Local Online Indexing for Faster Subgraph Queries

Sandeep Polisetty*
University of Massachusetts-Amherst

Juelin Liu*
University of Massachusetts-Amherst

Aidan O'Neill
University of Massachusetts-Amherst

Marco Serafini
University of Massachusetts-Amherst

ABSTRACT

Subgraph queries are a fundamental functionality in graph data management. The multi-way join operator is commonly used for these queries, motivating the development of fast implementations. Recent work focuses on leveraging SIMD instructions by statically converting the input graph into an optimized binary format. This paper proposes a different approach called *Local Online Indexing*: instead of building one static, query-independent index a priori, we build many query-dependent instances of an index during query execution. Each instance encodes partial results local to a specific area of the input graph and deleted once that area is explored. We use these principles to implement NOTI, an implementation of the multi-way join operator based on a local online index called the Rooted Triangle Matrix (RTM). We find that NOTI and the RTM result in significant speed-ups over state-of-the-art algorithms.

1 INTRODUCTION

Graphs represent and integrate large amounts of data in domains such as knowledge bases, finance, security, user recommendations, social networks, and biology [28, 29]. This has lead to much recent work in graph databases [2], graph processing [6, 19, 26, 32, 37], and graph mining frameworks [5, 20, 27, 34]. One of the fundamental problems in graph databases and graph mining is subgraph search. Here, we find all subgraphs of an input graph that match a given query graph or pattern. A subgraph query can be seen as a sequence of *binary* self-joins over the edge table of a database, where each join matches *one* edge of the query graph. An alternative is to use *multi-way* joins to match *multiple* edges incident to one vertex of the query graph at once. Multi-way joins are fast and worst-case optimal: their worst-case complexity is not larger than the size of their output, which is not necessarily true of binary joins [23]. As such, multi-way joins are used by state-of-the-art graph databases [1, 21]. Other graph mining systems use pattern matching algorithms that are also similar to multi-way joins [11, 20].

Existing work has explored efficient implementations of the multi-way join operator, focusing on in-memory graph data. The fundamental bottleneck in multi-way joins is set intersection, which is used to find common vertices among multiple adjacency lists. State-of-the-art algorithms [1, 7] focus on speeding up this operation by leveraging the vectorized SIMD instruction set of CPUs. They use *input indexing*: before executing the query, they convert the entire input graph into a format optimized for SIMD operations. Their general principle is to identify and represent dense regions of the graph using a compact binary format that can be efficiently processed using SIMD.

This paper proposes using *Local Online Indexing*. Subgraph queries match query graphs by incrementally visiting different areas of the input graph and building partial results along the way. Local online indexing *indexes partial results online* at query execution time. We build several instances of the index, each *local* to the currently examined area.

We use these principles to design NOTI, a multi-way join algorithm that leverages SIMD instructions using a local online index called the RTM. Query optimizers typically use multi-way joins for dense query subgraphs in which finding triangles is a common intermediate step [1]. After finding triangles, we can match more complex queries by performing subsequent triangle intersections. The RTM index leverages this observation by storing triangles in a binary matrix so triangle intersections become bitwise array operations which can be performed using vectorized SIMD instructions found in standard CPUs.

NOTI does not build a single index storing all triangles as such an index would be excessively large. Instead, it builds and deletes multiple compact instances of the RTM during query execution. Each instance indexes triangles incident to a specific root vertex. We also use the RTM index to prune automorphisms by supporting arbitrary canonicality checks. NOTI combines the RTM index with SIMD-optimized algorithms for general set intersection.

Our experiments show that NOTI often significantly outperforms state-of-the-art algorithms based on input indexing, reducing query execution time by up to two orders of magnitude even if we ignore preprocessing costs. This is due to the fact that NOTI indexes partial matches that are necessarily built by the query, with minimal extra processing. In general, a local online indexing algorithm is particularly advantageous when preprocessing costs are hard to amortize. Graph applications very often have to deal with dynamic data [28] and many graph databases and storage systems support updates [2, 38]. In these systems, the index may quickly become outdated. Shorter-running queries such as queries that have higher selectivity are also less tolerant to preprocessing costs. In addition to this benefit, local online indices are compact.

This paper makes the following contributions:

- We introduce the concept of local online indexing (Section 3);
- We present NOTI, a multi-way join algorithm, and RTM, the local online index NOTI leverages (Section 4)
- We review how NOTI leverages SIMD instructions in order to perform set intersection and other essential subfunctionalities efficiently (Section 5)
- We experimentally evaluate NOTI and show that it significantly outperforms state-of-the-art algorithms based on local online indexing (Section 6).

*Both authors contributed equally to this research.

2 BACKGROUND AND MOTIVATION

2.1 Definitions

Multi-way joins. The binary join operator $R \bowtie_\theta S$ computes the Cartesian product of R and S and then selects tuples according to a logical predicate θ. The most common join is the equi-join, where θ is an = predicate between two attributes. Multi-way joins extend binary joins to multiple relations. A multiway join $\bowtie_\theta (R_1, R_2, \ldots, R_n)$ computes the Cartesian product of the relations $R_1, R_2, \ldots, R_n$ and then selects tuples based on θ.

Subgraph Queries. A subgraph query takes as input a data or input graph $G(V, E)$, and a query graph $Q(V_q, E_q)$. V (resp. V_q) is the set of vertices of the data graph (resp. query graph) and E (resp. E_q) is the set of edges denoted as pairs of vertices. In this paper, we consider undirected and unlabeled graphs for simplicity, but this work can be extended to directed and labeled graphs. A subgraph query outputs all subgraphs of G isomorphic to the query graph.

Definition 2.1 (Query match). Let $G'(V', E')$ be a subgraph of G, that is, $V' \subseteq V$ and $E' \subseteq E$. We say that *the subgraph G' matches with the query graph Q* if and only if there exists a bijection $b : V' \rightarrow V_q$ such that:

$$(v_1, v_2) \in E' \iff (b(v_1), b(v_2)) \in E_q.$$

Two graphs are automorphic if they are symmetric to each other. Automorphic matches to the same query graph are typically considered redundant and are filtered. To do so, we can include a *canonicality constraint*. Canonicality constraints impose a partial ordering among the ids of the vertices in a match. Matches that do not respect the canonicality constraints of a query are not returned and so automorphic results are filtered.

Using multi-way joins for subgraph queries. Binary and multi-way joins are alternative approaches to extending partial matches. Binary joins match one edge of the query graph. If we represent the graph as a table T of (s, d) edges having two attributes (source vertex, destination vertex), matching one edge corresponds to performing a standard binary join operator between the table containing the partial matches and the graph table T. A subgraph query executor that uses only binary joins uses an "edge-at-a-time" approach. This is in contrast to multi-way joins, which match one vertex of the query graph at a time.

When the next query vertex to be matched has edges to two or more query vertices that have already been matched, matching one vertex requires matching multiple edges at the same time. This corresponds to performing one multi-way join on the graph table. Using only multi-way joins is a "vertex-at-a-time" approach.

Query optimization. Subgraph matching queries are computationally complex, and developing query optimizers for them is an active area of research. Current query optimizers assign different subgraphs of the query to different join types, as multi-way joins are generally more efficient than binary joins when matching dense query subgraphs and binary joins are generally more efficient than multi-way joins when matching sparse query subgraphs [1]. In addition to determining which type of join to use for different subgraphs of the query, query optimizers determine a *matching order* of the query vertices, which the multi-way join operator takes as input. This is the order in which the multi-way join implementation must match the query vertices, according to a unique integer identifier i associated with each vertex.

2.2 Related Work

Query Optimizers. Emptyheaded proposed a query optimizer combining binary and multi-way joins [1]. It uses the example of a "barbell" query, which consists of two triangles, with one vertex in the first triangle connected to one vertex in the second. An optimal plan uses a multi-way join to match the two triangles and a binary join to match the edge connecting the triangles. Follow up work on GraphFlow introduced a more sophisticated optimizer that combines binary and multi-way joins and considers the effect query vertex ordering has on multi-way join performance [13]. More recent work [22] has developed a cost-based optimizer that uses dynamic programming to determine the optimal query vertex ordering, and has integrated that work into GraphFlow.

Existing implementations of multi-way joins. Prior work has proposed several approaches to implementing multi-way joins. The main bottleneck in multi-way joins is set intersection among the adjacency lists of vertices in the partial match. Prior work proposed using binary search or binary trees to avoid scanning entire adjacency lists to perform this set intersection [35]. While avoiding a complete scan, this approach causes frequent cache misses due to the random access and branch mispredictions caused by the frequent conditional operators in binary search implementations.

More recent work [1, 8] has explored leveraging SIMD instructions. We call these *input indexing* techniques because they focus on converting the input graph G into a (typically binary) format optimized for SIMD. A graph can be seen as a binary adjacency matrix A having $|V^2|$ elements that is defined as follows: $A_{i,j} = 1 \iff$ vertices v_i and v_j are neighbors. With a binary representation we can intersect the adjacency list of two vertices v_i and v_j by performing a bitwise-logical-and operation & on the rows A_i and A_j of the adjacency matrix. The resulting binary vector $b = A_i \,\&\, A_j$ will have an entry $b_k = 1 \iff v_k \in N(v_i) \wedge \cap N(v_j)$. The bitwise-logical-and can be executed efficiently using SIMD instructions.

Representing the entire graph as a binary adjacency matrix typically requires more space than a standard adjacency list representation, as the matrix requires V^2 bits and is sparse when $|E|/2 \ll V^2$ as is typical. Recent work [1, 7] on input indexing compresses the adjacency matrix by arranging it into dense regions and skipping over sparse regions. Sparse regions are represented similarly to standard adjacency lists. SIMD instructions can then perform intersection on dense regions.

3 LOCAL ONLINE INDEXING

3.1 Motivation

Most, existing work focuses on performing expensive preprocessing of the graph in order to implement a fast SIMD-optimized multi-way join operator, with the understanding that the cost of this preprocessing will be amortized. Rather than performing this expensive preprocessing, we can index the graph online and achieve a larger benefit. Because query optimizers use multi-way joins when

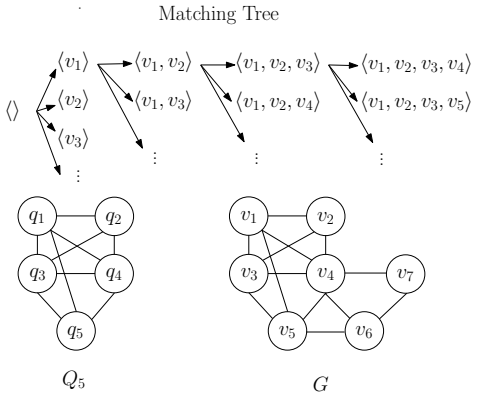


Figure 1: Matching Tree for a 0.9-Quasi-Clique Query of Size 5 (Partial) on Query Q_5 and pictured data graph G.

queries are dense, there are often many triangles in typical subgraph queries (i.e., a 4-clique has 4 triangle subgraphs). Through indexing these triangles and reusing them, we can speed up further query processing. In addition to providing an improvement on static graphs, an online approach is more readily extensible to dynamic graphs on which preprocessing cannot be amortized. This approach is lightweight and scales well. For subgraphs of the query graph that do not include triangles, we propose using SIMD-optimized set intersection algorithms that do not require preprocessing.

3.2 Defining Local Online Indexing

Visiting the matching tree. Local online indexing is based on a fundamental observation: implementations of the multi-way join operator have multiple degrees of freedom that we can exploit to expedite execution. The query optimizer invokes the multi-way join operator by indicating a query matching order and optional canonicality constraints. These specify *which* partial matches need to be explored by a multi-way join implementation, and the query optimizer typically defines a matching order that minimizes the set of partial matches. However, the implementation has ample flexibility in defining the *order* in which partial matches are explored. Local online indexing leverages this flexibility to find and efficiently index partial matches.

We can express the set of partial matches that must be explored by the multi-way join as a *matching tree* (Figure 1). Each node in the matching tree represents a partial match. Each parent-child relationship corresponds to extending the parent partial match with the next query vertex in the matching order. A subset of the matching tree created by querying G with query Q_5 is shown in Figure 1.

Common multi-way join implementations use a backtracking strategy that correspond to a depth-first search (DFS) of the matching tree. For example, in the tree pictured in Figure 1, a backtracking strategy could match v_1, v_2, v_3, v_4 and then v_5, corresponding to q_1, q_2, q_3, q_4 and q_5 respectively. After finding this full match, the algorithm will backtrack to the parent match in the tree, $\langle v_1, v_2, v_3, v_4 \rangle$, and continue the DFS from there.

Visiting the matching tree in a DFS does not require materializing partial matches beyond the ones that are currently being visited. Thus, naive DFS is a better approach than naive breadth-first search (BFS), which materializes all partial matches at each level of the tree before finding full matches. In the case of the query pictured in Figure 1, a BFS' first level would consist of all nodes, second level would consist of all edges, third level would consist of all triangles, and so on and so forth. This clearly does not scale.

Local online indexing. Local online indexing is a flexible strategy that extends the naive DFS approach. We build an index of the partial matches found in a subtree of a node n of the matching tree. The index is built *online*, at query processing time. Different *local* instances of the index are built and then discarded for different subtrees to limit the storage cost of the approach.

The structure of a local online indexing algorithm is below:

(1) Perform a DFS on the matching tree;
(2) If the current node n is the starting node of an index:
  (a) Build an index representing a subset of the partial matches in the subtree of n;
  (b) Continue the DFS from n, using the index to find the partial matches in the subtree;
  (c) When the DFS backtracks to the parent of n, discard the index.
(3) Continue to the DFS from step (1), potentially building another index for a different node.

Consider the matching tree for Q_5 pictured in Figure 1. If $\langle v_1 \rangle$ is the root node, we could index all edges ($\langle v_1, v_2 \rangle$, $\langle v_1, v_3 \rangle$, and $\langle v_1, v_4 \rangle$). Alternatively, we could build our index around the triangles ($\langle v_1, v_2, v_3 \rangle$ and $\langle v_1, v_2, v_4 \rangle$).

There are several possible algorithms that can implement this strategy depending on (a) *where* to build a local index, i.e., how to select n, (b) how to *build* the local index, (c) how to *use* the index to speed up matching. Our NOTI algorithm provides a specific answer to these questions.

4 THE NOTI ALGORITHM

4.1 Overview

This paper proposes NOTI, a physical implementation of the multi-way join operator. We now give an overview of NOTI, using the example in Figure 1.

NOTI starts its DFS by matching the first vertex of Q_5. NOTI begins by building a local online index, RTM. The vertex from which RTM construction starts is the *root vertex* of the index. In our example, v_1 is the first root vertex. The RTM indexes *all triangles* that are incident to v_1 by finding all partial matches $\langle v_1, u, u' \rangle$ with $u \in N(v_1)$ and $u' \in \cap(N(v_1), N(u))$. This is done by performing a regular set intersection on the adjacency lists. These triangles are stored in a compact format using the RTM index, which we discuss in detail in Section 4.2. Given a root vertex v and a neighbor u, the RTM supports the efficient retrieval of the set $P = \cap(N(v), N(u))$.

After building the RTM, we continue the DFS. Suppose that the DFS reaches the triangle $\langle v_1, v_2, v_3 \rangle$. The next step is to find each vertex u forming the next vertex of Q_5, $\langle v_1, v_2, v_3, u \rangle$, where $u \in \cap(N(v_1), N(v_2), N(v_3))$. These are the children of $\langle v_1, v_2, v_3 \rangle$

in the matching tree. The RTM index already contains the triangles incident to v_1 as it stores the results of the set intersections $S = \cap(N(v_1), N(v_2))$ and $S = \cap(N(v_1), N(v_3))$. Therefore, we can perform the set intersection between the RTM entries S and T. This is cheaper than the standard approach of performing a three-way set intersection $\cap(N(v_1), N(v_2), N(v_3))$, as we will discuss shortly. After finding all matches starting with $\langle v_1 \rangle$, the DFS moves to the next root vertex (v_2 in Figure 1), discards the obsolete RTM index and computes a new index.

Advantages of Local Online Indexing. Performing set intersection of partial matches using the NOTI confers several advantages. The RTM encodes all triangles in S and T using binary vectors and uses an efficient bitwise-logical-and operation using SIMD between these vectors to perform the intersection. Both input indices and the RTM use binary vectors to speed up set intersection; the RTM uses these for intersection of partial matches (triangles), and input indices use these for adjacency lists. The RTM index is local and built online in a query-dependent fashion, whereas input indexes encode the entire graph and are therefore global and query-independent.

This conceptual difference has several consequences. The RTM index is compact, since it contains vectors of size $|N(v_i)|$, with v_i the root vertex of the RTM. If we select root vertices with few neighbors, these vectors are small and dense. In contrast, the size of adjacency list vectors is equal to $|V|$, independent of the query. While existing work [1, 7] shows that these vectors can be compressed, this comes at the cost of more complex logic for set intersection, which introduces branch mispredictions and random accesses, reducing CPU utilization. The RTM index does not use these forms of compression.

Another advantage of the RTM index is computation reuse. Suppose that after matching all children of $\langle v_1, v_2, v_3 \rangle$ in the matching tree, the DFS reaches the partial match $\langle v_1, v_3, v_4 \rangle$. To continue the DFS for Q_5, we must find all vertices adjacent to each of $\langle v_1, v_3, v_4 \rangle$. NOTI computes the intersection of $T = \cap(N(v_1), N(v_3))$ and $U = \cap(N(v_1), N(v_4))$; these sets are already stored in the RTM. Existing multi-way join algorithms intersect the same sets repeatedly, a suboptimal approach. For example, they intersect $\cap(N(v_1), N(v_3))$ both to find the children of $\langle v_1, v_2, v_3 \rangle$ and $\langle v_1, v_3 \rangle$.

4.2 The RTM Index

The RTM index stores triangles in a compact binary format. We represent triangles using a binary matrix with each row and column associated with a neighbor of the root vertex. Instead of including all vertex ids, we compress the adjacency matrix to the size of $|N(v)|$ with v the root vertex. To do so, we map all neighbors of v to projection ids while preserving ordering.

Definition 4.1 (Rooted Triangle Matrix). Let v be a vertex in input graph G. Let $f : N(v) \rightarrow [1, |N(v)|]$ be a bijection mapping each vertex $u \in N(v)$ to a unique projection id such that $i < j \implies f(u_i) < f(u_j)$. The RTM M of root vertex v is defined as:

$$M_{ij} = \begin{cases} 1 & \text{if } f^{-1}(j) \in N(v) \cap N(f^{-1}(i)) \\ 0 & \text{otherwise.} \quad \square \end{cases} \tag{1}$$

The RTM is a binary matrix with each neighbor of v assigned that row and column corresponding to its projection id. The entry M_{ij} is 1 if and only if $u_j = f^{-1}(j)$ is a common neighbor of v and $u_i = f^{-1}(i)$, i.e., there exists a triangle $\langle v, u_i, u_j \rangle$.

Properties. Every time a multi-way join operator adds a new vertex to a partial match p, it intersects the adjacency lists of some vertices in p. The RTM index instead performs a triangle intersection.

Many vertices in a partial match are neighbors, as the multi-way join operator is used for dense query graphs. In the example of Q_5 which we previously discussed, adding vertices to the partial match $\langle v_1, v_2, v_3 \rangle$ requires intersecting the adjacency list of the three vertices in the partial match, which are all neighbors to v_1.

The fundamental property of an RTM index for a root vertex v is that we efficiently intersect the adjacency lists of any subset of neighbors of v with $N(v)$ by intersecting triangles stored in M. We do so by replacing set intersection with a bitwise-logical-and operation on rows of M, which we efficiently compute using SIMD operations. We call the result of this operation a triangle intersection vector.

Definition 4.2 (Triangle intersection vector). Let v be a vertex in the input graph G, M be an RTM for a root vertex v and $f : N(v) \rightarrow [1, |N(v)|]$ be the bijection mapping a vertex to its projection id, and $U \subseteq N(v)$. A *triangle intersection vector* T for U is a binary vector defined as:

$$T = \&_{u \in U} M_{f(u)} \tag{2}$$

where & denotes the bitwise-logical-and operator.

If we want to compute the intersection $N(v) \cap N(u_1) \cap \ldots \cap N(u_k)$, where $U = \{u_1, \ldots, u_k\}$ is a subset of neighbors of v, we can compute T for U using Eqn. 2 and look at its entries. To check whether a vertex u' is in the intersection we can check whether $T_{f(u')} = 1$, as shown by the following lemma.

Lemma 4.3. *Let v be a vertex in the input graph G, $U \subseteq N(v)$ a subset of neighbors of v, and T a triangle intersection vector for U. It holds for each $u' \in N(v)$ that:*

$$T_{f(u')} = \begin{cases} 1 & \text{if} \quad u' \in \cap_{u \in U} \big(N(u)\big) \cap N(v) \\ 0 & \text{otherwise} \end{cases}$$

Proof. It follows from Eqn. 1 and the definition of T that:

$$T_{f(u')} = 1 \quad \text{iff} \quad \forall u \in U,\, u' \in N(v) \cap N(u)$$

or equivalently:

$$T_{f(u')} = 1 \quad \text{iff} \quad u' \in S = \cap_{u \in U} \big(N(v) \cap N(u)\big).$$

The set S is equal to $\cap_{u \in U} \big(N(u)\big) \cap N(v)$. Therefore, the binary vector T encodes the set intersection between the adjacency lists of v and all the vertices in U. $\square$

4.3 The NOTI Multi-Way Join Algorithm

The NOTI algorithm takes the inputs for a physical implementation of a multi-way join operator described in Section 2: data and query graphs, a matching order and canonicality constraints.

Algorithm 1 shows the NOTI algorithm. NOTI starts by determining which query vertex's matches in the input graph will become the roots of the RTM instances (line 2). It selects the query vertex with the largest number of triangles in the query graph formed only with vertices later in the matching order. The boolean predicate $\Delta_Q(q_k, q_i, q_j)$ states that the three vertices form a triangle in Q. In

Algorithm 1 NOTI Algorithm

```
1:  procedure SubGraphSearch(G, Q, C)
2:      r = argmax_{k∈[1,n]} |{(q_i, q_j) : Δ_Q(q_k, q_i, q_j)&(k < i, j)}|
3:      V_1 ← set of vertices matching q_1
4:      for v ∈ V_1 do
5:          p[1] ← v
6:          if r = 1 then
7:              M ← BuildRTM(v, G, C)
8:          v_r ← v
9:          rtmGraphSearch(G, Q, C, p, M, v_r)
10:         discard M
11: procedure rtmGraphSearch(G, Q, C, p, M, v_r)
12:     if |p| = |Q| then return p
13:     k ← |p| + 1
14:     if k > r and (q_r, q_k) ∈ E_Q then          ▷ Case 1: RTM built
15:         I ← {i < k : Δ_Q(q_r, q_i, q_k)}
16:         T ← &_{i∈I} M[p.b[i]]
17:         for h < k : (q_k, q_h) ∈ E_Q ∧ (q_r, q_h) ∉ E_Q do
18:             if N(v_r)[i] ∉ N(p[h]) then
19:                 T[i] = 0
20:         mask ← canonicalMask(p, C, N(v_r))
21:         T ← T & mask
22:         J ← {j : T[j] = 1}
23:         for j ∈ J do
24:             p' ← p
25:             p'[k|] ← (N(v_r)[j])
26:             p'.b[|k|] ← j
27:             rtmGraphSearch(G, Q, C, p', M)
28:     else                                       ▷ Case 2: RTM not built
29:         I ← {i ∈ [1, . . . , k − 1] : q_i has edge to q_k}
30:         V ← {∩_{i∈I} N(p[i])}
31:         for v ∈ V do
32:             if isCanonical(p, v, C) then
33:                 p' ← p
34:                 p'[k|] ← v
35:                 p'.b[|k|] ← ⊥
36:                 if k = r then
37:                     M ← BuildRTM(v, G, C)
38:                     v_r ← v
39:                 rtmGraphSearch(G, Q, C, p', M, v_r)
```

the case of Q_5, we iterate over the vertices in the query graph and find that q_1 has 2 triangles made with vertices that come later in the matching order, q_2 and q_3 each have one, and q_4 and q_5 have none, and so select q_1. The algorithm then initializes the set V_1 of vertices that match q_1 (line 3). For each vertex v in V_1 (line 4), we initialize the partial match p as v (line 5). We check if the first index of the query graph has the most triangles (as is the case in Q_5) (line 6) and if so, build the RTM instance M as described by Eqn. 1 (line 7). We initialize our current vertex, v_r as the root vertex (line 8), and at that point are ready to begin the DFS, which is performed by recursively invoking rtmGraphSearch (line 9).

If the partial match is a full match (line 12), the algorithm returns the match and recurses. Otherwise, rtmGraphSearch initializes k as the next vertex in the matching order (line 13) and finds the vertices that match the next query vertex q_k and recurses on them. The algorithm considers two cases: (i) q_k is a neighbor of the root query vertex q_r and we can use the RTM index (lines 14-27); (ii) q_k is not a neighbor of q_r, so we cannot use the RTM index and must instead use regular set intersection (lines 28-39).

Case (i). In this case we have already built an RTM (since $k > r$) and vertices matching q_r are neighbors or the root vertex v_r of the current RTM. To match q_k we intersect the adjacency lists of the vertices q_i neighboring q_k in the query graph. If q_i is a neighbor or q_r, then there is a triangle (q_r, q_i, q_k). We can find matches of q_k by intersecting all such triangles using the RTM index. NOTI first builds a set I of the indices of all query vertices forming a triangle with q_r and q_k and then computes the triangle intersection vector T (lines 15-16). The entry $p[i]$ of the partial match p contains the id of the vertex matching q_i, and $p.b[i]$ contains the corresponding projection id. The latter is used to retrieve the correct row of RTM index M, as required by Eqn 2.

Some neighbor q_h of q_k might not form a triangle with q_r. In that case, we invalidate the entries of T corresponding to the vertices that are not members of the adjacency list of $p[h]$, which is the vertex matching q_h in p (lines 17-19). We then eliminate those vertices that do not respect the canonicality constraints (lines 20-21). We postpone our discussion of canonicality for ease of exposition. We build a set of matches of the triangle intersection vector T, J (line 22). Those indices that are 1 are those that correspond to the next vertex and do not violate the canonicality constraint. Finally, NOTI iterates over all matches in the triangle intersection vector T, expands the partial match p with them, and then recursively calls the rtmGraphSearch procedure (lines 23-27).

Case (ii). NOTI does not have an active RTM ($k \leq r$), or the new query vertex q_k is not a neighbor or q_r. NOTI performs regular set intersection between adjacency lists (line 28). In this case, we construct I as the set of those vertices earlier in the matching order that have an edge to our current vertex (line 29). We initialize V as the set of vertices that satisfy each of these edges and construct it by performing regular set intersection (line 30). We again postpone our discussion of the canonicality filtering on line 32.

We extend the partial match p (lines 33 - 35). If k matches the query vertex that we use to build the RTM, we build a new RTM instance and initialize our base vertex (lines 37 - 38). Finally, we recursively call rtmGraphSearch (line 39).

Canonicality. Canonicality constraints filter out redundant results, or automorphisms (see Section 2.1). These can be enforced as a filter step after all matches are generated. In doing so, we generate matches to the query that are automorphic to each other only to discard them. If the query graph is a triangle and $\langle v_1, v_2, v_3 \rangle$ forms a triangle in the data graph, we would generate all permutations of $\langle v_1, v_2, v_3 \rangle$ only to discard all but one. This is suboptimal.

To avoid generating redundant matches, NOTI enforces canonicality constraints during query execution. This is analogous to pushing down selection predicates in traditional databases and serves as a similar optimization. We only include canonical triangles in the NOTI, which helps keep the NOTI small. When using the RTM to perform the matching, NOTI generates a binary *canonical mask* to represent canonicality constraints and enforce them using bitwise-logical-and operations (line 20).

NOTI enforces the constraint that the order of the projection ids must be consistent with the order of the ids of the vertices in $N(v_r)$, where v_r is the root of the RTM. Suppose that we are matching query vertex q_k. Let the previous vertex be q_i, and furthermore let $q_i < q_k$. To generate our canonical mask, we find that vertex $v_j \in N(v_r)$ with the smallest id larger than that of of the previously matched vertex (v_n). We create a mask with all bits with index smaller than n set to 0 and all other bits set to 1. Performing a bitwise-and with this mask precludes any projection ids that violate our consistency constraint. Similarly, if we have $q_i > q_k$, we find that vertex $v_j \in N(v_r)$ with the largest id smaller than that of the previously matched vertex. We then create a mask with all bits with index greater than n set to 0 and all others set to 1, so that we can only match vertices with id smaller than n.

If there are multiple canonicality constraints (i.e., the current vertex we are matching must be larger than some vertex and smaller than some other vertex), CANONICALMASK performs the bitwise-logical-and of all the corresponding masks to ensure that all constraints are satisfied.

If NOTI does not use the RTM to match the next query vertex, it explicitly checks whether a new matching vertex is consistent with the canonicality constraints (line 32), and only then extends the partial match.

Multi-threading. Thus far, we have described the NOTI algorithm as a single-threaded algorithm. There are two strategies for parallelizing NOTI. In the first, we statically partition the set of vertices matching the first query vertex among the threads. Each such vertex is the root of a subtree in the matching tree, which can be visited by each thread independently. In the second, we dynamically assign vertices to threads whenever they are idle.

5 OPTIMIZATIONS

To further improve the efficiency of the core NOTI algorithm, we use several optimizations.

5.1 Optimizing Set Intersection

When a query vertex cannot be matched using the RTM, NOTI performs set intersection among adjacency lists. We implemented SIMD-optimized algorithms that do not require input indexing [10, 30], though one could combine NOTI with input indexing techniques to further optimize set intersection.

5.2 Handling Triangle Intersection Vectors

We developed dedicated algorithms to handle triangle intersection vectors, a key potential bottleneck. We propose an optimization technique for CPUs that do not support SIMD operations [17] and one for CPUs that support SIMD. If the CPU of the system supports SIMD, the NOTI algorithm defaults to the SIMD-based optimization. Even though the SIMD-based optimization requires shuffling and compacting the vector, we do so branchlessly, which results in better overall performance.

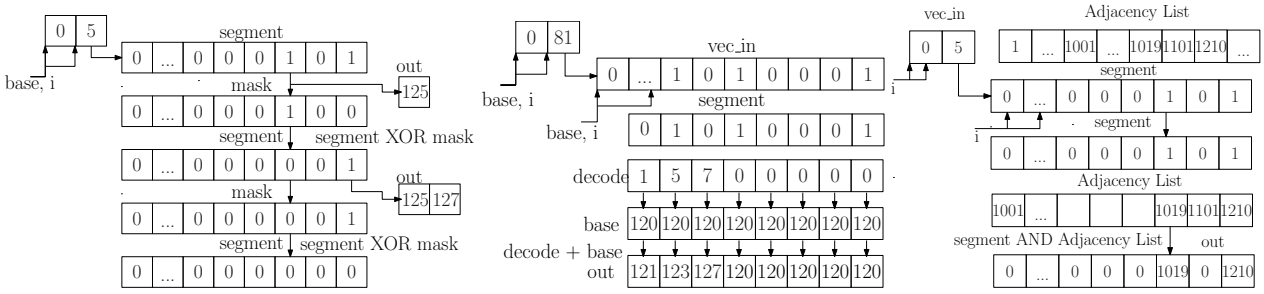
Expanding Indices in the RTM. In Algorithm 1, after computing the triangle intersection vector T, the algorithm expands the partial match p with each match of the triangle intersection, then recurses. To expand p and recurse efficiently, the EXPANDTOINDEX procedure computes the set of indices that match the triangle vector; we call this set J (line 22). The most intuitive approach iterates over the entries of the vector to find the indices equal to 1. This process incurs significant overhead. Therefore, we propose several techniques to alleviate this overhead.

Non-SIMD version. The first implementation of EXPANDTOINDEX uses regular CPU instructions. We represent the triangle intersection vector T as an array of 64 bit values. For each element i of the array, we count the leading zeros to find the index position of the first 1, denoted as p. We then add the position $64 \cdot i + p$ to J. We unset the bit we found and repeat the above steps until all 1 bits are found, and then move to the next element. To unset a bit, we create a mask containing a 1 at its corresponding position and XOR the mask with the element.

Figure 2a shows an example of a triangle intersection vector T represented as an array of 64 bit elements ($vec\_in = \langle 0, 5 \rangle$). Let i be the index in the array vec_in, $segment = vec\_in[i]$ be the 64 bits that we are currently considering, and $base$ be our offset from the beginning of the array. When i equals 0, base is 0, when i equals 1, base is 64, and so forth. Our initial $segment$ is 0. As there are no 1s, we increment i and $base$. Our next $segment$ is 5, or $\langle 0, \ldots, 0, 1, 0, 1 \rangle$, so we need to find the set bits in $segment$. We use the CPU intrinsic __BUILTIN_CLZLL() to get the number of leading zeros: in this case, there are 61. Thus, there is a 1 in position 64 + 61, so we add 125 to our out vector. We unset the bit we just counted using an XOR with a mask. Our segment is now $\langle 0, \ldots, 0, 0, 0, 1 \rangle$. We then use __BUILTIN_CLZLL() to find that the last set bit is the 63rd bit. It is therefore the 64+63rd bit in our vec_in, so we add 127 to our output. Our output is $\langle 125, 127 \rangle$ and the $indexSize$ is 2. This is correct, as the 125th and 127th bits of T were set. Thus, we can use CPU intrinsics to efficiently expand an incidence vector to an index.

SIMD version. Our SIMD algorithm to expand indices works similarly, though we examine multiple bits in parallel and do not unset bits. We use AVX2 instructions, which allow the CPU to fetch, modify and write contiguous 256-bits of data in parallel. To use the AVX2 instructions, we break our index into 32-bit segments rather than 64-bit segments. AVX2 instructions either set, load, add or store 8 32-bit integers. Our inputs and outputs are identical to the non-SIMD EXPANDTOINDEX described previously.

Consider the input $\langle 0, 81 \rangle$, pictured in Figure 2b. We initialize i and $base$ to 0, the beginning of the input vector. As our initial input is 0, we have no set bits and increment i and $base$. The next 64-bit segment is 81, or $\langle 0, 0, \ldots, 0, 1, 0, 1, 0, 0, 0, 1 \rangle$. We load the first 8 bits ($\langle 0, 0, 0, 0, 0, 0, 0, 0 \rangle$) into $segment$. The $decodeTable$ is a look-up table that maps binary numbers to their respective indices. Thus, $decodeTable[0] = \langle 0 \rangle$ as there are no set bits in 0, $decodeTable[1] = \langle 7 \rangle$ as the 7th bit in 1 is set, $decodeTable[2] = \langle 6 \rangle$ and $decodeTable[3] = \langle 6, 7 \rangle$, and so on. As $decodeTable[0]$ corresponds to 0, we add 0 to our out vector. The length table is a look-up table that maps binary numbers to the number of indices they contain. As there are no 1s in our segment, the length table is 0; we have found no set bits. We then increment i and $base$ to examine the next 8 bits of the segment. This process repeats 7 times until we reach the $segment$ $\langle 0, 1, 0, 1, 0, 0, 0, 1 \rangle$. At this point, our base is $64 + 56 = 120$, as we have reached the 120th bit. The decode table returns the indices $\langle 1, 3, 7 \rangle$. As these are at an offset of 1, 3, and 7 from the 120th



(a) Non-SIMD ExpandToIndex Example **(b) SIMD ExpandToIndex Example** **(c) SIMD ExpandToID Example**

Figure 2: Efficient ExpandToIndex and ExpandToId Algorithms

Algorithm 2 Query-Specific Code for the 4-Clique Query

```
1:  procedure 4-Clique(G, C)
2:      V_1 ← G.V
3:      for v_1 ∈ V_1 do
4:          p_1 ← ⟨v_1⟩
5:          M ← BuildRTM(v_1)
6:          mask_2 ← canonicalMask(p_1)        ▷ recursion for k = 2
7:          T_2 ← [1, ..., 1]
8:          T_2 ← T & mask_2
9:          J_2 ← {j : T[j] = 1}
10:         for j_2 ∈ J_2 do
11:             p_2 ← p_1
12:             p_2.b[2] ← j_2
13:             mask_3 ← canonicalMask(p_2)        ▷ k = 3
14:             T_2 ← M[p_2.b[2]] & mask_3
15:             J_3 ← {j : T[j] = 1}
16:             for j_3 ∈ J_3 do
17:                 p_3 ← p_2
18:                 p_3.b[3] ← j_3
19:                 mask_4 ← canonicalMask(p_3)        ▷ k = 4
20:                 T_4 ← M[p_3.b[2]] & M[p_3.b[3]] & mask_4
21:                 J_4 ← {j : T[j] = 1}
22:                 for j_4 ∈ J_4 do
23:                     p_4 ← p_3
24:                     p_4.b[4] ← j_4
25:                     return p_4
```

bit, we store $\langle 121, 123, 127\rangle$ in *out* using the AVX2 instructions. As there are 3 set bits in *segment*, $lengthTable[0, 1, 0, 1, 0, 0, 0, 1]$ returns 3, so we know that we have stored 3 numbers in *out*. This way, even though we necessarily store 8 values at a time using AVX2, the return value p allows us to know to discard any indices after the pth index. The output $\langle 121, 123, 127\rangle$ is correct as the 121st, 123rd and 127th bits of our input incidence vector are set.

Expanding IDs. When we match the last vertex in our subgraph query, we need the bit indices corresponding to vertex ids, which we fetch from the adjacency list of the query vertex. We do so by expanding the indices of the triangle intersection and then accessing the appropriate vertex ids from the adjacency list. To accelerate this specialized case, we develop an expandToId function that completes both steps synchronously. The approach is similar to that of SIMD expandToIndex, but instead of outputing the indices of the set bits, we output the vertex ids to which the indices correspond. In addition to *vec_in*, we take the adjacency list of our vertex as input. We fetch 64 bits at a time. When the 64 bits are nonzero, we fetch 16 bits at a time from them using the AVX512 intrinsics. If any of these are nonzero, the corresponding vertex ids are added to our output using an AVX512 intrinsic that allows us to use our incidence vector as a mask for the adjacency list.

Consider the case pictured in Figure 2c. Our *vec_in* is $\langle 0, 5\rangle$. We fetch the first 64 bits, skip them as they are 0, and then fetch the next 64 bits. We then fetch 16 bits at a time until we reach the last 16 bits. As these are nonzero, we fetch the corresponding portion of the adjacency list. These ids are $\langle 1001, ..., 1019, 1101, 1210\rangle$. We mask the ids using the segment, and get $\langle 1019, 1210\rangle$ as output: these are the vertex ids to which the input incidence vector corresponds. Now that we have iterated through all bits in the incidence vector, we return *out*.

For both SIMD algorithms (expandToIndex and expandToId), if *vec_in* is not a multiple of 32 or 64, respectively, we iterate over the last bits. For example, if *vec_in* is n bits, the last $n \mod 32$ bits are parsed without SIMD for expandToIndex.

5.3 Query-Specific Code

We can use the version of NOTI described in Algorithm 1 to compute the output for any query. However, this generality is less efficient than query-specific code. Since the cost of compiling query specific code is generally much smaller than the cost of computing the query, we compile queries to optimized query-specific code. In the case of NOTI, it is possible to statically eliminate several checks that only depend on the query graph and not on the input graph a priori. Algorithm 1 recursively calls the rtmGraphSearch procedure. Rather than recursing, one can unroll the calls to the procedure as the number of nested calls is fixed and depends on the size of the query. This results in a nested loop structure, with each level corresponding to an invocation of the function subgraph search. After this nested loop, we can directly output the final match without executing the check on line 12. Since the query structure is known, we know whether or not the RTM instance has been initialized and do not perform the checks on 14 and 28. We also unroll the loop of lines 17-19 since we statically know the values of h. Similarly, since the set I of lines 15 and 29 are query-dependent,

Graph	Vertexes	Edges	Triangles	Clustering Coefficient
amazon	334,863	925,872	667,129	0.3967
youtube	1,134,890	2,987,624	3,056,386	0.0808
lj	3,997,962	34,681,189	177,820,130	0.2843
orkut	3,072,441	117,185,083	627,584,181	0.1666

Table 1: Input Data Graphs

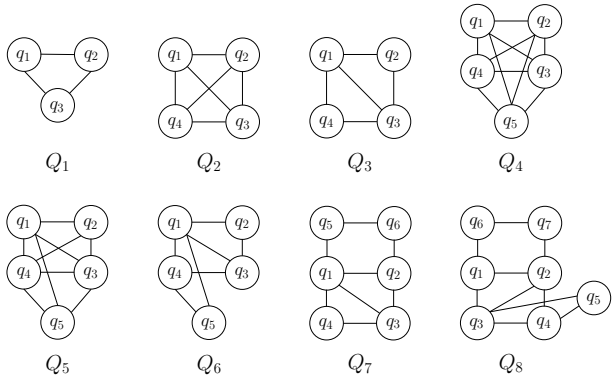


Figure 3: Query Graphs

we can statically determine how to perform the bitwise-logical-and operations of line 16 and the set intersections of line 30. The index r of the root vertex is known at compile time, making the checks of lines 36 and 6 unnecessary. Finally, in some cases, the canonical masks can be pre-defined based on the constraints C. The CANONICALMASK function can be optimized statically as well. Rather than performing a bit-wise-and or checking if a given partial match satisfies canonicality, we only check for those matches that already satisfy canonicality statically, skipping over those positions which the bit-wise-and would eliminate. Iterating instead of recursing as well as the optimizations listed above result in this version performing better in practice than the more general algorithm. Algorithm 2 provides an example of how to transform Algorithm 1 to a 4-clique query specific version.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of NOTI and compare it with state-of-the-art input indexing algorithms.

6.1 Baselines and NOTI Variants

Standard Multi-Way Join. To evaluate the impact of using local online indexing, we use a *standard* multi-way join implementation as a baseline (`std`). This baseline employs the backtracking vertex-at-a-time algorithm described in Section 2. To make the comparison with NOTI as fair as possible, we use the same codebase for both algorithms. NOTI extends the baseline to use the RTM index. We also use a variant of the baseline that uses the same SIMD optimizations for set intersection as NOTI (`std-SIMD`).

NOTI. We consider a few variants of NOTI in the experiments. The basic NOTI version (denoted `NOTI`) uses our non-SIMD method `expandToIndex` to build the matrix and handle projection indices. `NOTI-v1` version uses AVX2 and the SIMD `expandToIndex` algorithm. The `NOTI` and `NOTI-v1` variants do not use `expandToId`, while the `NOTI-v2` variant does, as described in Section 5.

Canonicality. We use canonicality constraints from GraphPi for all non-clique queries [31], and refer the reader to their work for a complete description of the constraints. For clique queries, we filter automorphisms by only considering those neighbors of a vertex with larger vertex id than the vertex. We call this the $N+$ *neighborhood* of the vertex.

EmptyHeaded. EmptyHeaded is a query execution engine that leverages multi-way joins [1]. It performs input indexing by using a fixed bitset encoding for the entire graph. A binary representation is only used for dense areas of the graph. Emptyheaded compiles datalog queries into efficient low-level code. It holds the query results in memory, which leads to system out of memory (OOM) in some experiments. EmptyHeaded is a full-fledged query execution system, which includes a query optimizer. Other systems we compare to only run a physical implementation of a known query plan. Emptyheaded does not support canonicality, so we can only fairly compare it to NOTI and BSR on cliques on which we can filter the input graph using the $N+$ neighborhood.

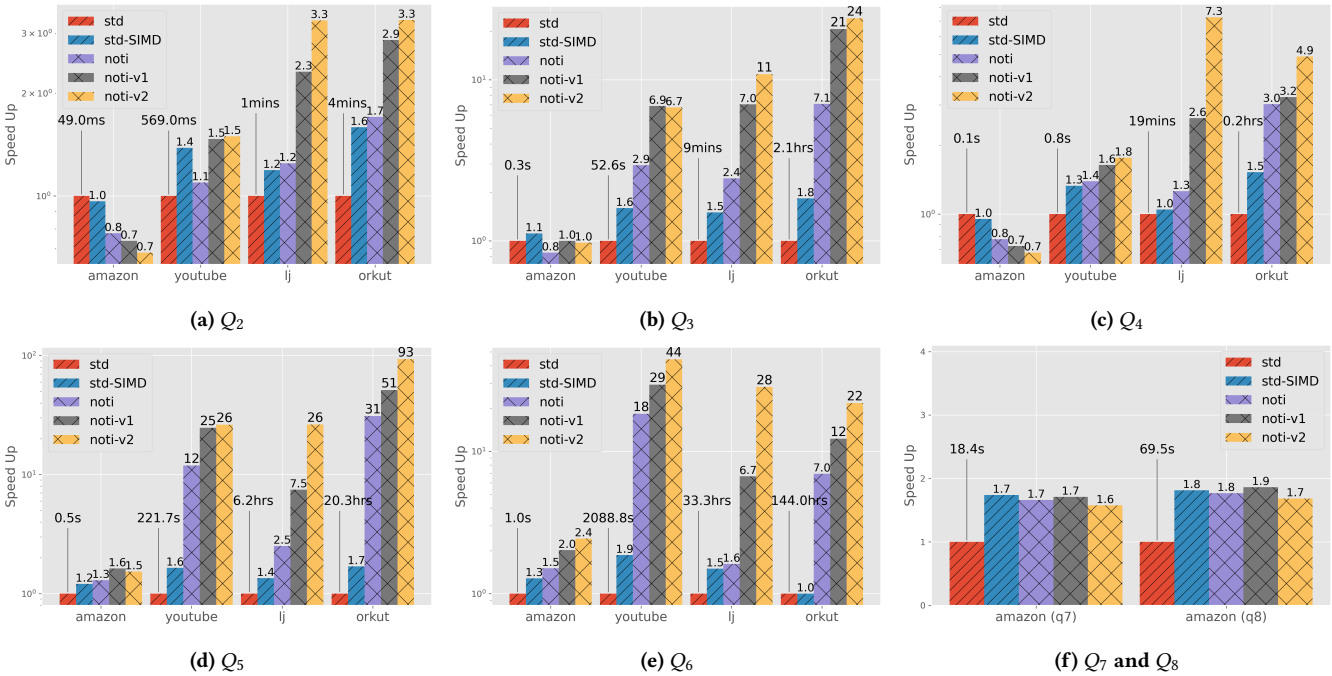
BSR, QFilter and GRO. QFilter is the state-of-the-art algorithm to speed up set intersection in graphs [7], achieving substantial speedups over EmptyHeaded's multi-way join implementation. QFilter performs input indexing on the entire graph using a representation called BSR. BSR divides the domain of 32-bit vertex ids into different segments of range 32. Due to BSR, QFilter can use the bit-wise-logical-and operator to perform set intersection on the segments associated with the same base value. QFilter also uses SIMD instructions to perform bit-wise-logical-and operations on multiple segments in parallel. The density of the state segments has a significant impact on QFilter's efficiency. GRO is a graph reordering technique that can be applied on the input graph before the graph is indexed by BSR [7]. It results in a more tightly packed bit encoding for BSR but requires expensive preprocessing. Our `QFilter` baseline extends our standard multi-way join implementation with the QFilter algorithm for set intersection. We use GRO reordering by default.

Both NOTI and QFilter run query-specific code that implements a specific query execution plan for the particular query being executed. We use the same plan for both algorithms.

6.2 Evaluation Setup

Data and query graphs. The real-world graphs we use in our evaluation are taken from SNAP [18]and shown in Table 1. All graphs fit in memory and are passed to all algorithms using the same edge pairs format. The algorithms then load the graphs into their internal representation.

Figure 3 shows the query graphs we use in the evaluation. We use these as they represent dense subgraphs queries similar to those on which we expect query optimizers to schedule multi-way joins. The basic triangle query Q_1 is used only for micro-benchmarks that evaluate the cost of building the RTM index. We also include queries with squares (Q_7 and Q_8). For queries that do not include



(a) Q_2 **(b)** Q_3 **(c)** Q_4

(d) Q_5 **(e)** Q_6 **(f)** Q_7 **and** Q_8

Figure 4: Subgraph Enumeration Performance of NOTI vs. the Standard Algorithm

any triangles, there is no difference between NOTI and a standard multi-way join.

We consider two variants of query execution. In addition to the *subgraph enumeration* problem, we also consider the *subgraph counting* problem. Here, rather than enumerating all the matches of the query in the input graph and materializing these matches, we only count how many matches there are without storing the results in a memory buffer. NOTI performs better counting subgraphs than it does enumerating them as we use a compressed format and the time it takes to expand the ids is significant.

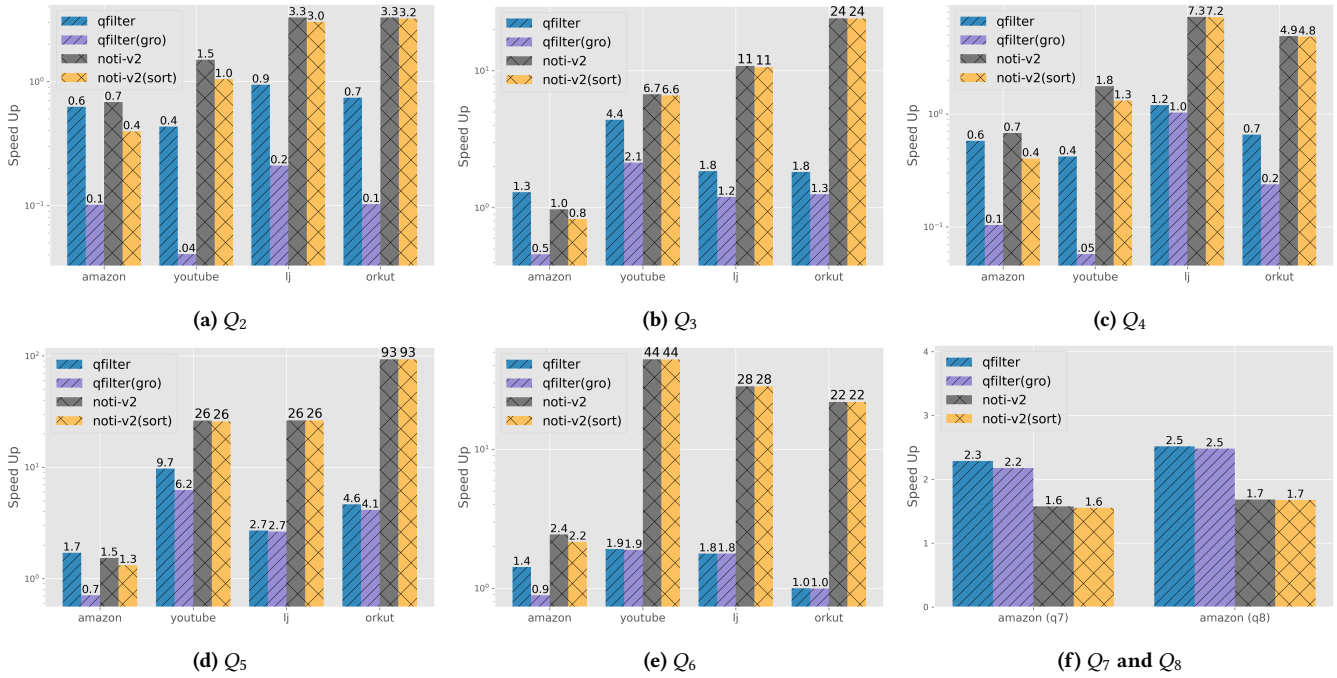
Hardware and software setup. We ran all our experiments on a server class machine running Ubuntu 18 with 32 cores (64 threads) and 128 GB of RAM. The machine has two Intel(R) Xeon(R) Silver 4216 CPUs. All tests use one CPU thread unless otherwise stated. Our software is written in C++ and compiled with gcc 4.9 (-O3).

6.3 NOTI Performance

NOTI vs. standard. We begin by comparing NOTI to the standard algorithm on a variety of graphs and queries. As we use the same codebase for both, we use this test to evaluate the effect local online indexing has on query runtime. Figure 4 reports the performance of the two methods. The `youtube`, `lj`, and `orkut` graphs are larger than `amazon`. Overall, vectorization is more beneficial on larger graphs. NOTI is faster than the standard algorithm, and the speedups become more substantial with larger input graphs. For example, on Q_5, we improve on the standard by a factor of 1.5 on `amazon`, a small graph, a factor of 26 for `youtube` and `lj`, larger graphs, and a factor of 93 for `orkut`, a very large graph. In addition, we see a positive trend for queries with more triangles. On the `youtube` graph, for Q_2 (1 triangle) we see a speedup of 1.5x, for Q_4 (two triangles) a speed up of 1.8x, and Q_3 (4 triangles) a speed up of 6.7 times. In addition, larger input graphs and queries take longer in absolute time and this makes the relative speedups of Figure 4 more significant. On the `amazon` graph, the performance of NOTI is comparable to the standard method. The `amazon` graph has a low average degree, so has limited opportunities for vectorization as we must use non-SIMD instructions to handle edge cases. Thus, we see that merely adding SIMD set intersection to the standard implementation does not significantly improve performance, with the exception of queries Q_7 and Q_8.

For queries Q_7 and Q_8, the last query vertex is matched using a regular adjacency list intersection. These queries are much more expensive than the others, so we could only run them on the `amazon` graph. The running time with these queries is dominated by the speed of set intersection. That is why the standard algorithm with SIMD-optimized adjacency list intersection performs similarly to NOTI for these queries.

Comparing NOTI variants. Using SIMD instructions to convert triangle intersection vectors to projection indices, as done by the variants `NOTI-v1` and `NOTI-v2`, results in significant speedups in almost all cases. The best performance is typically achieved by `NOTI-v2`, which also uses SIMD instructions to convert projection indices into vertex ids. This operation is particularly important in the subgraph enumeration task, since it requires returning the ids of the vertices in each match as output. By processing 16 vertexes in the adjacency list in parallel, this version further improves RTM's performance significantly in large graphs.



(a) Q_2 **(b)** Q_3 **(c)** Q_4

(d) Q_5 **(e)** Q_6 **(f)** Q_7 **and** Q_8

Figure 5: Subgraph Enumeration Performance of NOTI vs. Input Indexing Approaches

The benefit of using SIMD `expandToIndex` varies depending on the input and query graph. On `amazon` and `youtube`, building the RTM dominates `expandToIndex`. On `lj` and `orkut`, SIMD leads to better performance in most cases, with the exception of Q_4, where the non-SIMD method performs slightly better. Q_4 is the densest query graph we consider and therefore generates the sparsest triangle intersection vectors. This favors our non-SIMD `expandToIndex`. For queries Q_7 and Q_8, the difference between the variants is smaller since there are fewer query vertices we match using the RTM.

Maximum size of the RTM. The RTM index is very compact due to our careful selection of root vertices and support for arbitrary canonicality. Across all queries and all graphs, its average case size is almost always less than one kilobyte (see Table 2). The one exception to this is the large `orkut` graph on non-clique queries, where the average size of the RTM was 3.75 KB, which is still very compact. The RTM tends to be smaller for clique graphs, as the $N+$ filter allows us to not build many possible triangles associated with a given root vertex. Even on non-clique queries for which we can filter fewer potential children through the use of canonicality, the RTM index is small: the largest RTM instance obtained was on the `lj` graph and its size was 274.3 Megabytes and the average case size was still only 0.28 kilobytes.

6.4 Input vs. Local Online Indexing

We now compare the performance of NOTI, which uses local online indexing, with state-of-the-art input indexing algorithms: EmptyHeaded and QFilter.

Preprocessing costs. Both QFilter and NOTI benefit from reordering the input graph. Sorting vertices by degree improves NOTI's performance. After sorting, the triangle intersection vectors progress from low degree vertices at the beginning of the vector to high degree vertices at the end of the vector. This creates sparse regions at the beginning of the intersection vectors which are quickly skipped and dense segments at the end which we quickly process with SIMD instructions, as described in Section 5.2.

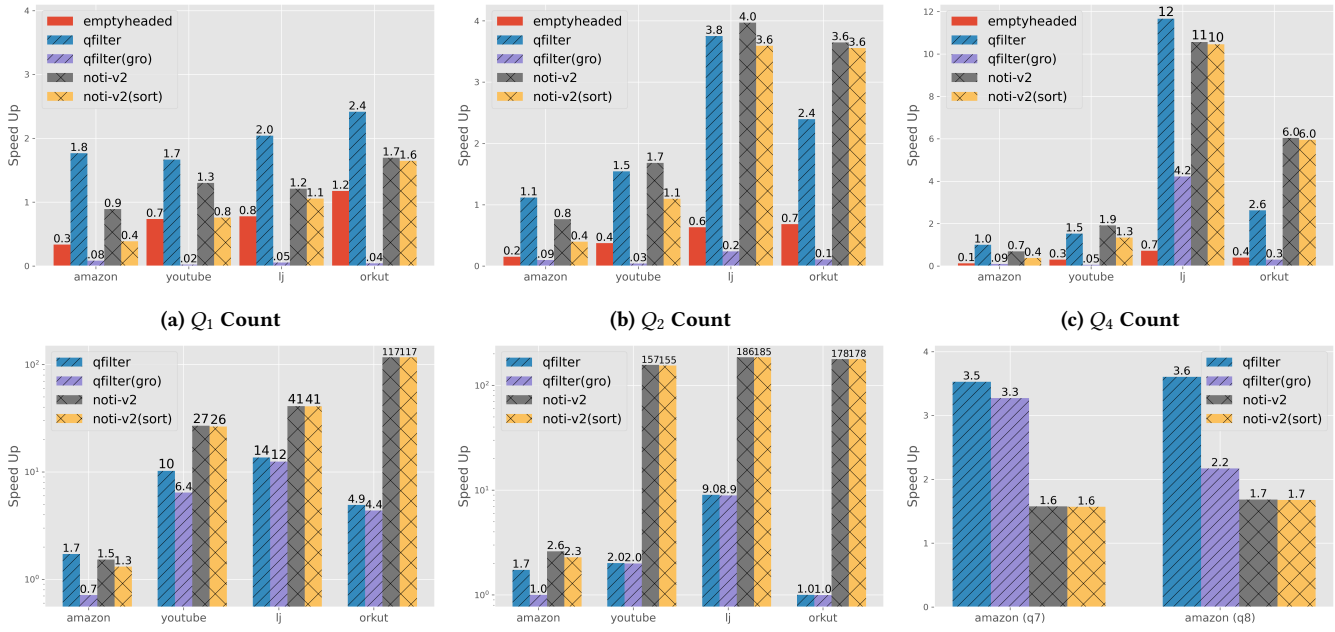
NOTI does not index the graph in advance, it performs a sorting which results in a small overhead, which we report in Table 3. The overhead is one order of magnitude lower than the graph loading time, so it is negligible. QFilter uses an ordering algorithm called

Graph	Clique		Other	
	Average	Maximum	Average	Maximum
`amazon`	0.002	0.007	0.01	37.9
`youtube`	0.005	3.43	0.33	103,370
`lj`	0.033	34,584	0.28	274,300
`orkut`	0.35	35,845	3.75	138,748

Table 2: RTM Size in KB

Graph	NOTI		QFilter	
	Sort	Load (CSR)	GRO	Load (BSR)
`amazon`	52	487	406	286
`youtube`	165	1,627	12,734	987
`lj`	1,017	21,917	152,423	13,135
`orkut`	1,478	80,330	1,900,059	45,588

Table 3: Graph Preprocessing Times (ms)



(a) Q_1 Count **(b) Q_2 Count** **(c) Q_4 Count**

(d) Q_5 Count **(e) Q_6 Count** **(f) Q_7 and Q_8 Count**

Figure 6: Subgraph Counting Performance of NOTI vs. Input Indexing Approaches Count

GRO to increases the compactness of BSR. This preprocessing step is a bottleneck in most cases and is at least one order of magnitude larger than the graph loading time for all graphs except amazon, which is very small.

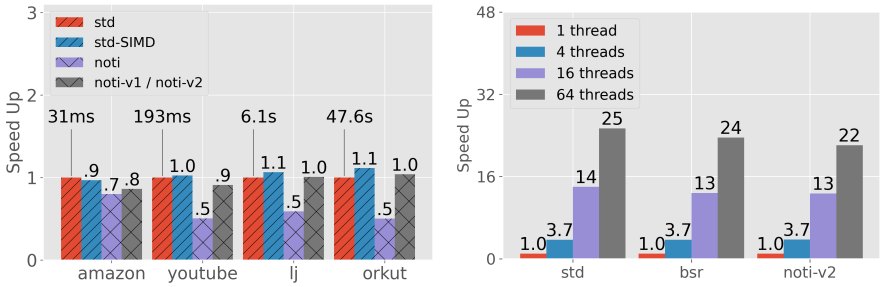
Enumeration. We measure the performance of BSR and NOTI in two scenarios. In the first scenario, we assume that the graph is pre-ordered. This is the case with a read-only graph that is queried repeatedly, so the reordering overhead can be amortized over multiple queries. In the second scenario, we consider graph reordering as part of the running time of the query. This corresponds to a graph that is queried only once. In addition, it provides a lower bound for the algorithm's efficiency on dynamic graphs, as in the case of a dynamic graph, the graph is already partially ordered.

Figure 5 reports the performance of BSR and NOTI algorithms in both scenarios. Running times are reported as speedups over the standard algorithm.We do not include a comparison to EmptyHeaded as EmptyHeaded stores query results in memory, which results in an out of memory error when evaluating Q_7 and Q_8.

NOTI provides a significant improvement over the standard algorithm and BSR for Q_2, Q_3, Q_4, Q_5 and Q_6 on graphs youtube, lj and orkut, performing almost an order of magnitude better than the standard algorithm, BSR and GRO on Q_6 on the large graphs lj and orkut. These queries have many triangles that we can leverage and larger graphs allow us to take full advantage of vectorization, which is why we see more of a speed-up from both NOTI and BSR for youtube, lj and orkut and less of a speed-up on amazon.

Q_7 and Q_8 are much slower than the other queries and could be run only for the amazon graph. The running time in this case is dominated by the set intersection required to match the last query vertex. Using an online SIMD-optimized set intersection algorithm can speed up these queries by a factor of up to 1.9x on the enumeration problem (see Figure 4f). QFilter can further speed up set intersection on those queries by up to 1.5x for enumeration (see Figure 5f) and 2.2x for counting (see Figure 6f) excluding preprocessing cost. QFilter can actually be seen as an online indexing algorithm in this case because the queries have a long running time (tens of seconds) and the cost of GRO (which is particularly low on amazon, see Table 3) can be amortized during the execution of one query. Therefore, we can replace the SIMD-optimized set intersection algorithm of NOTI with QFilter and achieve optimal performance. However, using GRO online is not always feasible. Q_7 and Q_8 are unlabeled, so they run on the entire graph and have low selectivity. For queries with higher selectivity and shorter running time, though, the preprocessing cost of QFilter can become a bottleneck because GRO is one order of magnitude slower than loading the entire graph, as discussed previously. Developing fast local online indexing algorithms to speed up square and other non-triangle cycle queries is an interesting direction for future work.

Counting. When tasked with subgraph counting, we use the binary vectors generated by RTM to accelerate pattern counting. In addition, we leverage CPU intrinsics that count the number of 1s in a register. Subgraph counting results, reported in Figure 6, show similar trends as subgraph enumeration: NOTI is substantially faster, especially when it comes to larger input and query graphs. For Q_7 and Q_8, QFilter is faster because it can be seen as an online algorithm. NOTI could thus use QFilter as set intersection algorithm and achieve performance parity.



(a) RTM Build Time **(b) Scalability with lj and Q2**

Figure 7: Micro-Benchmarks

6.5 Micro-Benchmarks

RTM Build Time. We now evaluate the overhead of building the RTM data structure. NOTI uses the RTM in queries that contain triangles, so its overhead compared to the standard baseline is the extra time we take to count triangles and to store them in the RTM. Figure 7a compares the running time of the standard algorithm and NOTI on the triangle query Q_1 with the counting task. The additional overhead of NOTI over the standard baseline on this query shows the cost of building the RTM data structure, since this cost will not be amortized by matching additional query vertices. The results show that building the RTM comes at minimal extra cost thanks to the use of SIMD instructions. The remaining overhead of building the RTM consists of bookkeeping metadata and memory management, which is comparably small.

Scalability. NOTI is easily parallelizeable, as discussed in Section 4. We found that even a simple static work partitioning approach, where each thread is assigned an equal partition of the vertices matching the first query vertex, scales reasonably well. Figure 7b shows the performance of NOTI with and other algorithms with different numbers of threads, on the `lj` graph, query Q_2, and the subgraph enumeration task. Here, we see that all methods scale comparably well given the same partitioning approach.

7 OTHER RELATED WORK

Section 2.2 discussed the prior work most directly related to NOTI. We now place this work in a broader context.

Graph databases are increasingly used in many applications and there is an increasing offering in terms of systems [2, 28, 29]. Subgraph search is a fundamental functionality in these systems. Join order has a significant impact on overall query execution time [24, 25]. Multi-way joins have been shown to be worst-case optimal, unlike binary joins [23]. This means that their worst-case complexity is in the order of the size of their output.

Much recent work has focused on algorithms for subgraph isomorphism and subgraph enumeration [9, 14–16] that focus on optimizing the average-case performance of subgraph search. Other work has focused on reusing computation [12, 33]. To improve on a strict DFS approach, Light introduces a Cartesian product operator across candidate sets in lieu of a simple loop [33]. Trie-cache caches intersections and reuses them [12]. These intersections must match both the number of vertices and the pattern vertex ids. This differs from our approach as we do not rely on exact matches and instead cache partial intersections. Both these optimizations are orthogonal to our work on dependent candidate sets and could be combined with NOTI.

Several frameworks [6, 19, 26, 32, 36, 37] tackle general graph processing. These frameworks provide an extensible high level task api while abstracting several low level components such as scheduling, graph-access, and parallelism. However, they are not particular well suited for graph mining problems such as motif counting and frequent subgraph mining. This has motivated the development of dedicated graph mining frameworks [3–5, 11, 20, 27, 34]. Recent graph mining systems work uses subgraph query matching as a subroutine [11, 20].

8 CONCLUSION

In this paper, we propose local online indexing as a novel approach to the execution of subgraph queries and multi-way joins. The fundamental idea is to store intermediate results obtained during the execution of a query into an index, which can be used to speed up subsequent recursive searching. Typically, multiple instances of the index can be built during the execution of a query, each local to a particular neighborhood of the input graph. A local online index is query-dependent and discarded at the end of the execution of the query. Online indexing is particularly well-suited to querying dynamic graphs, which are common in practice [28].

We also present NOTI, a dense-optimal implementation of the multi-way join operator based on the RTM local online index. The fundamental insight is that triangles can be indexed and intersected efficiently to match more complex patterns. Through reusing computation by storing triangles in a bitmatrix, NOTI outperforms state-of-the-art algorithms by roughly an order of magnitude on dense queries. The algorithm can be easily integrated with online SIMD-optimized set intersection algorithms to speed up sparser queries.

Local online indexing is a paradigm that goes beyond NOTI. Applying the paradigm to other algorithms such as supporting sparser query patterns is a promising avenue for future research.

REFERENCES

[1] C. R. Aberger, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: Boolean algebra based graph processing. *CoRR*, abs/1503.02368, 2015.

[2] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *arXiv preprint arXiv:1910.09017*, 2019.

[3] X. Chen, R. Dathathri, G. Gill, L. Hoang, and K. Pingali. Sandslash: A two-level framework for efficient graph pattern mining. *arXiv preprint arXiv:2011.03135*, 2020.

[4] X. Chen, R. Dathathri, G. Gill, and K. Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proc. VLDB Endow.*, 13(10):1190âĂŞ1205, Apr. 2020.

[5] V. Dias, C. H. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1357–1374, 2019.

[6] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. 2014.

[7] S. Han, L. Zou, and J. X. Yu. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD âĂŹ18, page 1587âĂŞ1602, New York, NY, USA, 2018. Association for Computing Machinery.

[8] S. Han, L. Zou, and J. X. Yu. Speeding up set intersections in graph algorithms using simd instructions. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1587âĂŞ1602, New York, NY, USA, 2018. Association for Computing Machinery.

[9] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 337âĂŞ348, New York, NY, USA, 2013. Association for Computing Machinery.

[10] H. Inoue, M. Ohara, and K. Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *Proceedings of the VLDB Endowment*, 8(3):293–304, 2014.

[11] K. Jamshidi, R. Mahadasa, and K. Vora. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[12] O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible caching in trie joins, 2016.

[13] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1695–1698, 2017.

[14] H. Kim, J. Lee, S. S. Bhowmick, W.-S. Han, J. Lee, S. Ko, and M. H. Jarrah. Dualsim: Parallel subgraph enumeration in a massive graph on a single machine. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1231–1245, 2016.

[15] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *Proc. VLDB Endow.*, 8(10):974âĂŞ985, June 2015.

[16] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang. Scalable distributed subgraph enumeration. *Proc. VLDB Endow.*, 10(3):217âĂŞ228, Nov. 2016.

[17] D. Lemire. Iterating over set bits quickly, Feb 2018.

[18] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135âĂŞ146, New York, NY, USA, 2010. Association for Computing Machinery.

[20] D. Mawhirter and B. Wu. Automine: Harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP âĂŹ19, page 509âĂŞ523, New York, NY, USA, 2019. Association for Computing Machinery.

[21] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692âĂŞ1704, July 2019.

[22] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *arXiv preprint arXiv:1903.02076*, 2019.

[23] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3), Mar. 2018.

[24] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *CoRR*, abs/1310.3314, 2013.

[25] D. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. RÃľ, and A. Rudra. Join processing for graph patterns: An old dog with new tricks, 2015.

[26] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 456âĂŞ471, New York, NY, USA, 2013. Association for Computing Machinery.

[27] A. Quamar, A. Deshpande, and J. Lin. Nscale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, 25(2):125–150, 2016.

[28] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment*, 11(4):420–431, 2017.

[29] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. Aref, M. Arenas, M. Besta, P. A. Boncz, et al. The future is big graphs! a community view on graph processing systems. *arXiv preprint arXiv:2012.06171*, 2020.

[30] B. Schlegel, T. Willhalm, and W. Lehner. Fast sorted-set intersection using simd instructions. *ADMS@ VLDB*, 1:8, 2011.

[31] T. Shi, M. Zhai, Y. Xu, and J. Zhai. Graphpi: high performance graph pattern matching through effective redundancy elimination. *arXiv preprint arXiv:2009.10955*, 2020.

[32] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 135âĂŞ146, New York, NY, USA, 2013. Association for Computing Machinery.

[33] S. Sun, Y. Che, L. Wang, and Q. Luo. Efficient parallel subgraph enumeration on a single machine. pages 232–243, 2019.

[34] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP âĂŹ15, page 425âĂŞ440, New York, NY, USA, 2015. Association for Computing Machinery.

[35] T. L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. *arXiv preprint arXiv:1210.0481*, 2012.

[36] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 763âĂŞ782, USA, 2018. USENIX Association.

[37] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens. Gunrock: Gpu graph analytics. *ACM Trans. Parallel Comput.*, 4(1), Aug. 2017.

[38] X. Zhu, G. Feng, M. Serafini, X. Ma, J. Yu, L. Xie, A. Aboulnaga, and W. Chen. Livegraph: a transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment*, 13(7):1020–1034, 2020.