Translating Anti-Vertex in Cypher for Graph Databases and Graph Mining Systems

by

Saad Ahmad

B.Sc. (Computer Science), University of Rochester, 2021

B.Sc. (Business Marketing), University of Rochester, 2021

Project Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY

Fall 2023

Declaration of Committee

Name:	**Saad Ahmad**
Degree:	**Master of Science**
Title:	**Translating Anti-Vertex in Cypher for Graph Databases and Graph Mining Systems**
Committee:	**Chair:** **Gregory Baker** Lecturer, Computing Science
	Keval Vora Supervisor Associate Professor, Computing Science
	Jiannan Wang Committee Member Associate Professor, Computing Science
	Nick Sumner Examiner Associate Professor, Computing Science

Abstract

An Anti-vertex is 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. Anti-vertices are natively supported in Peregrine, a state-of-the-art graph mining system, and has been proposed as a prototype extension to Cypher graph query language. The semantics of Anti-vertices have been well-established in accordance with semantics of subgraph matching, which includes isomorphism, homomorphism, and no-repeated-edge matching. This project focuses on enabling two cypher-based backends, Neo4j and Graphflow, to support anti-vertices. The two backends support anti-vertex queries by executing WHERE NOT EXISTS queries to match patterns based on the absence of a particular vertex. The query engine in Graphflow has been modified to support nested subqueries using the WHERE NOT EXISTS queries. The Cypher grammar in Graphflow is updated to accept anti-vertex query syntax as user input. The anti-vertex queries are translated to an appropriate WHERE NOT EXISTS queries by the backend. We have benchmarked the translated anti-vertex queries in Neo4j and Graphflow against the natively supported anti-vertex queries in Peregrine. The native anti-vertex query support in Peregrine performs significantly faster than the translated WHERE NOT EXISTS query in Graphflow and Neo4j. The performance gap highlights opportunities to improve graph database systems by natively support anti-vertex constructs.

Keywords: graph mining; cypher query language; graph query language; anti-vertex; graph databases; peregrine; neo4j; subgraph matching

Dedication

To my family

Acknowledgements

I extend my sincere appreciation to the individuals who played pivotal roles in my successful journey through the Master's program.

I am profoundly grateful to my advisor, Dr. Keval Vora, whose unwavering support, and invaluable expertise have been instrumental in shaping this project. His insightful feedback not only guided me through the intricacies of my research but also motivated me to elevate the quality of my work to new heights.

I am very grateful to all the members of my project committee. Dr. Nick Sumner, Dr. Jiannan Wang and Dr. Gregory Baker, thank you for your collaboration.

I also want to express my gratitude to my lab members: Kasra, Mugilan, Joanna, and Mobin. Their mentorship and unwavering assistance have been invaluable throughout this academic journey. Their insights and camaraderie have not only made the research environment enjoyable but have also contributed significantly to my personal and academic growth.

Finally, I would like to express my gratitude to my family for their unconditional support throughout my studies.

Table of Contents

List of Tables

List of Figures

List of Acronyms

GDBMS	Graph Database Management System
SFU	Simon Fraser University

Chapter 1. Introduction

Subquery execution is an integral feature of graph mining systems that allows users to model complex patterns. Graph mining systems like Peregrine [5] utilize pattern-matching semantics to find possible subgraphs in larger data graphs that match the user-specified pattern. Most graph mining systems come with out-of-the-box support for pattern matching where a particular vertex has an existing neighborhood that constitutes a pattern. On the other hand, intricate patterns indicating the absence of specific vertices in subgraph neighborhoods inside the data graph offer users insightful information. Existing graph mining systems do not support efficient pattern matching when finding patterns with missing vertices in graph neighborhoods.

Most existing graph mining systems [11, 12] do not have out-of-the-box support to define the absence of vertices in subgraph patterns. The absence of a vertex in a subgraph is formalized through an Anti-vertex [4], and Peregrine [5] is the first graph mining system to support Anti-vertex subgraph patterns. An Anti-vertex is a declarative construct that indicates the absence of a vertex. The resulting subgraphs matched to an anti-vertex should not have the specified vertex in its neighborhood. The current subgraph matching semantics, such as isomorphism, homomorphism, and no-repeated-edge matching, can be readily combined with anti-vertex semantics [4].

Figure 1 demonstrates how Anti-vertex queries simplify representing patterns with missing neighborhood vertices. The first query (on the left) is a Cypher [3] query representing patterns with missing vertices in Neo4j [3] and Graphflow [1] GDBMS. The second query (on the right) represents the same query using the Anti-vertex pattern syntax. The Anti-vertex pattern syntax simplifies the query Cypher query and makes it easier for the users to define more complex patterns concisely. Different system backends allow exploring such queries (e.g., using NOT EXISTS, as shown in Figure 1); this will enable us to extend Anti-vertex syntax to these backends by translating to an existing query syntax supported by the backend.

```
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
```

```
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
```

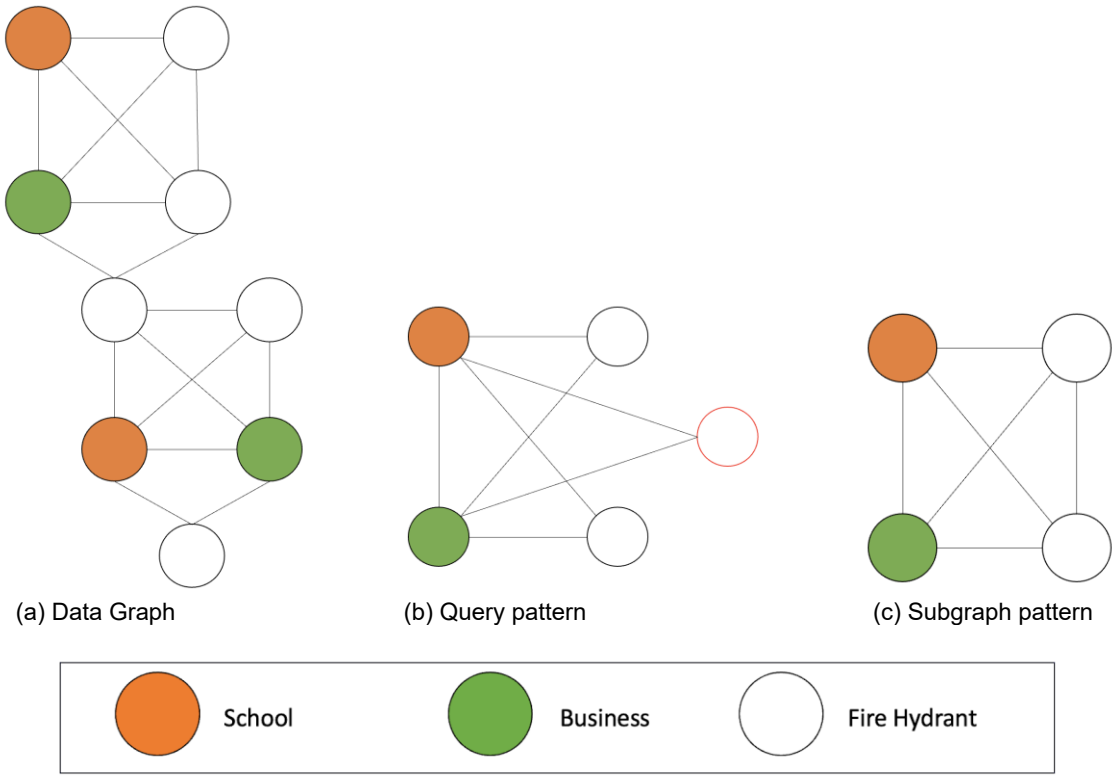


Figure 1. Anomaly detection use case for Anti-vertex. The anomalous subgraph of interest is the one where the school and the business are connected with two fire hydrants and not three. Cypher Anti-vertex queries to find anomalous subgraphs are on the top.

Simulating Anti-vertices in languages like Cypher, which lacks native support for Anti-vertex queries, involves translating the query to a nested MATCH structure. Utilizing the NOT EXISTS query semantics helps specify vertices absent in graph neighborhoods. Pattern matching in Cypher consists of creating a pattern of nodes and relationships a user

wants to find in the data graph. The WHERE clause filters the results of pattern matching based on user-defined constraints. In pattern matching, the NOT EXISTS keyword checks for the absence of a specific pattern or relationship involving the query variables. It is used within the WHERE clause to filter results where a particular condition is unmet. The NOT EXISTS clause in Cypher can represent anti-vertex queries through translation. The user's input of the anti-vertex query can be converted to an equivalent NOT EXISTS query for pattern matching. In this project, we will extend the support for anti-vertex queries to Cypher backends by query translation to NOT EXISTS queries.

Instead of translating anti-vertices to a different query, a graph mining system could support anti-vertex pattern mining natively. Native anti-vertex support means that the graph mining system can execute the anti-vertex query optimally without utilizing execution designed for a different set of queries (e.g., NOT EXISTS). Peregrine is a state-of-the-art graph mining system that supports anti-vertex queries natively. Peregrine offers pattern-aware graph mining where each query execution is tailored to specific patterns, including anti-vertex patterns. Peregrine's pattern-aware approach ensures faster pattern mining by avoiding the exploration of unnecessary subgraphs. The queries supported by Peregrine are analogous to the Cypher MATCH clause, and users can define patterns with an anti-vertex as a source or destination node in the input pattern.

These representations of the same pattern mining problem have further complexities associated with software implementation to capture the semantics of each query type. Due to these varying implementations, the execution of an anti-vertex query varies from platform to platform. This project report discusses extending anti-vertex for two existing graph database management systems: Graphflow [2] and Neo4j [3]. Both GDBMS support Cypher query processing to execute pattern-matching queries.

Neo4j was one of the first GDBMS with Cypher querying capabilities. It is designed for storing, managing, and querying graph data, making it particularly suitable for complex and highly connected data applications. The users interact with the graph database by defining a query pattern using Cypher language queries to determine subgraph patterns. Neo4j supports nested MATCH queries using the EXISTS query syntax. However, the

Cypher language in Neo4j does not support Anti-vertex query syntax. Anti-vertex queries can be run by translating to the NOT EXISTS nested Cypher language queries in Neo4j.

Graphflow is an open-source GDBMS. Like Neo4j, Graphflow has Cypher support for users to define subgraph patterns to mine in large data graphs. However, Graphflow offers limited Cypher query capabilities and restricts the query patterns a user can define. Graphflow backend does not support nested MATCH queries using the EXISTS syntax. Furthermore, there is no default implementation to support Anti-vertex queries natively. This gap in the Graphflow backend allowed us to add translated Anti-vertex query support on top of the existing backend implementation.

1.1. Contributions

For this project, we have added translated Anti-vertex support to two Cypher-based backends: Graphflow and Neo4j. The Cypher language backend in Neo4j supports nested MATCH queries using existential subqueries. We enabled translated Anti-vertex queries in Neo4j by converting the Anti-vertex query syntax to the appropriate NOT EXISTS query. Figure 1 illustrates the translation of an anti-vertex query to a NOT EXISTS query. By generating a Cypher language NOT EXISTS query, we provided a mechanism to execute translated Anti-vertex queries on Cypher-based backends. Graphflow and Neo4j are the two Cypher query backends covered within this project's scope.

We updated the Cypher language grammar to support anti-vertex query syntax for translation to NOT EXISTS Cypher language queries. Users interacting with both Graphflow and Neo4j do not have to define the complex NOT EXISTS queries. Instead, they can conveniently define a pattern with a missing vertex in the query with an "!" symbol (as shown in Figure 1). We implemented logic to translate the anti-vertex query syntax to the Cypher language's NOT EXISTS query. Referencing Figure 1, the Anti-vertex query (on the right) is converted to the Cypher query (on the left). The resulting query will be accepted by both Graphflow and Neo4j GDBMS to search for missing vertices in graph neighborhoods.

To enable translated Anti-vertex queries in Graphflow, we first had to implement the EXISTS/NOT EXISTS nested queries. The Cypher query language grammar was updated to accept nested queries as user inputs in Graphflow. The grammar parser for the Cypher language was tailored to generate a refined query plan for nested MATCH queries. On the query execution side, the nested MATCH query plans required modification to the sequencing of query operators. This ensured that our nested EXISTS queries performed reasonably compared to the original Graphflow backend for join operations.

To evaluate the performance of our translated anti-vertex queries with a comparable baseline, we have implemented a variant of Peregrine that executes translated Cypher NOT EXISTS queries. While Peregrine natively supports anti-vertices more efficiently, our variant is expected to be significantly slower than the original Peregrine since it naively executes the NOT EXISTS queries. We added Cypher translation support for Peregrine, where we generate separate pattern inputs for the outer and inner MATCH queries in NOT EXISTS. The motivation to use Peregrine without native anti-vertex support was to compare how effectively our anti-vertex queries get executed in Graphflow and Neo4j. This allowed us to uncover the performance gaps in single-threaded and multithreaded graph databases.

Our implementation is benchmarked against Peregrine's state-of-the-art native support for Anti-vertex and Peregrine without the native Anti-vertex support. Our experiments show that Peregrine's native anti-vertex support outperformed our translated Anti-vertex queries in Graphflow. The performance gap varied for each query and data graph, with the speed up for Peregrine being more than three orders of magnitude compared to Graphflow. Our results highlight the performance improvement that can be achieved in Cypher-based backends with native Anti-vertex support.

Chapter 2. Related Work

This chapter discusses literature relevant to graph mining techniques and systems. We discuss the three matching semantics used by graph mining systems and explain the concept of Anti-vertex in graph queries. The remaining chapter elaborates on the three systems in this project's scope: Neo4j, Graphflow, and Peregrine.

2.1. Matching semantics

Graph mining systems employ one of the three popular matching semantics: isomorphism [8], homomorphism [6], and no repeated edges [7]. Figures 2-5 illustrate simple examples of each pattern matching semantics.

Graph homomorphism is a relationship between two graphs where there may not be a one-to-one correspondence between nodes, and the structure of one graph can be partially preserved in another. In other words, a graph homomorphism is a function that maps nodes from one graph to nodes in another graph while preserving edges. Some nodes in the data graph may map to the same node in the query graph. Figure 3 shows an example of homomorphic matches when the data graph is used to map the query graph in Figure 2. The top right pattern match in Figure 3 shows that vertex s maps to vertices c and d in the query graph. Similarly, the singular match on the bottom indicates that edge e1 gets mapped to two edges in the query graph.

No Repeated Edge refers to a constraint on graph matching, where the goal is to find a mapping between nodes of two graphs while ensuring an injective mapping of edges between the data graph and the query graph. In other words, it enforces that a particular edge from the data graph cannot be associated with multiple edges in the query graph. Figure 4 shows an example of matches invalidated under no repeated edge constraints. Two edges in the query graph were matched to edge e1 from the data graph, which violates the constraints of no repeated edge pattern semantics.

Graph isomorphism is a relationship between two graphs in which they have the same structure but may differ in labeling nodes. Two graphs, G and H, are isomorphic if there

exists a one-to-one mapping (bijective function) between their nodes, such that for every edge (u, v) in G, there is a corresponding edge (f(u), f(v)) in H, and vice versa. As shown in Figure 5, the match on the right side was invalidated because vertex s was mapped to both vertices c and d. The bottommost pattern match in Figure 4 would be considered invalid under isomorphic matching semantics because edge e1 was mapped onto the query graph twice.

Figure 2. **A small undirected data graph is defined on the left. The query pattern used to query the data graph is defined on the right.**

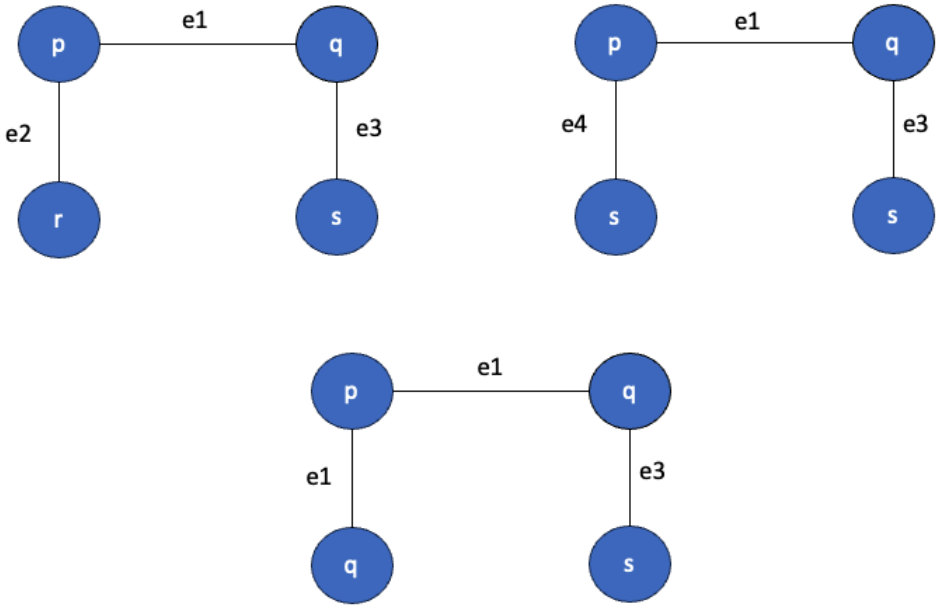


Figure 3. **Three of the pattern matches returned from the data graph using homomorphism pattern matching semantics.**

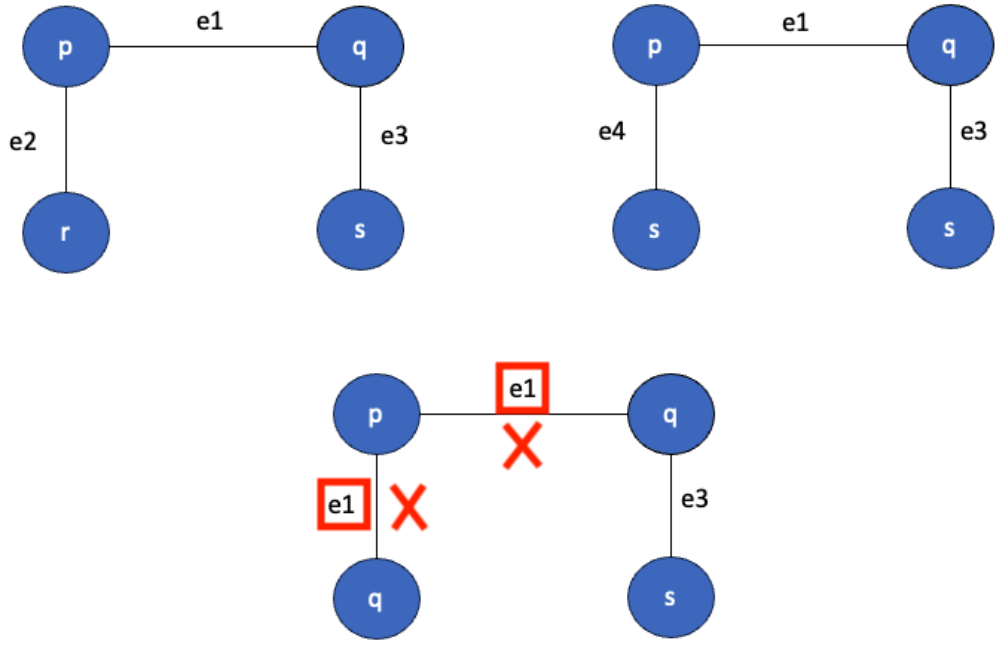


Figure 4. **Subset of patterns from the homomorphic matches which satisfy the no repeated edges constraint.**

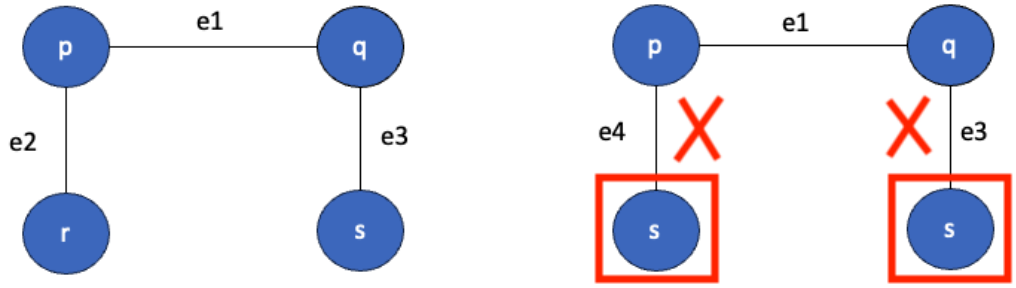


Figure 5. **Subset of patterns from the homomorphic matches which satisfy the isomorphism match semantics.**

2.2. Anti-vertex

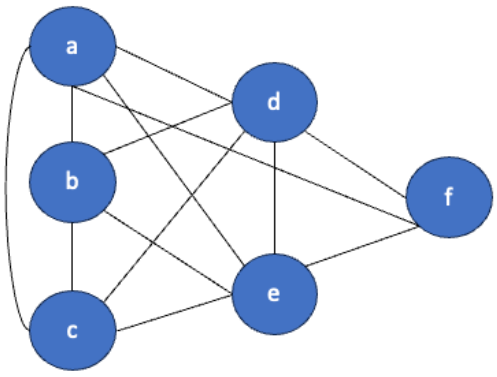
An Anti-vertex is a declarative construct that indicates the absence of a vertex, i.e., the resulting subgraph should not have a vertex in its specified neighborhood that matches the anti-vertex. Anti-vertices allow flexibility to write simplified queries with minimal changes to the existing syntax of a graph query language. Pattern matching support for anti-vertices can be extended to existing subgraph querying semantics, which includes isomorphism, homomorphism, and no-repeated-edge matching. Some prominent use cases that limit the relevant subgraphs by examining their surroundings include anomaly detection and maximum clique mining. Additionally, we discuss the current capability to express such constrained queries in the Cypher language with and without anti-vertices, emphasizing the necessity for a readily expressible construct indicating the absence of vertices.

Anomaly Detection: Figure 1 illustrates the application of anti-vertices in *anomaly detection* scenarios. The query graph imposes the constraint to exclude anomalous neighborhoods such that a business and a school are connected with less than three fire hydrants. City planning agencies vastly use such types of use cases [4]. Identifying subgraphs containing precisely two fire hydrants poses a challenge. Two apparent strategies for formulating a Cypher query to address this issue are illustrated in Figure 1. In the initial query, the problem involves matching subgraphs with two fire hydrants, excluding those within subgraphs containing three fire hydrants, and describing the absence of the third fire hydrant indirectly, resulting in the repetition of patterns and larger query sizes. This complicates the readability and management of queries (e.g., for incremental adjustments or the addition of new constraints) and introduces error-prone complexities when writing queries with multiple constraints. The second query exhibits less repetition in the subquery than the first approach but requires users to specify supplementary constraints in the outer query to achieve the intended semantics. Determining the correct constraints to ensure the subquery neither filters too many nor too few subgraphs becomes challenging in more significant queries, as users must mentally visualize how their query aligns with complex graph structures. Both approaches yield more meaningful, less declarative queries involving error-prone subqueries. Alternatively, an anti-vertex can easily express the absence of another fire hydrant neighbor.

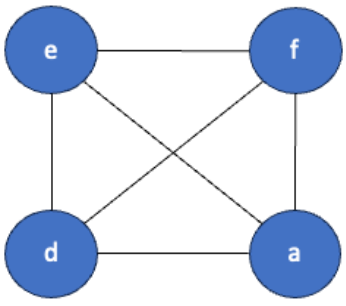
Maximal Cliques: Discovering and listing *maximal cliques* represent a popular graph mining challenge with applications in diverse fields such as social network analysis, financial analysis, security, and biology [4]. In Figure 6, the clique e-f-d-a is maximal, contrasting with other cliques that lack maximality, as each of them can be expanded into the larger clique b-c-e-d-a by introducing an additional vertex. While Cypher queries can easily define cliques of specific sizes, expressing the maximality criterion poses a challenge. The first query in Figure 6 redefines the task by seeking cliques of size k not nested within larger cliques of size k + 1. The second query identifies vertices adjacent to, yet distinct from, all k previously matched vertices. Given that the maximality constraint essentially restricts clique vertices from sharing a common neighbor, the absence of this shared neighbor can be explicitly conveyed using an anti-vertex.

```
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
```

```
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
```



(a) Data Graph



(b) Subgraph pattern

Figure 6. Maximal cliques use case. Only clique e-f-d-a is a maximal size-4 clique since all other size-4 cliques are part of larger size-5 clique b-c-e-d-a. Cypher queries to find maximal 4-cliques shown on right.

2.3. Peregrine

Peregrine [9] is a state-of-the-art pattern-aware graph mining system that directly explores the subgraphs of interest while avoiding the exploration of unnecessary subgraphs, simultaneously bypassing expensive computations throughout the mining process. Peregrine natively supports anti-vertex patterns, which allows users to specify query patterns representing the absence of a vertex directly. Peregrine can extract the semantics of anti-vertex queries and generate efficient graph exploration paths to ensure optimized pattern mining performance.

2.4. Graph databases: Neo4j and Graphflow

Neo4j is one of the most popular open-source graph databases. It is a fully transactional database, a persistent Java engine that can store structures in graphs instead of tables [10]. The query methods in Neo4j are written in the Cypher language. The Cypher language allows an expressive and efficient query execution and update of graph databases.

Graphflow is an open-source Graph Database Management System. It is a prototype graph database that evaluates general subgraph queries. Graphflow supports the property graph data model and the Cypher++ query language, which extends Cypher language with subgraph-condition-action triggers [2]. Given a query Q, Graphflow constructs its logical query execution plan. The query plan is a sequential ordering of the variables in Q using heuristics, organizing the intersection operations consistent with the picked variable ordering and deciding the placement of the filter operations in the query.

Chapter 3. Implementing nested subqueries in Graphflow

This chapter dives deeper into the implementation for enabling nested subquery execution using the EXISTS queries in Graphflow. The chapter focuses on grammar modifications, operator modifications, and the changes to query plan generation.

3.1. Grammar modifications for subqueries

Graphflow represents the Cypher grammar using ANTLR and consumes the given grammar to generate a parser. The parser inference is designed to recursively model the queries from the ANTLR language model to Java query objects. To extend the existing grammar syntax, we added oC_ExistentialSubquery to model a nested EXISTS/NOT EXISTS query that contains a nested pattern of query clauses. The following subsection from the grammar shows the grammar changes that allow the grammar to parse EXISTS queries:

```
oC_ExistentialSubquery : EXISTS SP? { SP? ( oC_Match | ( oC_Pattern ( SP? oC_Where )? ) ) SP? } ;

oC_Atom : oC_Literal
        | COUNT SP? '(' SP? '*' SP? ')'
        | oC_ParenthesizedExpression
        | oC_FunctionInvocation
        | gF_Variable
        | oC_ExistentialSubquery
        ;
```

The EXISTS keyword in Cypher is used in pattern matching to check for the existence of a second pattern related to the outer MATCH clause. The EXISTS clauses are tailored to the graph querying nature of Cypher and the specific requirements of expressing and checking graph patterns concisely and expressively. The above-mentioned grammar changes allow Cypher Grammar to accept additional MATCH clauses as part of the EXISTS subquery. These additional clauses in the EXISTS subquery enable users to define

more complex query patterns regarding the presence or absence of specific structures in the data graph.

Appendix B shows a detailed example of how a sample Cypher NOT EXISTS query is parsed by the grammar. The abstract syntax tree presented for the NOT EXISTS portion of the query shows how the tokens match the terminal and non-terminal symbols of the grammar portions above. The keyword EXISTS is the first non-terminal symbol that matches the beginning of an EXISTS expression. The oC_ExistentialSubquery production rule allows the definition of a nested query by accepting a Match clause followed by an optional Where clause. The oC_Atom production rule defines expressions that can be added to the Match query, including the EXISTS expression.

3.2. Parsing Cypher queries in Graphflow

The Graphflow query parser is defined based on the Cypher grammar. It recursively produces the final query object that is used for execution. The parser enforces the grammar constraints based on operator precedence to ensure that the Java query plan generated follows the same sequence of operators as defined in the grammar. Each query in Graphflow is represented by the PlainQueryPart object, which stores the query relationships, WHERE expressions, and RETURN expressions. Query relationships are defined by the edges in the user-defined query. WHERE expression for a query part consists of any constraints defined in the WHERE clause of a given query. The RETURN clause is the last part of the query, which specifies the mappings that the query should return as the output.

As enforced by the Cypher grammar, EXISTS/NOT EXISTS queries are wrapped by a WHERE expression. However, the execution of this query is different from how Graphflow processes equality constraints, which return Boolean results based on edge/vertex properties. As described above, our implementation defines the inner EXISTS query as a separate query part. The two query parts are connected by sharing context (query variables and edge definitions) from the outer query to the inner query.

The nested query defined within the EXISTS clause links to the outer query, which is similar to a Join operation. The inner query operator execution performs a join instead of returning Boolean results. The operators avoid extra computations by finding only a single pattern mismatch for the inner query. Graphflow has some predefined operators that are responsible for executing the query. These operators are described as follows:

1. **Scan:** The scan operator is called once on the first query variable in the pattern. It returns a set of potential vertex mapping for the given node from the data graph.

2. **Extend:** Performs an index nested loop join using the adjacency list index to match an edge of Q. Takes as input a partial match t that has matched k of the query edges in Q. For each t, Join extends t by matching an unmatched query edge x→y where x or y has already been matched.

3. **Filter:** Applies a given predicate in the query to any node or edge properties that are stored from the datagraph.

4. **Flatten:** Expands a given vertex by iterating over all potential mappings returned from the data storage.

Each query gets translated into a query plan, a chained combination of a series of the above operators. The operator ordering, as will be discussed later in this report, plays a vital role in the execution time of the query. For our implementation of the EXISTS query, we have added some additional backtracking and early termination logic to the existing operators. We have modified the ordering logic in query plan generation steps to improve the efficiency of the existential subqueries.

3.3. Storing intermediate results

In the query listed in Figure 7, we are required to make n-n join on the tuples returned by each query part. Every query section uses the previous query part's destination variable as its source variable. Given the values for a, b, c the combinations of tuples formed would

be [(a_1, b_1, c_1) ∪ (a_1, b_1, c_2) ∪ ... ∪ (a_1, b_1, c_j)]. Graphflow's DataChunks intermediate data format prevents repeat values in intermediate join stages. For the query below, the union of the list groups shown in Figure 7 is the DataChunk. DataChunk maintains factors for Cartesian products in list groups in the form [(a_1, 51) (b_1) (c_1 ... c_j)], which helps to ensure that values are not duplicated [1].

The DataChunk list groups have an index field representing whether each list group is flattened (containing only a single type of node). In the unflatten case, the list group holds more than one type of node value, which represents multiple combinations of tuples. DataChunk is shared between operators to pass intermediate stages for joins.

The first ListExtend in Figure 7 has: (i) flattened LG1 to tuple (a_1, 51); and (ii) filled a block of k1 b values in a new list group LG2. The second ListExtend has (i) flattened LG2 and iterated over it once, so its index field is 1, and LG2 now represents the tuple (b_2); and (ii) has filled a block of j number of c values in a new list group LG3. The last extends operator fills a block of j number of d values, also in LG3, by extending each c_j value to one d_j value through the single cardinality edge [1]. The flatten operator is responsible for keeping track of the index and selecting the value for the given index from the list group.

```
MATCH(a:Person)-[:Follows]→(b:Person),  (b)-[:Follows]→(c:Person),
(c)-[:Follows]→(d:Person)
WHERE a.age > 50 AND d.name = "UW"
RETURN *
```

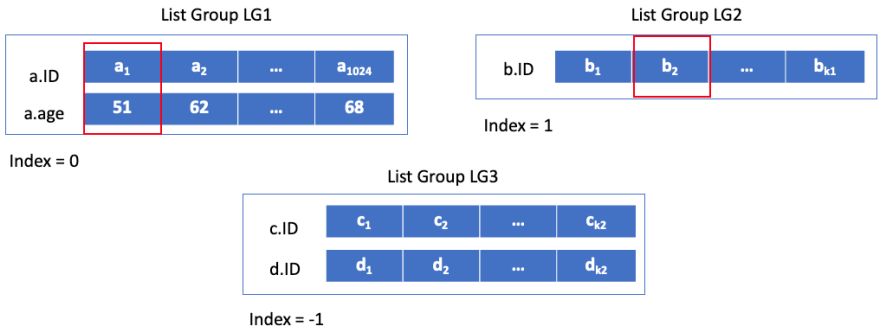


Figure 7. Intermediate data chunk for the query above. The first two list groups in the datachunk are flattened, while the last one is in an unflatten state with nodes of more than one type [1].

3.4. Filter operator execution with flatten and unflatten tuples

As discussed above, the intermediate results shared through DataChunk can be in flattened (containing a single type of nodes) or unflattened (multiple combinations of nodes) states. The implementation of the Filter operator in Graphflow is designed to handle the two states of the intermediate results based on the position of the Filter operators within the plan. The Filter operator checks whether a Flatten operator was placed before the Filter operator for the same variable(s). This means the DataChunk being passed to the Filter operator is already flattened. In this case, the evaluator compares the single value selected by the position index in the Flatten variable iteration. The evaluator returns an array of Boolean values specifying the index of the values selected based on the filter criteria. The second scenario is where the Flatten variable is placed after the Filter operator. In this scenario, the Filter operator iterates over all the possible mappings for the variables in the DataChunk and computes a list of positions with valid mappings for the comparison. This would cause extra computation by iterating over all possible mappings for a query variable to find which ones could be potential candidates. The intuition in EXISTS/NOT EXISTS subqueries is that we abandon that set of mappings as soon as the first pattern mismatch is found. By flattening the query variables before executing the Filter operator, we ensure that the Filter operator does not compute the pattern mismatches for all mappings.

Example 1: NOT EXISTS sample query plan

```
MATCH (b:Person)-[:Connects]->(a:Person),
(c:Person)-[:Connects]->(a)
WHERE NOT EXISTS {
      MATCH (c)-[:Connects]->(d:Person)
      WHERE d<>a
} RETURN *
```

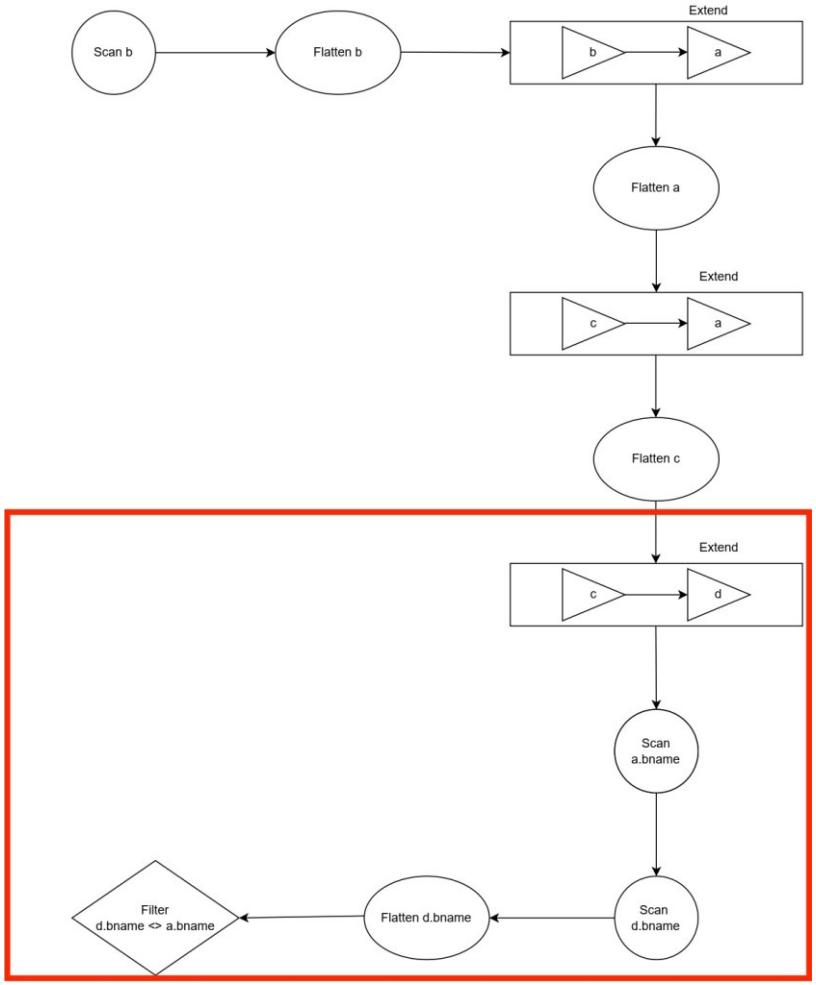


Figure 8. Query plan generated by Graphflow for the existential subquery above. The operators enclosed in the red rectangle are part of the inner NOT EXISTS query.

The heuristics Graphflow defined to order operators in the query plan were modified for the existential queries. To elaborate on Example 1, we do not have the Flatten d operator after the Filter d <> a operator. In the query plan above, the operator Flatten a selects a single value to map to the query variable a from a list of all possible mappings.

We would have to evaluate if we had Flatten d placed after the filter operator. The output from these evaluators are lists of Booleans where the index with actual value represents the position of the node values in storage that have been selected. This computation path requires iterating over all mapping values, which would result in unnecessary computations. As mentioned above, we are only looking for a single pattern mismatch for

the existential subqueries to rule out the entire mapping. By ensuring that we apply Flatten d before the filter, we can go over all the possible mappings for the given node d until a single value mapped to d violates the Filter condition.

Chapter 4. Implementing translated Anti-vertex query

4.1. Grammar modifications for Anti-vertex

Our current implementation of Graphflow accepts the Anti-vertex query syntax. The existing grammar modifications distinguish between oC_NodePattern and oC_AntiNodePattern based on the presence of an "!" symbol. The oC_RelationshipPattern clause, which models the relationship patterns defined in a query, allows anti-vertex nodes to be either the source or destination nodes in a single pattern. In Appendix B, we can see how a given Cypher query is parsed using the Cypher grammar. For simplicity, the appendix only shows the translated NOT EXISTS subquery in the abstract syntax tree.

```
oC_RelationshipPattern : oC_NodePattern SP? oC_Dash
oC_RelationshipDetail oC_RightArrowHead
                         oC_NodePattern | oC_AntiNodePattern SP?
oC_Dash oC_RelationshipDetail oC_RightArrowHead
                         oC_NodePattern |oC_NodePattern SP?
oC_Dash oC_RelationshipDetail oC_RightArrowHead
                         oC_AntiNodePattern | oC_NodePattern ;

oC_NodePattern : { SP? gF_Variable (oC_NodeType)? SP? } ;

oC_AntiNodePattern : { SP? EXCLAMATION_MARK SP? gF_Variable
(oC_NodeType)? SP? } ;
```

4.2. Node Comparison

We implemented node comparison logic that is used in the Anti-vertex query translation. If a node comparison is observed in the Cypher query while generating the query plan, the filter operator between the two nodes is further decomposed. The two nodes in a comparison operator are used to get their corresponding properties from the graph metadata storage. Each property is used to construct a single comparison predicate, and all predicates are linked together by the AND clause. For example, if we have two nodes c and d of type Person (properties: name, age, height) and a comparison predicate c<>d in the Cypher

query. The predicate c<>d would be decomposed into c.name<>d.name AND c.age<>d.age AND c.height<>d.height. Adding comparison operators on all properties ensures that the node comparison between two node objects is accurate.

4.3. Anti-Vertex translation

An Anti-vertex query is translated into a NOT EXISTS query plan, as shown in Figure 8. The logic is implemented as part of the PlainQueryPart. Here are the steps taken to translate the anti-vertex query to a NOT EXIST query in Cypher:

- Iterate over all the variable names from outer query to ensure we don't reuse the same query variable names.
- Iterate over all the chains containing an anti-vertex and add new variables if multiple chains share the same anti-vertex variable name.
- Once the complete NOT EXISTS expression is generated, it is set as the where expression of the query.

Example 2: Translating Anti-vertex query to NOT EXISTS query

```
MATCH (b:Person)-[:Connects]->(a:Person), (c:Person)-[:Connects]-
>(a), (a)-[:Connects]->(!d:Person), (b)-[:Connects]->(!d:Person),
(c)-[:Connects]->(!d:Person)
RETURN *
```

Translates to:

```
MATCH (b:Person)-[:Connects]->(a:Person), (c:Person)-[:Connects]-
>(a)
WHERE NOT EXISTS {
      MATCH (a)-[:Connects]->(e:Person), (b)-[:Connects]-
>(f:Person),
            (c)-[:Connects]->(d:Person)
      WHERE f = d AND e = d AND f <> a
} RETURN *
```

4.4. Operator.java Modifications

We have modified the behavior of existing operators in Graphflow by distinguishing between the operators defining the inner query and the outer query. In Example 1, the operators enclosed by the red rectangle are all the operators that are associated with the query contained within NOT EXISTS. This distinction is achieved by setting the existsQueryOperator flag true for operators belonging to an existential subquery. Distinguishing between the operators for both sets of queries is essential for the backtracking logic implemented for existential subqueries. The control is passed to the outer query once the inner query finds one valid mapping for any anti-vertices. The internal subquery operators backtrack until they find an operator associated with the outer query.

We have modified the behavior of existing operators in Graphflow by distinguishing between the operators defining the inner query and the outer query. In Example 1, the operators enclosed by the red rectangle are all the operators that are associated with the query contained within NOT EXISTS. This distinction is achieved by setting the existsQueryOperator flag to true for operators belonging to an existential subquery.

- **existsQueryOperator**
    - Records if an operator is generated for the query part of the WHERE EXISTS clause.
- **lastOperatorInChain**
    - This is set to true for the operator that has the next operator as the appendSink operator.
- **earlyTermination**
    - This flag is set to true in the NOT EXISTS case to exclude the EXISTS matches by preventing the mapping to be counted in appendSink
- **operatorFailed**
    - This flag is set to true when one of the operators from the NOT EXISTS query does not find valid mappings.
- **numberOfMatchesSkipped**
    - Records every time any operator does not find valid mappings.

4.5. Enumerator.java Modifications

In the context of NOT EXISTS queries, the Flatten operator is applied to all variables within the appendExtend before introducing a Filter operator. Alternatively, the updated logic in Enumerator also ensures that inner subquery operators are used after the outer query operators. As illustrated in Example 1, the Filter operator consistently appears as the last operator after the Flatten operation for NOT EXISTS queries. This ordering is essential to preserve mappings that do not adhere to the specified pattern constraints. When operators within the NOT EXISTS query—Extend, Filter, or Flatten—fail to identify a valid mapping, the operatorFailed flag is set to true. The backtracking logic then communicates this failure by propagating the flag value backward to the last operator, which is not part of the inner query. To accurately record the current mappings of outer query variables, the backtracking process skips all operators and progresses directly to the Sink operator. The goal of this execution flow is to avoid unnecessary computations within the operators of existential subquery. After we find a single valid mapping to a single anti-vertex, we invalidate the current mapping for outer query variables. Any remaining mapping candidates in the search space for inner query variables will be discarded, and a new mapping for the external query variables will be selected for further pattern matching. This helps in speeding up the existential subqueries in Graphflow.

4.6. Cypher Support in Peregrine

We implemented a translator to convert translated NOT EXISTS Anti-vertex queries to Peregrine input patterns. This enabled running translated Anti-vertex queries in Peregrine without using the native Anti-vertex pattern mining mechanism. This implementation aimed to allow the conversion of anti-vertex queries defined using richer programming constructs provided by Cypher to input patterns accepted by Peregrine.

RelationshipPattern parsing maintains an adjacency list for the query pattern using vertex labels. The edges containing anti-vertex are held separately in a second adjacency list (if present). A second map tracks the data type for each node vertex. For experiments, we use the original Peregrine and a second worse version without the native Anti-vertex support

to test the performance of the NOT EXISTS queries. In the first case, we generate a single pattern file where each line represents labels of the source and destination vertices along with the data types for each node. An anti-edge contains an anti-vertex as the source or destination vertex. If any edges are anti-edges, an extra digit is added at the end of the edge line. For the second case, without using native Anti-vertex support, the translator generates two pattern files with an outer query and a combination of both inner and external queries. Both queries are executed separately, and we get the matches by calculating the set difference between the results of the two queries. As an example, the following query is translated to the given pattern:

Integer node labels: [a:2, b:1, c:3, d:4]

Integer node datatype: [Person: 0]

Peregrine input patterns are defined by listing each relation/edge on a separate line. The source node label is added (e.g., 1), followed by the data type label of the node (e.g., 0). Next, we add the destination node label (e.g., 2) followed by the data type label of the node (e.g., 0). We can define an anti-edge (an edge with either the source or destination node being an anti-vertex) by adding an extra integer at the end of an edge. (e.g., in the edge defined as 1 0 4 0 1, the last 1 indicates an anti-edge).

Cypher query:

```
MATCH (b:Person)-[:Connects]->(a:Person), (c:Person)-[:Connects]-
>(a), (b)-[:Connects]->(!d:Person), (c)-[:Connects]->(!d:Person)
RETURN *
```

Peregrine pattern:

1 0 2 0
3 0 2 0
1 0 4 0 1
3 0 4 0 1

Peregrine without native anti-vertex:

Outer query:
1 0 2 0
3 0 2 0

Joint inner query:
1 0 2 0
3 0 2 0
1 0 4 0
3 0 4 0

We use the count function in Peregrine to find the number of matches. In Peregrine, without the anti-vertex version, we need the actual mappings to find the matches by calculating the set difference between the results of the two queries. For this purpose, we rely on the output function in Peregrine to output all subgraph matches in a text file.

Chapter 5. Evaluation

We enabled translated anti-vertex queries on Graphflow and Neo4j. Graphflow uses homomorphism pattern matching semantics to generate subgraph matches for directed subgraphs. Neo4j supports no repeated edge pattern matching semantics and supports both directed and undirected patterns. As a baseline model, we used a less efficient version of Peregrine without the native anti-vertex support. The inner and outer queries are executed separately to generate all isomorphic matches. The anti-vertex matches are computed by finding the difference between the set of matches generated by the outer query and the inner query. All three systems use non-native anti-vertex query execution. We also executed these queries on Peregrine using native anti-vertex support by defining patterns using anti-edges. All experiments were performed on an Intel(R) Xeon(R) 6242R processor clocked at 3.10GHz with 80 cores and 65GB main memory. The system was running 64-bit Ubuntu 20.04.5 OS. Both Peregrine versions leveraged concurrent graph mining, and we used all 80 cores to execute Peregrine queries. All queries used in experiments are listed in Appendix A. The queries are named in the format: "Q- [number of total joins/edges]-[number of anti-edges]".

Table 1. Undirected data graphs used for experiments.

Graph	Number of nodes	Number of edges
CA-HePh [13]	12,008	118,521
Slashdot [14]	77,350	516,575
Amazon [15]	334,863	925,872
DBLP [15]	317,810	1,049,866
Skitter [16]	1,696,415	11,095,298

5.1. Correctness tests

Table 1 lists the five directed input graphs used for evaluation. The first graph, CA-HePh, was used to confirm if the NOT EXISTS implementation produces correct results. We executed queries with stricter constraints, which guaranteed only isomorphic matches in both Neo4j and Graphflow. This ensured that the number of results returned by both systems would be equal, which confirmed that our implementation was producing matches as expected. These results are shared below in Table 2.

Table 2. Results for correctness tests showing same number of matches for Neo4j and Graphflow.

Query	Number of matches
Q-2-1	1494
Q-3-1	36,638
Q-4-2	1,787,381
Q-5-3	1,903,952

5.2. Performance of NOT EXISTS queries against Join queries

We executed join queries on Graphflow, which consisted of the same number of edges (joins) as our translated anti-vertex queries. This experiment aimed to compare the pattern mining execution time of our NOT EXISTS queries against the original Graphflow implementation of joins. We found that Graphflow does not scale well beyond a total number of four joins, which is why we could only collect results on three of the five queries. The queries with five and six total joins were not able to produce results after more than 24 hours of execution. Graphflow observed similar behavior for queries with more than two joins on the Skitter graph with approximately 10M edges. This led us only to compare the NOT EXISTS performance against joins on Slashdot, Amazon, and DBLP data graphs. As seen in Figure 9, the NOT EXISTS query implementation performed similarly to

Graphflow's existing join implementation. The NOT EXISTS query performed slightly faster on the query with four joins, avoiding unnecessary computations by abandoning the invalid mappings early on. Our results confirmed that our implementation showed reasonable performance overall compared to the existing backend implementation in Graphflow.

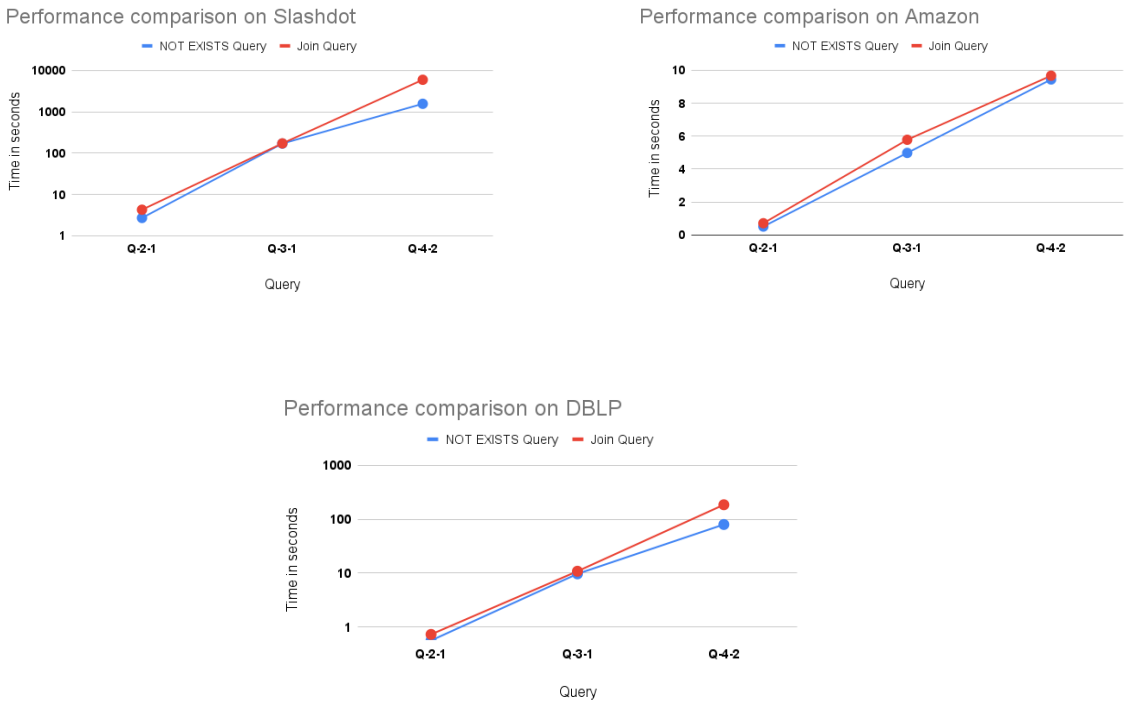


Figure 9. Query execution time of join queries and NOT EXISTS queries executed on Graphflow. Data graphs used for these experiments were Slashdot, Amazon and DBLP.

5.3. Benchmarking against Neo4j

The first performance evaluation experiment compared the query execution timings for Graphflow and Neo4j on the CA-HePh graph dataset. The results of this experiment are recorded in Table 3. Our Graphflow NOT EXISTS implementation performed almost 2.5x faster than Neo4j in the best-case scenario. The overall speed up on the executed queries ranged between 2x - 2.5x for Graphflow.

Table 3. Results for performance tests comparing query execution times on Neo4j and Graphflow.

Query	Graphflow: Time in seconds	Neo4j: Time in seconds
Q-2-1	0.769	1.349
Q-3-1	165.807	483.873
Q-4-2	1334.443	2675.671
Q-5-3	4911.921	12120.810

5.4. Benchmarking against Peregrine

We compare the performance of translated anti-vertex queries against two versions of Peregrine. The first version of Peregrine serves as a baseline model where we did not use the native anti-vertex query support. Instead, we naively executed the NOT EXISTS query by separately executing the outer and the inner queries. Using translated anti-vertex queries instead of native ones makes it a comparable baseline model to Graphflow. We recorded the timing of the larger outer query as it took longer to produce the results.

Furthermore, we also modified Peregrine to return all automorphic matches, which requires extra computation and results in performance degradation. Both queries produced outputs of unique sets of matches. The final matches for the anti-vertex query were computed by taking the difference of the two result sets. This reduced performance version of Peregrine serves as a baseline system to compare the translated query performance on the two systems. The second version of Peregrine uses natively supported anti-vertex queries and excludes any automorphic matches as intended.

Our next set of experiments includes a performance comparison of Graphflow and Peregrine. We could not collect sufficient results on the larger data graphs for Neo4j and

could not include them in this section. Any missing number of matches or execution times in Figure 10 indicate either no results were produced after more than 24 hours of execution, or the system ran out of memory during computing. The difference in the number of matches computed by each system for the same query shows variations due to the difference in matching semantics used by each system.

The overall trend for each system shows that the query execution time increases as the number of edges increases. Our baseline inefficient Peregrine variant performed similarly to Graphflow when executing translated anti-vertex queries, confirming that it is a comparable baseline model. On Slashdot and DBLP graphs, our baseline system ran out of memory as the intermediate matches generated were significantly higher. On the Skitter graph with 10M edges, Graphflow could not scale effectively for queries with three or more edges. Graphflow produced no results after more than 24 hours of execution.

On the other hand, the native anti-vertex support on Peregrine outperformed translated anti-vertex queries in both Graphflow and the baseline version of Peregrine. The overall execution timings on all queries for Peregrine were lower than the other two systems. Peregrine scaled effectively on the Skitter data graph, and it was the only system to produce results with reasonable execution timings. Furthermore, the efficiency of the native anti-vertex support can be gauged by observing the trends in the number of matches. Peregrine consistently produced lower numbers of matches as compared to Graphflow and baseline Peregrine for all data graphs. Peregrine generates the query exploration plan for the data graph by analyzing only the pattern graph. Peregrine identifies all anti-vertices and the edges connected to them in the pattern and distinguishes them from true vertices and edges. Based on its pattern-ware graph mining approach, Peregrine frequently checks for anti-vertex matches after computing a valid match for true vertices. Checking for anti-vertex matches on the fly while computing matches for true vertices helps Peregrine avoid storing intermediate results for additional computation to match anti-vertices separately [5]. This shows that the native anti-vertex support in Peregrine calculates subgraph matches more efficiently. Such Peregrine's processing model can be potentially incorporated in other backends to improve their performance.

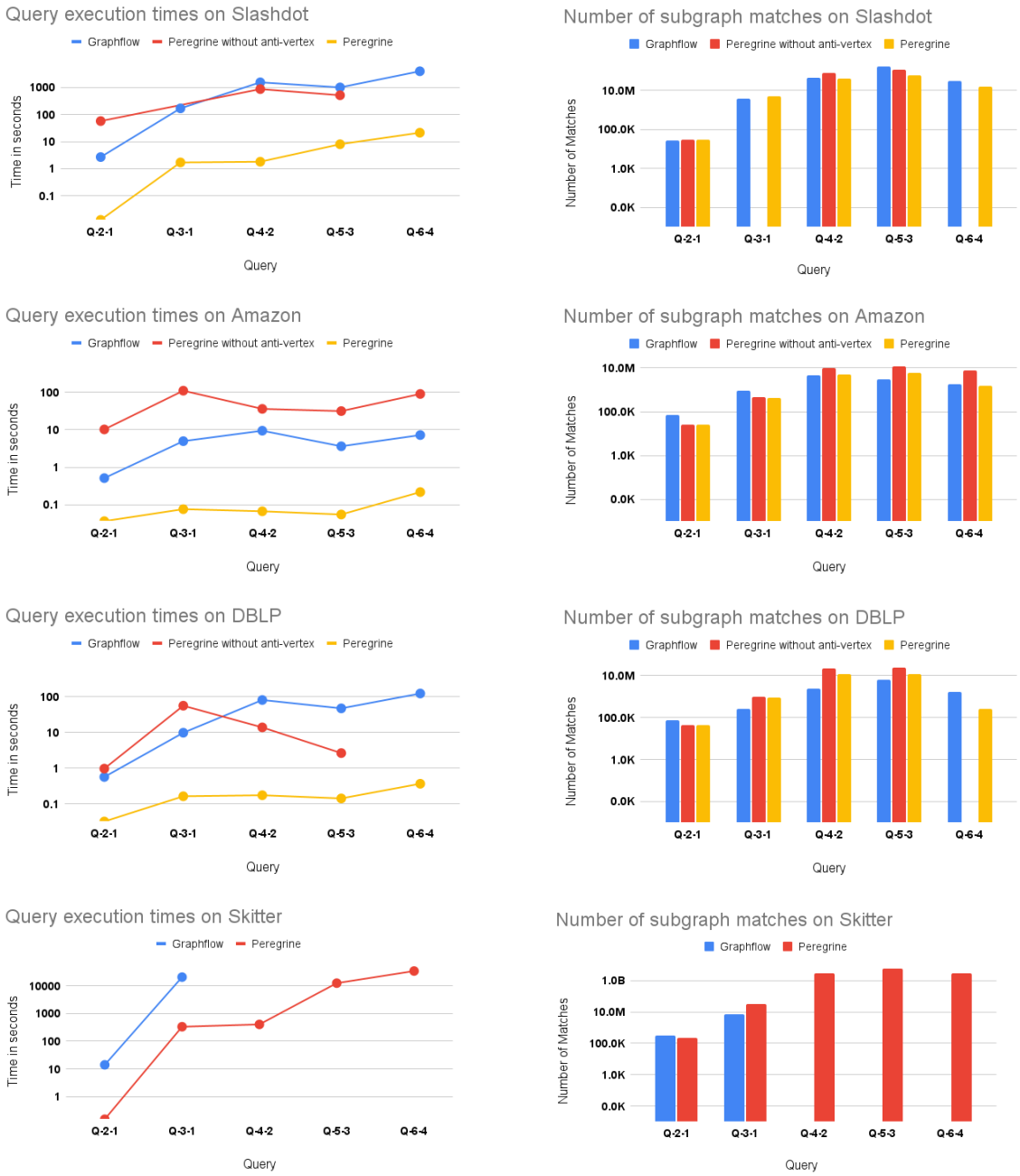


Figure 10. **On the right we have query execution times for all three systems. The bar charts on the left show the number of matches returned by each system on every query.**

Chapter 6. Conclusion

We presented two backends that support translated anti-vertex queries to find subgraph patterns without a specific vertex in the neighborhood. We enabled the anti-vertex query on Graphflow through translation to the NOT EXISTS query. The Cypher grammar in Graphflow was updated to accept the standard anti-vertex syntax and existential subqueries. We implemented the logic to support nested subquery execution in Graphflow by modifying the query plan generation logic and query operators. The backend successfully translates the anti-vertex query to a correct NOT EXISTS nested subquery for subgraph pattern matching in Graphflow. Our implementation performed comparably to Graphflow's current join implementation, and Neo4j was used to confirm that the number of matches was accurate.

Furthermore, we implemented translation logic to generate corresponding Peregrine input patterns for existential subqueries. This helped us develop a worse version of Peregrine as a baseline system, which executed translated anti-vertex queries instead of the native support. Our experiments confirmed that the native anti-vertex queries in Peregrine perform significantly faster than the translated NOT EXISTS queries in Graphflow and Neo4j. The performance gap between the two types of anti-vertex backends highlights opportunities to improve graph database systems by natively supporting anti-vertex constructs.

References

1. Gupta, Pranjal, Amine Mhedhbi, and Semih Salihoglu. "Columnar storage and list-based processing for graph database management systems." arXiv preprint arXiv:2103.02284 (2021). Brown, Bob. (2010). Books: Sustainable and Biodegradable Reading Technology. New York, NY: Hydraulic Press. doi:10.1026/0022-005X.52.6.803

2. Kankanamge, Chathura, et al. "Graphflow: An active graph database." Proceedings of the 2017 ACM International Conference on Management of Data. 2017.

3. Francis, Nadime, et al. "Cypher: An evolving query language for property graphs." Proceedings of the 2018 international conference on management of data. 2018.

4. Jamshidi, Kasra, Mugilan Mariappan, and Keval Vora. "Anti-vertex for neighborhood constraints in subgraph queries." Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). 2022.

5. Jamshidi, Kasra, Rakesh Mahadasa, and Keval Vora. "Peregrine: a pattern-aware graph mining system." Proceedings of the Fifteenth European Conference on Computer Systems. 2020.

6. Fan, Wenfei, et al. "Graph homomorphism revisited for graph matching." Proceedings of the VLDB Endowment 3.1-2 (2010): 1161-1172.

7. Deutsch, Alin, et al. "Graph pattern matching in GQL and SQL/PGQ." Proceedings of the 2022 International Conference on Management of Data. 2022.

8. Raviv, Dan, Ron Kimmel, and Alfred M. Bruckstein. "Graph isomorphisms and automorphisms via spectral signatures." IEEE transactions on pattern analysis and machine intelligence 35.8 (2012): 1985-1993.

9. Jamshidi, Kasra, and Keval Vora. "A deeper dive into pattern-aware subgraph exploration with peregrine." ACM SIGOPS Operating Systems Review 55.1 (2021): 1-10.

10. Robinson, J. Webber and E. Eifrem. Graph Databases. O‟Reilly Media Inc., California, 2013

11. Teixeira, Carlos HC, et al. "Arabesque: a system for distributed graph mining." *Proceedings of the 25th Symposium on Operating Systems Principles*. 2015

12. Dias, Vinicius, et al. "Fractal: A general-purpose graph pattern mining system." *Proceedings of the 2019 International Conference on Management of Data*. 2019.

13. J. Leskovec, J. Kleinberg and C. Faloutsos. Graph Evolution: Densification and Shrinking Diameters. ACM Transactions on Knowledge Discovery from Data (ACM TKDD), 1(1), 2007.

14. J. Leskovec, D. Huttenlocher, J. Kleinberg: Signed Networks in Social Media. 28th ACM Conference on Human Factors in Computing Systems (CHI), 2010.

15. J. Leskovec, J. Kleinberg and C. Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2005.

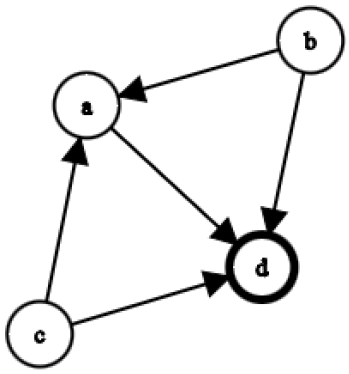
16. J. Yang and J. Leskovec. Defining and Evaluating Network Communities based on Ground-truth. ICDM, 2012.

Appendix A. Experimental queries

This appendix includes the 5 sample queries that were used for benchmarking in this project. The diagrams show anti-vertices as bolded in the pattern graph. Each WHERE NOT EXISTS query has two versions: 1) Correctness Test and 2) Translator generated. The correctness test queries have extra constraints which ensure that the matches are isomorphic and match the results from Neo4j. These were used to confirm the correctness of the WHERE NOT EXISTS implementation. The translator generator query is the one generated by the anti-vertex translator implemented in Graphflow. It follows the same query semantics as defined for the WHERE NOT EXISTS queries in the Anti-vertex paper [4].

Example 1 (Q-5-3)

Find a, b, c such that:
i) b and c Connects a, and
ii) the three do not Connect to a common PERSON



Correctness test:

```
MATCH (b:Person)-[:Connects]->(a:Person), (c:Person)-[:Connects]->(a)
WHERE NOT EXISTS {
    MATCH (a)-[:Connects]->(d:Person), (b)-[:Connects]->(e:Person),
(c)-[:Connects]->(f:Person)
    WHERE d = e AND d = f AND d <> a AND d <> b AND d <> c
}
AND a <> c AND a <> b AND b <> c
RETURN *
```

Translator generated query in Graphflow:

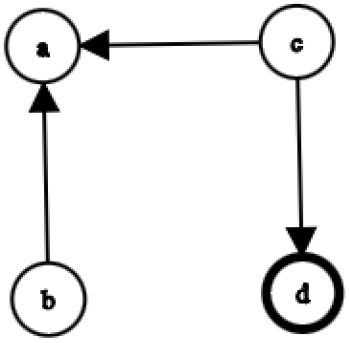
```
MATCH (b:Person)-[:Connects]->(a:Person), (c:Person)-[:Connects]->(a)
WHERE NOT EXISTS {
    MATCH (a)-[:Connects]->(d:Person), (b)-[:Connects]->(e:Person),
(c)-[:Connects]->(f:Person)
    WHERE d = e AND d = f AND d <> a
}
RETURN *
```

Join query for performance testing:

```
MATCH (b:Person)-[:Connects]->(a:Person), (c:Person)-[:Connects]->(a),
(a)-[:Connects]->(d:Person), (b)-[:Connects]->(e:Person), (c)-
[:Connects]->(f:Person)
RETURN *
```

Example 2 (Q-3-1)

Find a,b,c such that
i) b,c Connects a and
ii) c does not Connect to another PERSON d



Correctness test:

```
MATCH (b:Person)-[:Connects]->(a:Person), (c:Person)-
[:Connects]->(a:Person)
WHERE NOT EXISTS {
    MATCH (c)-[:Connects]->(d:Person)
    WHERE d <> a
}
AND b <> c
RETURN *
```

Translator generated:

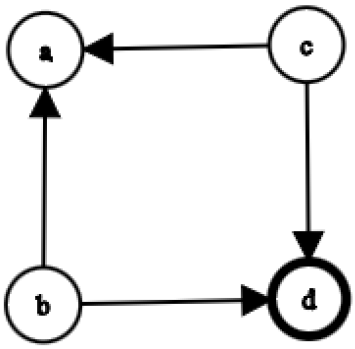
```
MATCH (b:Person)-[:Connects]->(a:Person), (c:Person)-
[:Connects]->(a:Person)
WHERE NOT EXISTS {
    MATCH (c)-[:Connects]->(d:Person)
    WHERE d <> a
}
RETURN *
```

Join query for performance testing:

```
MATCH (b:Person)-[:Connects]->(a:Person), (c:Person)-
[:Connects]->(a:Person), (c)-[:Connects]->(d:Person)
RETURN *
```

Example 3 (Q-4-2)

Find a,b,c such that
i) b and c Connect a, and
ii) b and c does not Connect with d, and
iii) c does not Connect with d



Correctness test:

```
MATCH (b:Person)-[:Connects]->(a:Person), (c:Person)-
[:Connects]->(a)
WHERE NOT EXISTS {
    MATCH (b)-[:Connects]->(d:Person), (c)-[:Connects]-
>(e:Person)
    WHERE d = e AND a <> d AND c <> d
} AND b <> c AND a <> c
RETURN *
```

Translator generated:

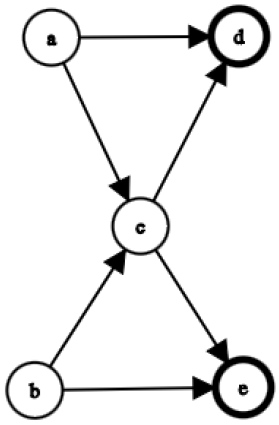
```
MATCH (b:Person)-[:Connects]->(a:Person), (c:Person)-
[:Connects]->(a:Person)
WHERE NOT EXISTS {
    MATCH (b:Person)-[:Connects]->(e:Person), (c:Person)-
[:Connects]->(d:Person)
    WHERE e = d AND d <> a
}
RETURN *
```

Join query for performance testing:

```
MATCH (b:Person)-[:Connects]->(a:Person), (c:Person)-
[:Connects]->(a:Person), (b:Person)-[:Connects]-
>(e:Person), (c:Person)-[:Connects]->(d:Person)
RETURN *
```

Example 4 (Q-6-4)

Find a,b,c such that
i) a and b Connect with c, and
iii) a and c do not Connect with another common PERSON, and
iii) b and c do not Connect another common PERSON



Correctness Test:

```
MATCH (a:Person)-[:Connects]->(c:Person), (b:Person)-[:Connects]-
>(c)
WHERE NOT EXISTS {
    MATCH (a)-[:Connects]->(d:Person), (c)-[:Connects]-
>(x:Person), (b)-[:Connects]->(e:Person), (c)-[:Connects]-
>(y:Person)
    WHERE e = y AND d = x AND c <> d AND c <> e AND d <> e
} AND a <> b
RETURN *
```

Translator Generated:

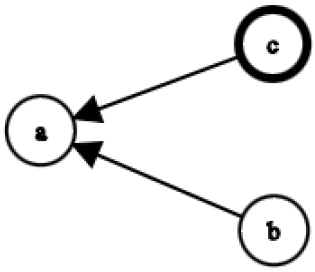
```
MATCH (a:Person)-[:Connects]->(c:Person), (b:Person)-[:Connects]-
>(c:Person)
WHERE NOT EXISTS {
    MATCH (a)-[:Connects]->(f:Person), (c)-[:Connects]-
>(d:Person), (b)-[:Connects]->(g:Person), (c)-[:Connects]-
>(e:Person)
    WHERE g = e AND f = d AND f <> c AND g <> c
}
RETURN *
```

Join query for performance testing:

```
MATCH (a:Person)-[:Connects]->(c:Person), (b:Person)-[:Connects]-
>(c:Person), (a)-[:Connects]->(f:Person), (c)-[:Connects]-
>(d:Person), (b)-[:Connects]->(g:Person), (c)-[:Connects]-
>(e:Person)
RETURN *
```

Example 5 (Q-2-1)

Find a,b,c such that:
i) b Connects with a
ii) no other person c Connects with a



Correctness test:

```
MATCH (b:Person)-[:Connects]->(a:Person)
WHERE NOT EXISTS {
    MATCH (c:Person)-[:Connects]->(a:Person)
    WHERE b <> c
}
RETURN *
```

Translator generated:

```
MATCH (b:Person)-[:Connects]->(a:Person)
WHERE NOT EXISTS {
    MATCH (c:Person)-[:Connects]->(a:Person)
    WHERE b <> c
}
RETURN *
```

Join query for performance testing:

```
MATCH (b:Person)-[:Connects]->(a:Person),
(c:Person)-[:Connects]->(a:Person)
RETURN *
```

Appendix B. NOT EXISTS query grammar parsing

Query:

```
MATCH (b:Person)-[:Connects]->(a:Person), (c:Person)-[:Connects]-
>(a)
WHERE NOT EXISTS {
      MATCH (c)-[:Connects]->(d:Person)
      WHERE d<>a
} RETURN *
```

Parsed NOT EXISTS nested query grammar:

```
oC_Where WHERE
(
  oC_Expression (
    oC_OrExpression (
      oC_AndExpression (
        oC_NotExpression NOT (
          oC_ComparisonExpression (
            oC_AddOrSubtractExpression (
              oC_MultiplyDivideModuloExpression (
                oC_PowerOfExpression (
                  gF_UnaryNegationExpression (
                    oC_PropertyOrLabelsExpression (
                      oC_Atom (
                        oC_ExistentialSubquery EXISTS
                        {
                          oC_Match MATCH
                          (
                            oC_Pattern (
                              oC_RelationshipPattern (
                                oC_NodePattern (
                                  (gF_Variable c)
                                )

              (oC_Dash -)
                                (oC_RelationshipDetail [ :
(oC_RelationshipLabel (gF_Variable Connects)) ]
                                (oC_Dash -))
                                (oC_RightArrowHead >)
                                oC_NodePattern (
                                  (gF_Variable d)
                                  (oC_NodeType : (gF_Variable Person))
                                )
                              )
                            )
                          )
                          (
                            oC_Where WHERE
                            (
                              oC_Expression (
                                oC_OrExpression (
                                  oC_AndExpression (
                                    oC_NotExpression (
                                      oC_ComparisonExpression (
                                        oC_AddOrSubtractExpression (
                                          oC_MultiplyDivideModuloExpression (
```

```
                                        oC_PowerOfExpression (
                                          gF_UnaryNegationExpression (
                                            oC_PropertyOrLabelsExpression
(
                                              oC_Atom (
                                                gF_Variable d
                                              )
                                            )
                                          )
                                        )
                                      )
                                      (gF_Comparison <>)
                                      (
                                        oC_AddOrSubtractExpression (
                                          oC_MultiplyDivideModuloExpression
(
                                            oC_PowerOfExpression (
                                              gF_UnaryNegationExpression (
                                                oC_PropertyOrLabelsExpressi
on (
                                                  oC_Atom (
                                                    gF_Variable a
                                                  )
                                                )
                                              )
                                            )
                                          )
                                        )
```