matching semantics, the query directly expresses the anomalous subgraph using an anti-vertex to denote absence of a third fire hydrant. Anti-vertices are expressed similarly to standard vertices, including specifying labels and properties.

(2) *Maximal Cliques.* Figure 7 shows the Cypher query with anti-vertex for finding maximal cliques of size 4. Previously without anti-vertex support (shown in Figure 2), the constraints induced by the matching semantics on the subgraph were explicitly enforced in a `WHERE` clause. Now, the query directly expresses the maximality constraint by connecting all vertices to an anti-vertex, guaranteeing consistency with the underlying matching semantics. The anti-vertex e is unconstrained, thus if any data vertex can be mapped to e (i.e., there is a vertex adjacent to the matches for all of a,b,c,d), then the subgraph will be discarded, as matching e would create a clique of size 5.

6 ANTI-VERTEX AND SYMMETRY BREAKING

Subgraph matching frameworks operating under isomorphism semantics [24, 36, 42] often find unique subgraphs and avoid generating automorphic subgraphs (subgraphs that are isomorphic to each other). They achieve this by employing a technique known as *symmetry breaking* [18] that ensures a given set of data graph vertices

```
MATCH (a:SCHOOL)--(b:BUSINESS),
      (a)--(fh1:FIRE_HYDRANT)--(b),
      (a)--(fh2:FIRE_HYDRANT)--(b),
      (a)--(!fh3:FIRE_HYDRANT)--(b)
RETURN a, b
```

Figure 6: Anomaly detection using anti-vertex in Cypher.

```
MATCH (a)--(b), (a)--(c), (a)--(d),
      (b)--(c), (b)--(d), (c)--(d),
      (a)--(!e), (b)--(!e), (c)--(!e), (d)--(!e)
RETURN a, b, c, d
```

Figure 7: Maximal 4-clique using anti-vertex in Cypher.

is only matched once to the query graph. This section studies how anti-vertices in the query graph interact with symmetry breaking.

Symmetry breaking works by enforcing an ordering on the ids of vertices which match symmetric sets of query graph vertices. The following example illustrates how symmetry breaking avoids duplicate subgraphs.

Example 6.1. Consider an undirected, unlabeled 3-vertex path a-b-c as the query graph. The query vertices a and c are symmetric: they are both constrained only by an edge to b. As a result, any data vertex matched to a can also be matched to c, hence resulting in automorphic (duplicate) subgraphs. Symmetry breaking enforces an additional constraint on a match m, e.g. $m(\mathsf{a}) < m(\mathsf{b})$, where $m(\mathsf{a})$ is the identifier for the data vertex mapped to a. Now, due to the isomorphism semantics requiring that all data vertices involved in the match are distinct, the matches for a and c cannot swap roles, hence ensuring unique subgraph results.

Intuitively, an anti-vertex can be thought of as an additional constraint on matches. We advocate that anti-vertices should be considered when breaking symmetries to maximize flexibility. This is because disregarding anti-vertices during symmetry breaking can yield unintuitive results as it becomes equivalent to adding symmetric anti-vertices. We illustrate this case below.

Example 6.2. Consider Q3 in Table 1. When disregarding the anti-vertex d, query graph vertices b and c are symmetric, which induces the ordering $m(\mathsf{b}) < m(\mathsf{c})$ on any match m. This means the sole match p-r-q under isomorphism semantics would be discarded, since $m(\mathsf{b}) = r > q = m(\mathsf{c})$. Such a result would be expected if the query graph has a new anti-vertex adjacent to b.

When d participates in symmetry breaking, b and c are no longer symmetric since b is not adjacent to an anti-vertex while c is. Hence, no ordering constraint is enforced, which yields the match p-r-q.

By considering anti-vertices when breaking symmetries, the resulting subgraphs are intuitive as they align with the basic ideology of constraining the neighborhood of matched vertices. Moreover, this also retains flexibility for cases where the anti-vertex constraint needs to be enforced on symmetric vertices in the match; this can be achieved by simply adding symmetric anti-vertices.

7 RELATED WORK

Graph Query Languages. Graph query languages and their data models have been extensively researched [5]. SPARQL [22] is one of the first graph query languages to provide pattern matching alongside SQL constructs, and operates on sets of RDF triples. It can support property graphs through [40], which translates SPARQL queries into Gremlin queries. Cypher [17] is a query language on property graphs first developed as part of Neo4j [32] that introduced "ASCII-art" syntax to specify path patterns. PGQL [41] offers regular path expressions in the pattern matching syntax, and introduces novel operators to construct new graphs as the result of a query. G-CORE [4] proposes a new graph query language using similar syntax to Cypher and PGQL, but operating in the *path property graph* data model, where paths are treated as a first-class entity with labels and properties. GSQL [12] allows for computing aggregate values from the results of graph queries for sophisticated graph analytics. GQL [11] is a recent effort to create a standard graph query language for property graphs. It provides several novel constructs for query expression, such as partial edge direction restrictions and edge predicates. Gremlin [37] is a functional graph traversal language with a simple grammar meant to facilitate embedding within a general-purpose programming language. Unlike the SQL-like syntax, Gremlin users define queries as trees of functions through method-chaining in a host language.

These query languages expose syntax and operators for specifying edges, paths, and constraints on query results, but cannot easily express neighborhood constraints. The anti-vertex construct provides a declarative method for specifying neighborhood constraints. Anti-vertex is a generic concept and can be incorporated in any modern query language in a similar fashion to our proposed extensions to Cypher.

Querying Constructs. There has also been work on subgraph query models and programming constructs for subgraph queries.

Anti-vertex originated from Peregrine [24, 25], where it was used in simple undirected graphs without edge labels. Due to lack of formal semantics, the behavior remained ambiguous when extended to general graph models like the property graph model with different matching semantics. By contrast, this paper formalizes anti-vertex semantics consistent with all three commonly used matching semantics, under the property graph model where graphs can have directions and edge labels. We also study how native anti-vertex support can be added in existing graph query languages by extending the Cypher language and analyze how anti-vertices should participate in symmetry breaking.

[16] allows expressing functional dependencies on graphs (GFD). GFDs cannot be used to implement the anti-vertex construct, because they only constrain vertices within a match, without access to the surrounding data graph. [15] proposes the concept of conditional graph pattern (CGP) which enforces conditions on edges, but it cannot express absence of a vertex like the anti-vertex construct.

Absence of entities has been studied in other contexts. Graph grammars [14] provide rule-based mechanisms for generating and manipulating graphs, where the productions are applied to a graph in order to obtain its derived graph when certain application conditions are met. [19] studies negative application conditions that include non-existence of nodes and edges in order to restrict how and where productions get applied. In relational algebra [9], the antijoin operator is similar to semijoin, except its result contains tuples from one relation that do not match on the common attribute from the other relation. Antijoins in SQL are achieved using WHERE clause coupled with logical operators like NOT EXISTS, limited in a similar manner as shown in Figure 1 and Figure 2.

Graph Query Engines. The backends to graph query languages are graph query engines. Recent works include PGX.D [23, 38] using PGQL [41]; GraphFlow [26] using Cypher [17]; and GAIA [34] using Gremlin [37]. These works consider backend systems details regarding efficiently executing graph queries, whereas this paper focuses on the expression of graph queries.

Subgraph Matching. There is a broad literature concerned with matching subgraphs [3, 6, 7, 20, 21, 25, 27–31, 35, 36, 39, 42] in large graphs. These works develop novel methods for efficiently matching query graphs according to isomorphism semantics in simple graphs. These works do not consider graph query expression. The set-based anti-vertex semantics can be incorporated into subgraph matching algorithms, where set operations compute match candidates.

8 CONCLUSION AND FUTURE WORK

In this paper, we described *anti-vertex*, a declarative construct that expresses absence of vertices in the subgraph. We formalized the semantics of anti-vertex for isomorphism, homomorphism and no-repeated-edge matching semantics, and generalized for the property graph model. Several examples were presented to illustrate how anti-vertices simplify expressing constraints on subgraph neighborhoods. We further studied how anti-vertex construct can be added to graph query languages by showcasing extensions to Cypher language's pattern matching grammar. Finally, we discussed how anti-vertices in the query graph interact with the symmetry breaking technique employed in subgraph matching frameworks.

As future work, we aim to generalize the philosophy of anti-vertex to be able to express higher-order constraints on neighborhoods. For example, path patterns are commonly used subgraph queries that look for multi-hop paths between input vertices. Absence of paths, or *anti-path*, can potentially be a useful construct. For example, querying vertices that are separated by at least a few hops could be declaratively expressed using anti-paths. This can be further generalized to queries where certain subgraphs may not be induced between the vertices in the data graph. For instance, identifying triplets that are connected to a common neighbor, but that are themselves pairwise-disconnected from each other can be expressed using an anti-triangle pattern (abusing the 'anti' keyword). There are several open questions in this direction, ranging from semantics of such higher-order constraints to systematically incorporating them in graph query languages. Since this paper does not consider edges between anti-vertices, formalizing those semantics would be crucial. We envision the final semantics of anti-vertices would naturally capture the higher-order extensions.

REFERENCES

[1] Roberto Alonso and Stephan Günnemann. Mining contrasting quasi-clique patterns. *CoRR*, abs/1810.01836, 2018.

[2] Amazon, Inc. Amazon Neptune, 2022. Version 1.1.0.0.

[3] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *Proceedings of the VLDB Endowment*, 11(6):691–704, February 2018.

[4] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. G-CORE: A core for future graph query languages. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '18, page 1421–1432, 2018.

[5] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of Modern Query Languages for Graph Databases. *ACM Computing Surveys*, 50(5), September 2017.

[6] Bibek Bhattarai, Hang Liu, and H. Howie Huang. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '19, page 1447–1462, 2019.

[7] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '16, pages 1199–1214, 2016.

[8] James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu, and Linhong Zhu. Finding Maximal Cliques in Massive Networks. *ACM Transactions on Database Systems*, 36(4), December 2011.

[9] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[10] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax. Technical report, W3 Consortium, 2014.

[11] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Hannes Voigt, Oskar van Rest, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. Graph Pattern Matching in GQL and SQL/PGQ. *CoRR*, abs/2112.06217, 2021.

[12] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. Aggregation Support for Modern Graph Analytics in TigerGraph. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '20, page 377–392, 2020.

[13] William Eberle and Lawrence Holder. Discovering structural anomalies in graph-based data. In *Seventh IEEE international conference on data mining workshops (ICDMW 2007)*, pages 393–398. IEEE, 2007.

[14] Hartmut Ehrig, Annegret Habel, and Hans-Jörg Kreowski. Introduction to Graph Grammars with Applications to Semantic Networks. *Computers & Mathematics with Applications*, 23(6-9):557–572, 1992.

[15] Grace Fan, Wenfei Fan, Yuanhao Li, Ping Lu, Chao Tian, and Jingren Zhou. Extending graph patterns with conditions. In *Proceedings of the ACM International Conference on Management of Data*, pages 715–729, 2020.

[16] Wenfei Fan, Yinghui Wu, and Jingbo Xu. Functional Dependencies for Graphs. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1843–1857, 2016.

[17] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '18, page 1433–1445, 2018.

[18] Joshua A. Grochow and Manolis Kellis. Network Motif Discovery Using Subgraph Enumeration and Symmetry-Breaking. In *Research in Computational Molecular Biology*, pages 92–106, 2007.

[19] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996.

[20] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '19, pages 1429–1446, 2019.

[21] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turbo$_{ISO}$: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '13, pages 337–348, 2013.

[22] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. Technical report, W3 Consortium, 2013.

[23] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. PGX.D: A Fast Distributed Graph Processing Engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, 2015.

[24] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A Pattern-Aware Graph Mining System. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020.

[25] Kasra Jamshidi and Keval Vora. A Deeper Dive into Pattern-Aware Subgraph Exploration with Peregrine. *ACM SIGOPS Operating Systems Review*, 55(1):1–10, 2021.

[26] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An Active Graph Database. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '17, page 1695–1698, 2017.

[27] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath H.A. Jarrah. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '16, pages 1231–1245, 2016.

[28] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '18, pages 411–426, 2018.

[29] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable Subgraph Enumeration in MapReduce. *Proceedings of the VLDB Endowment*, 8(10):974–985, June 2015.

[30] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. Scalable Distributed Subgraph Enumeration. *Proceedings of the VLDB Endowment*, 10(3):217–228, November 2016.

[31] Amine Mhedhbi and Semih Salihoglu. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proceedings of the VLDB Endowment*, 12(11):1692–1704, July 2019.

[32] Neo4j, Inc. Neo4j Graph Database, 2022. Version 4.4.

[33] Caleb C. Noble and Diane J. Cook. Graph-Based Anomaly Detection. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, page 631–636, 2003.

[34] Zhengping Qian, Chenqiang Min, Longbin Lai, Yong Fang, Gaofeng Li, Youyang Yao, Bingqing Lyu, Xiaoli Zhou, Zhimin Chen, and Jingren Zhou. GAIA: A System for Interactive Analysis on Distributed Graphs Using a High-Level Language. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '21, pages 321–335. USENIX Association, April 2021.

[35] Miao Qiao, Hao Zhang, and Hong Cheng. Subgraph Matching: On Compression and Computation. *Proceedings of the VLDB Endowment*, 11(2):176–188, October 2017.

[36] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. Fast and Robust Distributed Subgraph Enumeration. *Proceedings of the VLDB Endowment*, 12(11):1344–1356, July 2019.

[37] Marko A. Rodriguez. The Gremlin Graph Traversal Machine and Language. In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, page 1–10, 2015.

[38] Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine. In *Proceedings of the Fifth International Workshop on Graph Data-Management Experiences & Systems*, GRADES '17, 2017.

[39] Shixuan Sun and Qiong Luo. In-Memory Subgraph Matching: An In-Depth Study. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '20, page 1083–1098, 2020.

[40] Harsh Thakkar, Dharmen Punjani, Jens Lehmann, and Sören Auer. Two for One: Querying Property Graph Databases Using SPARQL via Gremlinator. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '18, 2018.

[41] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, 2016.

[42] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. HUGE: An Efficient and Scalable Subgraph Enumeration System. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '21, page 2049–2062, 2021.

[43] Stéphane Zampelli, Yves Deville, and Pierre Dupont. Approximate Constrained Subgraph Matching. In Peter van Beek, editor, *Principles and Practice of Constraint Programming*, CP 2005, pages 832–836, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.