VertexSurge: Variable Length Graph Pattern Match on Billion-edge Graphs

Weiyu Xie
Tsinghua University
Beijing, China
xwy21@mails.tsinghua.edu.cn

Mingxing Zhang
Tsinghua University
Beijing, China
zhang_mingxing@mail.tsinghua.edu.cn

Xia Liao
Tsinghua University
Beijing, China
liaoxia5018@163.com

Kang Chen
Tsinghua University
Beijing, China
chenkang@tsinghua.edu.cn

Jinlei Jiang
Tsinghua University
Beijing, China
jjlei@tsinghua.edu.cn

Yongwei Wu
Tsinghua University
Beijing, China
wuyw@tsinghua.edu.cn

Abstract

Variable-Length Graph Pattern Matching (VLGPM) is a critical functionality in graph databases, pivotal for identifying patterns where the number of connecting edges between two matched vertices is variable. This function plays a vital role in analyzing complex and dynamic networks such as social networks or bank transfers networks, where relationships can vary extensively in both length and structure. However, despite its importance, current graph databases, optimized primarily for single-hop subgraph matching, struggle with VLGPM over large graphs.

To bridge this gap between essential user requirements and the lack of efficient support in existing systems, we introduce VertexSurge. Central to VertexSurge is an innovative variable-length expand (VExpand) operator, which incorporates several microarchitecture-friendly optimizations to efficiently compute the reachability matrix between two sets of vertices. These optimizations enable VertexSurge to handle the surge of vertex count due to variable length with high performance. Additionally, VertexSurge combines VExpand with effective multi-set intersection for pattern matching, ruled-based planning, and disk offloading for large datasets, to implement a full-fledged VLGPM engine. Our evaluations with real-world graph datasets and representative patterns demonstrate that VertexSurge significantly outperforms existing systems in VLGPM, validating its efficacy in handling large-scale graph pattern matching challenges.

ACM Reference Format:
Weiyu Xie, Mingxing Zhang, Xia Liao, Kang Chen, Jinlei Jiang, and Yongwei Wu. 2024. VertexSurge: Variable Length Graph Pattern Match on Billion-edge Graphs. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3622781.3674173

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

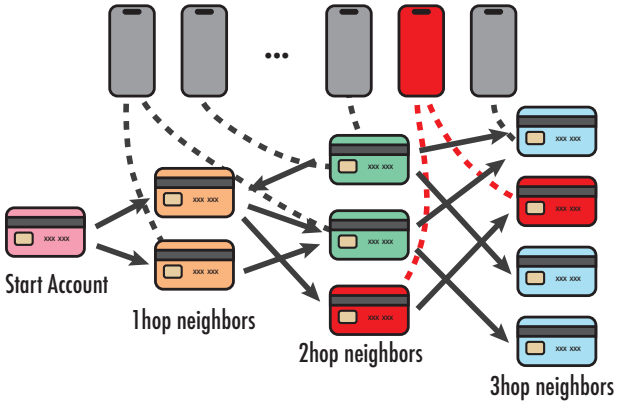
ACM ISBN 979-8-4007-0391-1/24/04.
https://doi.org/10.1145/3622781.3674173

1 Introduction

Efficient graph pattern matching lies at the very heart of modern graph databases and is used in a diverse range of domains [31, 33, 44, 49, 57, 59]. Although numerous optimizations [9, 17–19, 24, 39, 55] have been proposed to speed up the matching process, existing research predominantly focuses on single-hop pattern matching, where matched vertices are directly connected. In contrast, when searching for a variable-length graph pattern match (VLGPM), graph databases need to first enumerate all potential paths for each variable-length edge in the pattern before the pattern matching process. This enumeration phase may constitute the majority of execution time, particularly when selective filters are applied to vertices. In fact, due to the exponential increase in potential paths, matching VLGP with non-trivial lengths (e.g., $\geq$ 4-hop) on large graphs with billions of edges is often considered unfeasible. Our experiments on leading graph databases and graph mining systems (e.g. TigerGraph [23], Kuzu [28], Peregrine [32]) also confirm this problem.

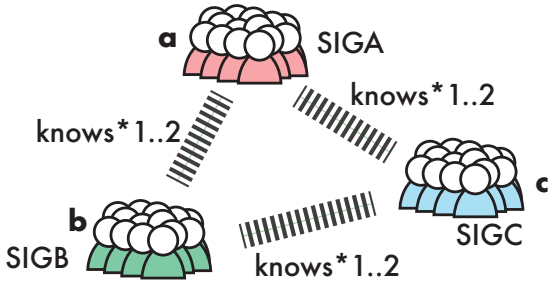
Nevertheless, VLGPM plays a pivotal role in various applications, including financial scenarios [25, 38, 60], fraud detection [30, 44], recommendation systems [51], and business intelligence [42]. For instance, the recently introduced LDBC FinBench (Financial Benchmark) [45] is dedicated to emulating the workloads found in the financial industry, modeling financial entities like bank accounts and transactions as vertices and edges in a graph. It incorporates many VLGPM queries within its Transaction Complex Read (TCR) workload, which plays a key role in identifying suspicious accounts. As demonstrated by Figure 1, TCR1, titled " Blocked medium related accounts ", targets accounts accessible with

Our project is opensource at https://github.com/madsys-dev/VertexSurge



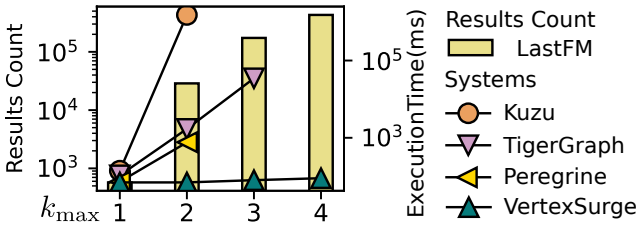

```
MATCH p=(start:Account{id:$id})
-[:transfer*1..3]->(neighbor:Account),
(neighbor)<-[:signIn]-(medium:Medium)
WHERE medium.isBlocked = true
RETURN neighbor,length(p);
```

Figure 1. This query finds accounts within 3 transfers ever signed in by blocked medium.




```
MATCH (a:Person:SIGA)-[knows*1..2]-
(b:Person:SIGB) MATCH (b)-[:knows*1..2]-
(c:Person:SIGC) MATCH (a)-[:knows*1..2]
-(c) RETURN COUNT(DISTINCT a,b,c);
```

(a) Community Triangle Query



(b) Results of community triangle on LastFM.

Figure 2. (a) This query counts triangles with variable-length paths. (b) The count and execution time of 4 systems.

3 forward transfers from specific starting accounts and authenticated by a blocked medium. A medium, defined as a device used for signing into accounts in the real world (e.g., phone numbers), is also modeled as a vertex in the graph. TCR1 is crucial for detecting and mitigating risks associated with bank accounts. Similarly, in Figure 2a, the data analyst aims to count the number of triangles in which each vertex belongs to a distinct community, and every pair of vertices is connected within 2-hop. This pattern exemplifies a typical social network recommendation application. VLGPM queries are also found useful in many other scenarios, such as bioinformatics [8], brain science [59], and industrial management [34].

To study the limitations of contemporary graph databases in executing VLGPM queries, we benchmarked the community triangle query using KUZU [28], TigerGraph [23], and Peregrine [32] on LastFM [37]. These systems stand as benchmarks for state-of-the-art open-source, commercial graph database solutions, and graph mining systems respectively. As depicted in Figure 2b, despite a substantial reduction in candidate vertices to approximately 2000 via the application of stringent filters, the number of community triangles grows exponentially as the maximum length $k_{\max}$ grows. Kuzu, TigerGraph, and Peregrine exhibit a rapid increase in execution time proportionate to the input size, with execution failing to complete when the number of community triangles is large.

In response to this challenge, we introduce VertexSurge, a VLGPM query system that ensures ten-second level VLGPM query latency on billion-edge graphs, even when the maximum searched length nears the graph's diameter. Figure 2b shows that VertexSurge completes community triangle queries on LastFM in just milliseconds, and the time taken does not increase exponentially with the increase volume of the final results.

At its core, VertexSurge implements an efficient variable-length expand (VExpand) operator. This operator accepts a set of candidate source vertices and constraints such as minimum and maximum length as inputs. The outcome of this operator finds the connection relationships between the input set and all vertices in the graph in a dense bit matrix format, which can be easily aggregated or converted to a common flat table. According to our study, major limitations of existing graph databases is the usage of the join operator and low microarchitecture-friendly design, which leads to a low hardware utilization, such as low instructions per cycle (IPC) and low memory bandwidth utilization. To overcome this problem, various optimizations are proposed to accelerate our novel *VExpand* operator, including *1)* both a breath-first search based and a sparse bit-matrix multiplication based computation kernels that are suited for different scenarios; *2)* a highly efficient microarchitecture-friendly system design, with a highly parallel and conflict-free implementation that take advantage of SIMD instructions; *3)* specific graph partitioning for bit matrix multiplication, traverse order, and the use of prefetch instructions that reduce the redundant and random memory accesses caused by multi-source searching. Bit-matrix multiplication ensures that increasing $k_{\max}$

will only proportionally increase the overall execution time of VExpand, which makes VertexSurge fearless of VLGPM queries with large $k_{\max}$. These together with other optimizations make VExpand achieve a maximal performance.

Furthermore, VertexSurge combines the above optimized VExpand operator with a specially designed multi-set intersect (*MIntersect*) operator to enable a full-fledged implementation of VLGPM query on complex property graphs, which is based on the worst-case optimal join (WCOJ) algorithm [41]. It helps us to achieve the least computation overhead and memory usage during the post-expanding pattern matching procedure. We also design a rule-based planner that aims to seek the best query plan for VertexSurge. VertexSurge is capable of running on disk when memory cannot hold all the intermediate results.

We have evaluated VertexSurge using various benchmarks and real-world graph datasets, comparing its performance with state-of-the-art graph database and graph mining systems. The evaluation results indicate that VertexSurge significantly surpasses existing solutions, improving performance by a factor ranging from 3× to 1215×. Our thorough evaluation underscores the effectiveness of the proposed optimizations and offers insights for maximizing hardware utilization during VLGPM query processing.

2 Preliminaries

2.1 Problem Definition

In this paper, we primarily discuss problems related to property graphs, a data model that is extensively employed in contemporary graph databases. Specifically, we give the definition of property graph as follows:

Definition 1. ***Property Graph:*** *A property graph G is defined as a tuple $G = (V, E, \lambda)$, where*

- *V is a finite set of vertex identifiers.*
- *E is a finite set of edges. Each edge is a 3-tuple, (eid, s, d), where eid is the edge identifier, and s, d are the vertex identifiers for the source and destination of the edge, respectively.*
- *$\lambda : (V \cup E) \times String \rightarrow Value$, attaches properties to its vertices and edges. Labels are stored in a special property that has a set of strings.*

An example of a labeled property graph, such as a social network, is illustrated in Figure 3. In this figure, dashed lines between vertex 1 and 4 represent paths consisting of more than one "knows" edge. Labels, prefixed with ':', identify different categories. In our example, there are four vertex labels, :Person, :SIGA, :SIGB, and :SIGC, with the latter three indicating which community a person belongs to.

When it comes to graph query, subgraph matching stands out as a pivotal task. For a graph G, a subgraph matching system identifies all subgraphs of G that are isomorphic to a pattern graph G_p. While this area of study has garnered

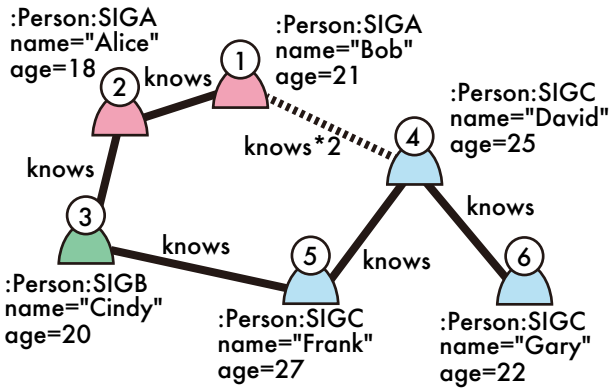


Figure 3. An Example Social Network

significant interest in the research community, it primarily centers on single-hop pattern matching [11, 32, 36, 55]. Variable-Length Graph Pattern Matching (VLGPM) poses unique challenges as a distinct type of matching problem. We will examine its definitions and complexities to deepen our understanding.

Definition 2. ***Variable-length path determiner*** *A variable-length path determiner can be denoted as $D = (k_{min}, k_{max}, dir, t)$, where k_{min} and k_{max} are minimum and maximum path lengths. $dir \in \{\rightarrow, \leftarrow, -\}$ restricts edges direction of selected paths. And $t \in \{ANY, SHORTEST\}$ indicates selected path type.*

Given a graph G, source vertex s and destination vertex d, a variable-length path determiner D determines whether vertex s and d are connected by a VLP that meets all requirements in definition 2. This determination can be denoted as $D(s, d) \rightarrow \{\text{True}, \text{False}\}$. When $t = \text{ANY}$, $D(s, d) = \text{True}$ if and only if d is accessible from s by $k_{\min}$ to $k_{\max}$ edges. When $t = \text{SHORTEST}$, $D(s, d) = \text{True}$ if and only if the shortest path connecting s and d satisfies conditions all above. For example, in the example social network shown in Figure 3, for $D_1 = (1, 2, -, \text{ANY})$, we have $D_1(1, 6) = \text{False}$ and $D_1(1, 2) = \text{True}$. For $D_2 = (2, 4, -, \text{SHORTEST})$, we have $D_2(1, 6) = \text{True}$ and $D_2(1, 2) = \text{False}$.

Definition 3. ***Variable-Length Graph Pattern And Matching*** *A variable-length graph pattern (VLGP) is defined as a tuple $P = (V_p, E_p, \sigma)$, where*

- *V_p is a set of vertex identifiers.*
- *E_p is a set of edges with variable-length path determiner, which are tuples like (s, d, D).*
- *$\lambda : (V_p \cup E_p) \times String \rightarrow Value$, attaches properties of vertices and edges of pattern graph.*
- *$\sigma : ((V \cup E) \times (V_p \cup E_p)) \rightarrow \{True, False\}$ is a property comparator, determining whether two vertices/edges are matched or not.*

Given a graph G and a VLGP P, variable-length graph pattern matching of P on G is to find all matched vertices sets.

A vertices set is a match, if and only if there exists a bijection $\theta : V_p \rightarrow V$ that for every vertex v_p, $\sigma(v_p, \theta(v_p)) = \text{True}$, and for every edge in E_p, $D(\theta(s), \theta(d)) = \text{True}$.

A VLGPM query can be processed in two steps, *1)* VLP search, and *2)* join. The key difference between VLGPM and subgraph matching is the first step. Subgraph matching systems typically require isomorphism, matching vertices and edges precisely. However, VLGPM uses a VLP determiner to determine vertex matches, focusing on vertices and not exact paths.

An example VLGP, community triangle, is shown in Figure 2a. Community triangle finds person triangles belong to three different communities with direct or indirect friendships. The pattern can be denoted as

$$P = \left(\{a, b, c\}, \left\{ \begin{array}{l} (a, b, (1, 2, -, \text{ANY})), \\ (b, c, (1, 2, -, \text{ANY})), \\ (a, c, (1, 2, -, \text{ANY})) \end{array} \right\}, \left\{ \begin{array}{l} \text{"SIGA"} \in a.Labels, \\ \text{"SIGB"} \in b.Labels, \\ \text{"SIGC"} \in c.Labels \end{array} \right\} \right)$$

For a vertex tuple $(1, 3, 4)$, vertex 1 belongs to the community "SIGA", vertex 3 belongs to "SIGB", vertex 4 belongs to "SIGC". Vertex 1 and 3 can be connected by 2 knows edges $1 - 2$, $2 - 3$, so the VLP determiner $D_1 = (1, 2, -, \text{ANY})$ results in $D_1(1, 3) = \text{True}$. Similarly, $D_2 = (1, 2, -\text{ANY})$ and $D_3 = (1, 2, -, \text{ANY})$ have results $D_2(1, 4)$ and $D_3(3, 4)$ both equal to True. So tuple $(1, 3, 4)$ is one of the matched results. 2 tuples are found on the example graph, they are $(1, 3, 4)$, and $(2, 3, 5)$.

The applicability of VLGPM is well recognized by the industry, with its applications spanning various sectors. For instance, TigerGraph utilizes VLGPM queries for anti-money laundering initiatives [6, 23] and has issued several technical reports [20, 61] highlighting the utility of deep link analysis in machine learning and customer analysis through VLGPM queries. Additionally, Nebula Graph leverages VLGPM queries in knowledge graphs for fraud detection and interactive mini-games [4, 48, 58]. Other entities like Create Link [2] and Ultipa [54] have also documented case studies demonstrating the versatile application of VLGPM in addressing complex analytical challenges.

2.2 VLGPM in Cypher

OpenCypher [29] is a widely used declarative language for graph querying. It is capable of formulating a variety of VLGPM queries through patterns of variable length, as demonstrated in the example queries depicted in Figure 2. In these queries, an edge marked with a Kleene star and dots represents a variable-length pattern. The numbers adjacent to these dots, denoted as k_{min} and k_{max}, specify the minimum and maximum lengths of the pattern to match. For instance, the pattern $(start) - [: transfer * 1..3] - > (neighbor)$ in Figure 1 identifies all neighbor accounts accessible from a starting account within 3 transfers. Similarly, a query for a variable community triangle, illustrated in Figure 2a, begins by identifying pairs of individuals connected by the pattern $(a) - [: knows * 1..2] - (b)$. Since VLGPM queries focus on vertices, these queries return results with DISTINCT. VertexSurge also only supports VLGPM queries that returns distinct tuples.

However, it's important to note a significant difference in the community pattern matching behavior shown in Figure 2a compared to the following Cypher query:

```
MATCH (a:Person:SIGA)-[:knows*1..2]-
(b:Person:SIGB)-[:knows*1..2]-
(c:Person:SIGC), (a)-[:knows*1..2]-(c)
RETURN COUNT(DISTINCT a,b,c);
```

A critical difference to observe here is that the above Cypher query utilizes only one MATCH pattern. According to OpenCypher's specifications, the matching of a single MATCH pattern must adhere to the "relationship uniqueness" rule (specified on page 11 in [29]). This rule ensures that the results from the same MATCH pattern do not include instances where the same edge is traversed multiple times. As an illustration, executing the above Cypher query on the example social network in Figure 3 would result in excluding the triplet $(2, 3, 5)$ as a valid match. This is because vertices 2 and 5 are linked through edges $2 - 3$ and $3 - 5$, which edges are already used in connecting vertices 2, 3 and vertices 3, 5, thereby violating the "relationship uniqueness" rule. Essentially, a single MATCH pattern in Cypher follows the "trail" semantics, whereas the VLGPM query defined in our paper employs "walk" semantics, which permits duplication [21]. The adoption of "walk" semantics is crucial as it enables the pattern matching system to bypass expensive path tracking, thus enhancing scalability.

2.3 Gaps in Supporting VLGPM

2.3.1 Graph Database. In the field of graph databases, it has been observed that both academic [28] and industrial [23] state-of-the-art systems struggle with the efficient handling of VLGPM queries within a practical period of time, especially those with a large k_{max}. We adjusted the example community triangle pattern by varying k_{max} using VLP determiners, then tested it on LastFM. As shown in Figure 2b, VertexSurge consistently outperformed other systems across all k_{max} values.

The limited performance of current graph databases on VLGPM queries primarily stems from the superfluous intermediate results generated by the join operation. In databases, the join operation links records from two tables based on a specific condition, so it is used to match paths and patterns.

However, using the join operation to implement VLP determiners could lead to an exponential growth in the number of paths between vertices. Most of these paths are redundant, as the VLP determiner only needs to check the existence of one path to produce a result. As Figure 2b shows, Kuzu spent a considerable time processing the community triangle ($k_{\max} = 2$) query. A detailed analysis of the profiling results reveals that 17% of the execution time was consumed by hash joins, and 81% of the time was dedicated to finding 1.3 million distinct vertex tuples out of 4.8 billion intermediate results. This implies more than 99.9% of intermediate results generated by Kuzu are superfluous.

This difference in performance, which we will explore in more detail in §4, is mainly due to the use of flat data formats, random memory access patterns, and implementations that are not optimized for the microarchitecture of CPU. Note that VertexSurge is a read-only query system, whereas graph databases are complex systems that also support orthogonal features, such as updates, transactions, and distributed execution. These optimizations can be employed by VertexSurge to improve the performance in handling VLGPM queries. For example, the factorized data structures used by Kuzu can minimize the cost of property scans of VertexSurge.

2.3.2 Graph Mining Systems. Graph pattern mining systems [17–19, 24, 32, 55] introduce innovative methods to reduce intermediate results and accelerate computation for graph mining problems, such as subgraph matching, frequent subgraph mining, etc. Although they may appear similar, VLGPM is not the problem that GPM systems aim to solve, nor can GPM systems be easily utilized or extended to address VLGPM queries.

To process a VLGPM query with GPM systems, we need to convert a single VLGPM query to **multiple** subgraph matching queries by converting each VLP into multiple fixed-length paths. After the GPM systems find subgraphs as the results, we convert the subgraphs back into VLGPM query results by extracting vertices in the VLGPM query. The results extracted from subgraphs contain many duplicate vertex tuples, necessitating deduplication to obtain the final results. For instance, the variable community triangle, as described in Figure 2a, would be converted into eight subgraph matching problems due to the different possible edge counts for each VLP. This results in $2^3 = 8$ subgraph matching scenarios where 2 is the possibility of edge number in each VLP and 3 is the number of VLPs. The final step requires aggregating the results to identify unique tuples of three individuals from different communities, which, like in graph databases, produces unnecessary intermediate results. In this way, Peregrine [32] achieves similar performance to TigerGraph when $k_{\max} \leq 2$, and times out when $k_{\max}$ is larger, as Figure2b shows.

Even though GPM systems are not designed to solve VLGPM queries, the optimizations they propose can still be applied to optimize VLGPM queries. A common optimization in GPM systems is leveraging pattern symmetry to reduce the matching of identical patterns [32, 55], which can be used to optimize the VLP search phase of a VLGPM query. For example, in an undirected graph of a social network, if Alice is accessible from Bob within 2 knows edges, then Bob is also accessible from Alice within 2 knows edges. Moreover, the approach used by DecoMine [17], which integrates pattern decomposition and user-defined functions with a compiler to generate an execution plan, can also be applied to optimize VLGPM counting queries.

3 Overview

In §2.3, we highlighted our observations about the existing graph databases. Existing graph databases generate superfluous intermediate results during VLGPM query execution. This poses two significant challenges: reducing the generation of unnecessary intermediate results in VLGPM queries and improving the overall performance of VLGPM query processing through better hardware utilization.

To address our first challenge, we introduce two new operators: VExpand and MIntersect. VExpand effectively tackles the issue of generating excessive intermediate results by utilizing an expand operation instead of a join operation. This operator processes multiple starting vertices simultaneously and generates a connecting matrix for these vertices. In this matrix, different paths connecting the same source and destination vertices are counted only once, thereby significantly reducing the duplication of paths in the intermediate results.

MIntersect, on the other hand, utilizes the Generic Join algorithm [41], which is a type of WCOJ (Worst Case Optimal Join) algorithm. These algorithms are known for guaranteeing optimal execution time in the worst case, relative to the size of the output. MIntersect works by retrieving the edge lists connected to a newly added vertex and performing intersections. The intersection results are then appended to the existing tuples to form newly matched patterns.

The key aspect of VLGPM that differentiates it from single-hop subgraph matching is the VLP search phase. The expand operation is more data-intensive than computation-intensive, as each edge adds at most one destination vertex to the intermediate results of a starting vertex. To maximize the performance of VExpand, we have implemented several optimizations at the microarchitectural level, including SIMD enhancements, memory layout adjustments, and synchronization improvements.

Additionally, our VLGPM query system incorporates a rule-based planner and disk-based storage, which helps manage both large outputs and intermediate results. More details about these aspects are discussed in §5.

4 VExpand

With the definitions provided in §2, this section explores the detailed optimizations employed in the implementation

of our core VExpand operator. The VExpand operator, applied to each starting vertex s in the input set S and coupled with a VLP determiner D, aims to identify all destination vertices d in the graph satisfying $D(s, d)$ = True. This task can be achieved through $k_{\max}$ iterative expand operations, beginning from S. If D specifies a path type of SHORTEST, VExpand also maintains visited vertices set V. After each expand operation, VExpand excludes visited vertices from intermediate results to ensure correctness. After $k_{\max}$ iterative expand operations, only results from $k_{\min}$ to $k_{\max}$ are merged as final results to keep $k_{\min}$ constraints.

4.1 A Bit Matrix Multiplication based Straw-man Solution

A straightforward method to implement the expand operation is by using a join operator that performs $k_{\max}$ rounds of breadth-first search. In the cth round, this operator joins each vertex in the current intermediate result set M_c with its outgoing edges, thereby adding destination vertices to M_{c+1}. Typically, these intermediate results are stored as arrays or hash sets of flat tuples, using uncompressed 64-bit integers to represent neighbor vertices. However, in VLGPM queries, the number of multi-hop neighbors for a vertex can increase exponentially in the initial stages of the expand operation. This surge in numbers makes the traditional flat representation of intermediate results inefficient in terms of memory use. An effective alternative is to employ compressed formats such as bitmaps for storing intermediate results. Theoretically, when the size of these results surpasses $|V|/64$, using bitmaps becomes more space-efficient.

Given that the input set S includes multiple starting vertices, the intermediate results M_c can be represented as a bit matrix with $|S|$ rows and $|V|$ columns. In this matrix, $M_c[i][j]$ represents a bit (0 or 1) indicating the reachability from the ith starting vertex in S to the jth vertex in V during the cth round. This compressed format effectively turns the expand operation into a bit matrix multiplication, multiplying a dense matrix (depicting reachability) with a sparse matrix (representing the edge list), formulated as $M_{c+1} = M_c \times E$. This method not only optimizes memory consumption but also significantly improves the efficiency of the expand operation in processing VLGPM queries. Also, both visited vertices and intermediate results are stored in bit matrices, element-wise bit operations can be efficiently used to maintain the set of visited vertices and get final results.

As shown in Figure 4a, to implement this bit matrix multiplication, we can first partition the result matrix along the column dimension to avoid concurrent updates. Subsequently, within each partition, a thread iterates over the ith starting vertices, ranging from 1 to $|S|$. For every k where $M_c[i][k] = 1$, it then further iterates over all outgoing edges of the kth vertex, which can be stored in a CSR (Compressed Sparse Row) format. The final step involves executing a *set_bit* operation for each destination vertex j from vertex k,

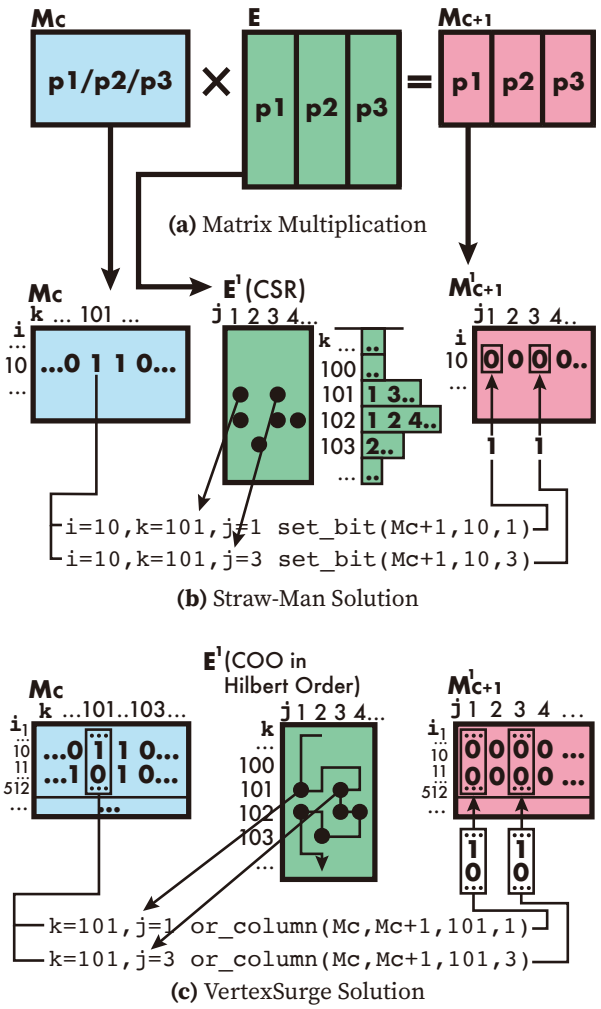


Figure 4. (a) The expand operations can be regarded as a bit matrix multiplication. Partition is used for multi-thread concurrency control. (b) Straw-Man solution implements expand operation in a traditional breadth-first search way, using set_bit. (c) VExpand of VertexSurge utilizes or_column to complete expand operations.

thereby setting the bit $M_{c+1}[i][j]$ to 1. Finally, for each destination vertex j from vertex k, the thread executes a *set_bit* operation to set the bit $M_{c+1}[i][j]$ to true. For example, as illustrated in Figure 4b, for the 101st column of $M_c[10]$, the operator iterates over the edge list $E[101]$, then calls *set_bit* twice to set $M_{c+1}[10][1]$ and $M_{c+1}[10][3]$ to true.

To optimize the cache utilization of this method, widely recognized techniques such as 2D partitioning [15, 62] can be used. However, despite their effectiveness in standard floating-point matrix multiplication, their direct application in a bit matrix context has resulted in suboptimal outcomes. The implementation used less than 10% of the memory bandwidth while fully occupying the CPU, which is an unusual

result for a typically memory-bound operation. This indicates potential inefficiencies in our bit matrix multiplication method, suggesting the need for further investigation and optimization.

4.2 Microarchitecture-friendly Optimizations

Our research found that the main inefficiencies in the approach are not because of its computational demand, but the code's inability to fully utilize the CPU's microarchitecture. A key inefficiency was the high number of instructions required to expand a single vertex along an edge. In the straw-man implementation, each update to a single bit of M_{c+1} necessitated a call to the *set_bit* function. This function involved heavy division and modulo operations to pinpoint the exact memory address to modify the target bit. Moreover, this procedure required loading the memory address, executing an OR operation, and then storing the updated value. These steps collectively resulted in a bottleneck in CPU instruction processing, considerably reducing the efficiency of the operation.

Another challenge encountered was the amplification of memory access due to the random distribution of target vertices for outgoing edges from a single source vertex. This randomness often led to adjacent destination vertices not being located on the same cache line. Consequently, altering a single bit typically involved reading and writing an entire cache line, leading to memory amplification of up to 512 times. These inefficiencies are underscored by performance metrics, such as a low Instruction Per Cycle (IPC) rate of 0.02, as reported by the Performance Monitoring Unit (PMU).

To address the challenges, we make the best use of the efficient SIMD instructions to both reduce the count of necessary instructions and minimize write amplification. By using AVX-512 instructions such as *VPORD*, we can simultaneously compute the bitwise logical OR of packed int32 elements. This method can effectively supplant up to 512 instances of *set_bit* when applied properly. However, the effective utilization of such instructions necessitates a corresponding organization of the data layout.

As illustrated in Figure 4c, rather than employing the conventional 2D partitioning approach, we have developed a unique data organization format for the dense intermediate result bit matrices M_c and M_{c+1}. In this format, M is partitioned every 512 rows into stacks, with the bits within each stack organized in a columnar major format. This arrangement is termed the stacked columnar major format.

In this stacked columnar major format, the 512 bits from $M[1][i], M[1][i], \ldots, M[512][i]$ are packed and stored consecutively, which is followed by the next 512 bits from $M[1][i+1], M[1][i+1], \ldots, M[512][i+1]$. Employing this format, we developed the *or_column(M_c, M_{c+1}, i, k, j)* function, which essentially executes "$M_{c+1}[i : i+512][j] = M_{c+1}[i : i+512][j]$ bitwise or $M_c[i : i+512][k]$" with a single *VPORD* instruction. This approach not only significantly enhances the operation's efficiency by reducing the number of instructions but also mitigates write amplification, as all destination bits are stored contiguously within the same cache line. This design is based on the observation that M_c typically has a high occupancy rate, with a large proportion of its bits set to 1. Hence, the additional overhead of processing unnecessary 0 bits in M_c is outweighed by the benefits of employing SIMD instructions. Other bit-matrix operations also benefit a lot from SIMD instructions. Element-wise bit operation of bit matrix can be accelerated, e.g. updating visited vertices by intermediate results can be achieved by *VPXORD*. Element counting of the bit matrix is also blazingly fast because of the use of *VPOPCNTQ*.

The efficiency of SIMD instructions emphasizes cache locality as a key factor that greatly impacts overall performance. To optimize this aspect, we arrange the sparse matrix of the edge list in a COO (Coordinate List) format, following the order of the Hilbert space-filling curve [40]. This optimization offers two significant benefits: *1)* It enables intuitive and effective prefetching, reducing the need to read unnecessary data blocks. In our implementation, if a thread is currently processing the xth edge and the $(x+20)$th edge is (s, d), we proactively prefetch the sth and dth columns of the corresponding stack from M_c and M_{c+1}, respectively. This approach provides a more refined level of prefetching compared to standard block-level prefetching techniques. *2)* The approach is cache-oblivious [43], seamlessly fitting scenarios where M_c and M_{c+1} are too large to fit entirely in the memory. This feature enables our methodology to be effortlessly extended to out-of-core scenarios.

5 Support Full-fledged VLGPM

5.1 Mintersect and Other Operators

After the VExpand operator processes the data, a filter operator refines the VLP search results by removing vertex pairs that fail to meet the query's property/label constraints. Subsequently, the Mintersect operator joins and matches the remaining vertex pairs by the input pattern.

Mintersect utilized Generic Join [41], a version of the Worst Case Optimal Join (WCOJ) algorithm, to ensure that the size of intermediate results remains asymptotically optimal. Its implementation entails $|V_p| - 1$ rounds of intersections, where $|V_p|$ represents the number of vertices in the pattern graph. Starting from a single matched edge, each round includes one additional vertex. During these rounds, VertexSurge performs a series of efficient bitwise AND operations between the result bit matrix columns.

The process of Mintersect is depicted in Figure 5, which illustrates the handling of a community triangle query detailed in Figure 2a. VertexSurge selects an edge (e.g., *AB*) from the

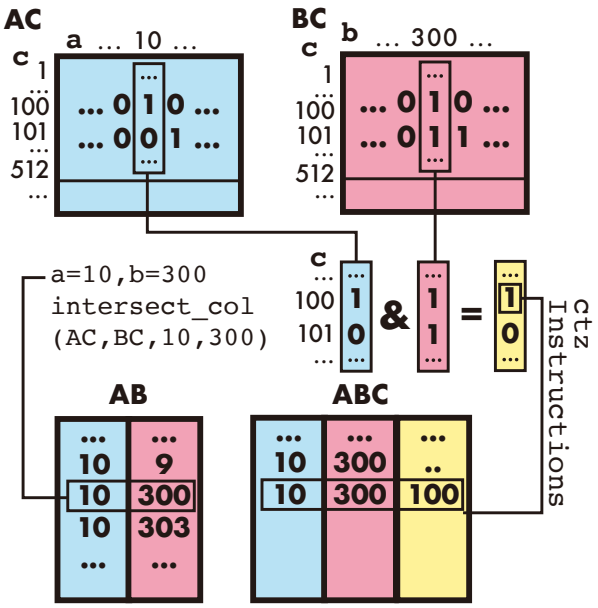


Figure 5. The Mintersect operator fetches the columns of bit matrix AC and BC, intersects them and generate results using ctz instructions.

pattern graph and iterates through its reachability bit matrix generated by VExpand. For each vertex pair in AB, like $(10, 300)$, Mintersect invokes $intersec\_col(AC, BC, 10, 300)$ to intersect the 10th column of AC with the 300th column of BC via bitwise AND operations. Each matched bit in the result equates to a matched tuple in the final results, as exemplified by the tuple $(10, 300, 100)$.

The columnar major bit matrix used by VertexSurge is crucial for Mintersect's efficiency. It facilitates intersections using SIMD instructions, surpassing the performance of traditional uncompressed data formats. VertexSurge also uses the ctz (count trailing zeros) instruction for rapid navigation through the result bitmap post-intersection. Additionally, when only the count of matched tuples is needed, Mintersect is optimized to leverage SIMD instructions for rapid and efficient counting. This method is further applied to other aggregation-related operators in VertexSurge, ensuring a comprehensive processing capability for VLGPM queries.

5.2 Planner

Similar to traditional subgraph matching, different physical execution plans for the same VLGPM query can also lead to significant variations in performance. To tackle this, we developed a specialized planner, which is built on a core principle: minimizing the size of intermediate results. Initially, it scans vertices based on filters to ascertain the size of vertex sets. The planner then estimates the size of VLP pairs by vertex count, $k_{\max}$, and average degrees. In selecting the first vertex for pattern matching, the planner opts for the vertex end of VLP pairs with the smallest estimated size. It then selects the next vertex, which has the smallest total estimated size of VLP pairs connecting to the existing matched patterns. The planner also makes fast online decisions during the execution of the VExpand operator by beginning the expansion from the smaller side.

5.3 Disk Based Design

Because VLGPM often involves scanning large graph data, VertexSurge stores them in a columnar format. Properties of vertices and edges are stored separately. Sources and destination of edges are stored separately based on their labels, in Coordinate List (COO) format. A metadata manager records which files contain edges of different labels. During VLGPM queries, the optimizer refers to this metadata to determine which files need to be scanned to retrieve the edges. If the query imposes constraints on the properties of edges, the system applies a filter operator after scanning to filter the results accordingly. VLGPM queries often result in the generation of intermediate and final results that not only surpass the size of the original graph but also exceed the constraints of physical memory. To manage this vast amount of data, VertexSurge utilizes mmap for efficient disk mapping of all graph data, intermediate results, and final results. This technique, combined with the cache-oblivious data organization optimizations of VExpand, ensures effective caching and fine-grained prefetching. We also incorporate partitioning techniques to facilitate concurrent writes. By allocating a dedicated file to each thread, the system effectively eliminates conflicts and hence maximizes disk bandwidth usage.

6 Evaluation

6.1 Evaluation Setup

Implementation and Platform: We implemented VertexSurge from scratch using C++. All experiments were conducted on a server equipped with 2 Intel(R) Xeon(R) Platinum 8452Y CPUs and 2TB of DDR5 memory. The server has a total of 72 cores and 144 threads. The experimental environment was Ubuntu 22.04. The test code was compiled using g++ 11.3.0 under the stdc++20 standard, with O3 optimization enabled.

Datasets: The basic information of graphs we used in the evaluation is shown in Table 1. We utilized two types of graphs, social network, and bank transfer graphs, for our evaluation. LastFM [47], Epinions [37], LiveJournal [37], and Twitter2010 [13, 14], are real social networks. For these real-world social network, we generate random vertex properties such as name and community. LDBC-SN are generated social networks by LDBC social network benchmark [10]. The scale factor indicates the rough total size of the generated graph in gigabytes. Rabobank [50] is a real-world bank transfer graph, consisting of bank accounts as vertices and transfers as edges. It is the only public bank transfer graph that we

can find. LDBC-FinBench is a generated financial graph by LDBC Financial Benchmark [45]. In our evaluation, we also assigned random risk tags to some specified accounts.

Dataset	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	Size
LastFM	7.6k	27.8k	3.66	9.3M
Epinions	75k	509k	6.79	66M
LDBC-SN-SF100	480k	23m	47.9	3.5G
Rabobank	1.62m	4.13m	2.55	204M
LDBC-SN-SF1000	3.2m	202m	63.13	28G
LiveJournal	4.8m	68m	14.17	14G
LDBC-FinBench-SF10	5.1m	22m	4.31	1.9G
Twitter2010	41m	1.47b	35.9	152G

Table 1. Information of graph dataset in evaluation, Rabobank and LDBC-FinBench-SF10 are bank transfer graphs, others are social networks. The column Size represents the total size of the graph, including vertices, edges, labels, and properties.

6.2 Case Studies

VLGPM queries have gained widespread application across various domains [8, 57, 59]. This section focuses on two critical application areas, social networks, and bank transfer graphs, to showcase the effectiveness of VertexSurge in processing VLGPM queries. We begin by describing the specific queries used in our evaluation and then comparing the performance of VertexSurge with two leading graph databases, TigerGraph and Kuzu, that represent the state-of-the-art in industry and academia, respectively. Systems are allowed to utilize all 144 threads. A warm-up query is executed before the performance test to load data from disk to memory. One query is executed 3 times, and the average execution time is reported.

6.2.1 Social Network. Social networks represent relationships among persons. In these networks, vertices symbolize *Person* and edges denote relationships like *knows* or *follows*. Each vertex may carry one or more labels, such as "SIGA," indicating community affiliation.

Queries: VLGPM queries are instrumental in revealing intricate patterns hidden within these complex networks. We have selected five representative queries to highlight various facets of VLGPM in real-world scenarios:

```
-- case 1
MATCH (p:SIGA)-[:knows*..3]-(q:SIGA)
RETURN COUNT(DISTINCT p,q);
-- case 2
MATCH (p:SIGA)-[:knows*..3]-(q:Person)
WHERE NOT q:SIGA RETURN COUNT(DISTINCT p)
as c,q ORDER BY c DESC LIMIT 100;
-- case 3
MATCH (p:SIGA)-[:knows*..3]-(q:SIGA)
RETURN COUNT(DISTINCT p) as c,q
ORDER BY c ASC LIMIT 100;
-- case 4
MATCH (a:Person:SIGA)-[knows*1..2]-
(b:Person:SIGB) MATCH (b)-[:knows*1..2]-
(c:Person:SIGC) MATCH (a)-[:knows*1..2]-(c)
RETURN COUNT(DISTINCT a,b,c);
-- case 5
UNWIND $person_ids AS pid
MATCH (p:Person{id:pid})<-[:knows*2..3]-(q:
Person) RETURN pid,COUNT(DISTINCT q);
```

Case 1: Community Cohesion Analysis: This query aims to quantify the interconnectedness within a community by counting the number of person pairs within a 3-hop range. The resulting count offers insights into the closeness of relationships within the community.

Case 2: External Influence Identification: It identifies the top 100 individuals outside a specified community who have the most 3-hop connections with its members. This analysis can help in understanding external influences and connections.

Case 3: Internal Community Dynamics: Conversely, this query focuses on members within a community with fewer connections to other members. This information is valuable for community moderators to understand engagement levels and potentially recommend new members or advise on membership adjustments.

Case 4: Inter-Community Interaction: The community triangle query is designed to count VLP triangles between three communities. It serves as a metric to gauge interaction intensity among different communities, facilitating collaborative opportunities.

Case 5: Influence Assessment: This query calculates the number of 2-hop and 3-hop neighbors for each person. In platforms like Twitter, this metric, encompassing both direct and indirect followers, is crucial for assessing an individual's influence. This analysis assists in tasks such as targeted advertising and influence assessment. Two thousand random person IDs are generated as the input for evaluation.

Results: The overall performance, depicted in Figure 6, was evaluated with a ten-minute timeout. VertexSurge consistently outperformed both Kuzu and TigerGraph in terms of execution time across various social network graphs. On most graphs, except Twitter2010, VertexSurge completed queries within a sub-second duration. For the Twitter2010 graph, VertexSurge processed each query in about 10 seconds, while the baseline systems failed to complete within the allocated timeout threshold, showcasing VertexSurge's superior efficiency in managing extensive graphs with over a billion edges.

Excluding instances of timeouts, VertexSurge achieved up to a 933x speedup compared to the best-performing baseline in Case 5 of the LDBC SF100. The smallest speedup, around threefold, was observed in Case 4 of the LiveJournal

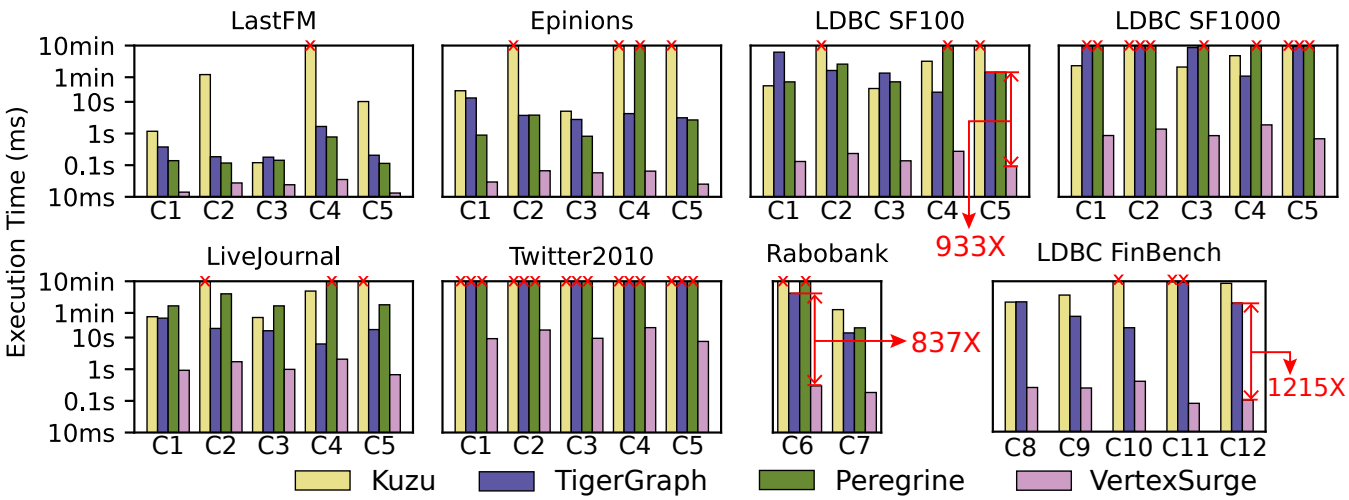


Figure 6. Execution time (Timeout=10min) of queries from Case 1 to 12, on different graphs, by different systems. Property constraints are removed for queries tested on Peregrine .

graph. According to our analysis, VertexSurge's performance improvement is more significant in graphs with higher average degrees, such as LDBC graphs and Twitter2010. This outcome aligns with our optimizations, which rely on the assumption that the reachability matrix will rapidly evolve into a dense matrix with the majority of bits set to 1. Furthermore, the time VertexSurge takes to complete queries is remarkably stable, showing minimal variation irrespective of the specific characteristics of the graph data. This consistency is primarily attributable to the use of a constant-size bit matrix in the VExpand module, ensuring VertexSurge's robustness against fluctuations in edge count and enhancing its suitability for diverse graph analysis scenarios.

6.2.2 Bank Transfer and Financial Graph.

Bank transfer graph is a directed graph consisting of *Accounts* and *transfer* activities. In addition to *Account* and *transfer*, the financial graph contains more vertices and edges, such as *Loan*, *Medium*, *Person*, *withdraw*, *signIn*, and *deposit*.

The application of VLGPM in this context is critical for financial institutions to analyze suspicious transactions, monitor accounts at high risk, and detects activities indicative of money laundering.

Queries: Similarly, for our evaluation, we selected two representative queries that exemplify the use of VLGPM in bank transfer graphs. For the financial graph, we select all the 5 VLGPM queries from the LDBC FinBench benchmark.

```
-- case 6
MATCH (a:Account:RISKA)-[:transfer*1..6]->
(b:Account:RISKA) WITH DISTINCT a,b
RETURN COUNT(*);
-- case7
MATCH (a:Account{id:rid})-[:transfer*1..3]
->(b: Account) RETURN DISTINCT b;
```

```
-- case8 (TCR1)
MATCH p=(start:Account{id:$id})
-[:transfer*1..3]->(neighbor:Account),
(neighbor)<-[:signIn]-(medium:Medium)
WHERE medium.isBlocked = true
RETURN neighbor,length(p);
-- case9 (TCR2)
MATCH (person:Person{id:$id})-[:own]->
(account:Account)<-[:transfer*1..3]-
(other:Account)<-[:deposit]-(loan:Loan)
RETURN other.id,SUM(DISTINCT loan.balance),
COUNT(DISTINCT loan);
-- case10 (TCR3)
MATCH (a:Account{id:$id1}),
(b:Account{id:$id2}),
p=shortestPath((a)-[:transfer*1..]->(b))
RETURN length(p);
-- case11 (TCR6)
MATCH (a:Account{id:$id})<-[:withdraw]-
(mid:Account)<-[:transfer]-(other:Account)
RETURN mid.id, other.id;
-- case12 (TCR8)
MATCH (loan:Loan{id:$id})-[:deposit]->
(src:Account)-[p:transfer|withdraw*1..3]->
(other:Account) RETURN DISTINCT
other.id, length(p);
```

Case 6: Cyclic Transaction Detection: This query aims to identify cyclic transactions within a 6-hop range among a predetermined set of risk-flagged accounts. The outcomes of this query are essential for banking monitoring systems to identify and flag accounts potentially involved in money laundering activities.

Case 7: Risk Account Connection Analysis: Post identification of risk accounts, the next step often involves tracing the flow of funds to other accounts. Case 7 is tailored to discover common transaction neighbors within these risk account groups, thereby aiding in the detection of new suspicious accounts.

Case 8: Blocked medium related accounts (TCR1): Given an account, this query finds all the accounts that are signed in by a blocked Medium and have funds transferred via transfer by at most 3 steps. Return the distance and id of these accounts. This query aims to find risk accounts related to one account.

Case 9: Fund gathered from the accounts applying loans (TCR2): Given a Person, find an Account owned by the Person that has funds transferred from other Accounts by at most 3 steps, which have funds deposited from a loan. Return the sum of distinct loan amounts, the sum of distinct loan balances, and the count of distinct loans.

Case 10: Shortest transfer path (TCR3): Given two accounts, find the length of the shortest path between these two accounts by the transfer relationships. The intent is to determine the minimum number of transactions required to connect two accounts, which can be instrumental in assessing the closeness of financial relationships or the efficiency of money flow within the network and identifying potential channels of money laundering.

Case 11: Withdrawal after Many-to-One transfer (TCR6): This query finds all the middle accounts that withdrew from the given account, and other accounts ever transferred to middle accounts. This query aims to find the middle accounts to detect abnormal money collection.

Case 12: Transfer trace after loan applied (TCR8): Give a Loan and trace the fund transfer or withdraw by at most 3 steps from the account the Loan deposits. Return all the accounts' IDs and their distance from the loan.

Results: For bank transfer, the execution time comparison for Cases 6 and 7 on the Rabobank dataset is illustrated in Subfigure Rabobank in Figure 6. In Case 6, VertexSurge achieved a staggering 837x speedup over TigerGraph. This remarkable efficiency gain can be attributed to the longer VLP length in Case 6. In Case 7, despite the similarities to Case 5 from the social network scenario, VertexSurge's speedup was less pronounced. This discrepancy arises because Case 7 involves identifying common neighbors, reducing the baseline systems' need to store intermediate results for every starting vertex. However, VertexSurge, still requires the storage of intermediate results and subsequent aggregation post-VExpand, managed to maintain a substantial lead, outperforming the baseline systems by 70×.

For the financial graph, as depicted in Subfigure LDBC FinBench of Figure 6, Cases 8 through 12 on the LDBC FinBench showcase execution time comparisons. Cases 8 and 9 exhibit nearly identical performance due to sharing the same VLGPM pattern, which involves 3-hop queries on the *transfer* edges. Surprisingly, VertexSurge achieves performance in Case 10, a shortest path query, that is approximately 50 times better than TigerGraph, which employs a bi-directional search. The query results indicate that over 95% of the shortest paths found in Case 10 have a length equal to or greater than 10. Even with bi-directional search, TigerGraph needs to expand at least 5 hops to obtain results, making it slower than VertexSurge. Case 11 involves only a 2-hop expansion query. However, neither TigerGraph nor Kuzu could complete all Case 11 queries within 10 minutes. This is because Case 11 queries go backward along edges, and neither Kuzu nor TigerGraph employs reverse edges. VertexSurge achieves the fastest speedup in Case 12, at an acceleration of 1215 times. This is due to the expand in Case 12 involving 2 types of edges, *transfer* and *withdraw*, over 4 steps. The number of transfer edges is only 6.2M, and together with withdraw, there are a total of 13M edges. The substantial overhead of the expand operation in Case 12 highlights the pronounced advantage of VertexSurge in this scenario.

6.2.3 Comparison with Peregrine. The current implementation of Peregrine [32] does not support properties and directed graphs. Therefore, we made some modifications to the query cases to enable Peregrine to execute them. Initially, we extracted subgraphs from the graph that included the relevant vertices and edges related to the queries. This approach avoids filtering labels with a large number of vertices, such as "Person" and "Account" labels. Also, Peregrine's implementation only supports attaching one integer as a label on a vertex and it only allows label constraints on one vertex of the input pattern. Thus, for each case, we attached a special label to vertices that are selected by property/label constraints with the lowest selectivity. For example, we attached a special label to persons with SIGA in Case 1. After Peregrine finishes its execution, we apply further filter based on other labels and property constraints. We used the C++ programming interface provided by Peregrine, and employed Intel TBB's concurrent hash set for deduplication. Peregrine currently does not support multiple edge labels and directed edges, so we skip evaluation on LDBC FinBench.

The execution times are shown in Figure 6. VertexSurge only achieves 4× speedup over Peregrine when executing Case2 on LastFM. On small graphs, superfluous intermediate results generated by Peregrine are not much yet. For a billion-edge graph like Twitter2010, Peregrine fails to complete any query cases. As expected, Peregrine exhibited its worst performance in Case 4, it failed to finish Case 4 except LastFM. Case 4 requires GPM systems to convert it into multiple subgraph matching queries, and then perform deduplication, as discussed in §2.3.2.

6.2.4 Fearless of large $k_{\max}$. As discussed in §2.3, while existing methods face large costs with increasing $k_{\max}$, VertexSurge is optimized for handling VLGPM queries even with higher $k_{\max}$ values. Our performance tests, varying $k_{\max}$ from 1 to 6, confirm this through a linear trend in execution time,

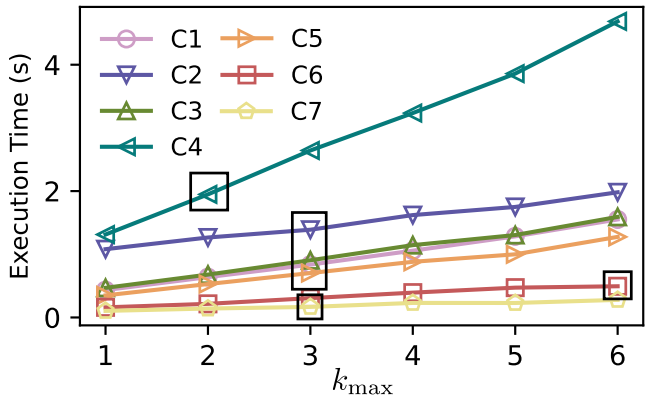


Figure 7. Execution time of all queries with different $k_{\max}$, on LDBC SF1000(C1-C5) and Rabobank(C6,C7). Data points in black boxes are test cases used in end-to-end overall tests.

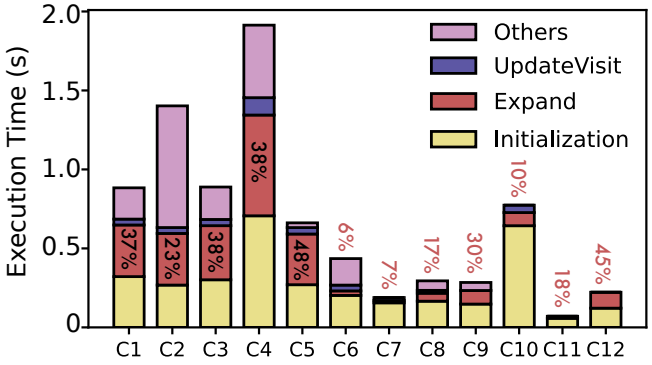


Figure 8. Execution time of different parts of a query.

as illustrated in Figure 7. To better understand system performance, we profiled VertexSurgeto get processing time of its components, so that we can check whether the optimized VExpand still remains a bottleneck.

As illustrated in Figure 8, during the execution of Cases 1-5, the proportion of time allocated to expand operations averages around 35%. Notably, VertexSurge spent less than 10% of its time on the VExpand operation when executing cases 6 and 7. This reduced time requirement can be attributed to Rabobank's smaller edge count compared to LDBC SF1000. C8 and C9 all contains 3hop transfers, but C9's "Expand" starts from more vertices, consuming more time. C10 is to find the length of the shortest paths, so a large number of steps of "Expand" need to be executed, leading to multiple matrices that need to be initialized. C11 and C12 are just simple expand query on different edges. Their path type is ANY, so they do not spend time on "UpdateVisit". These findings indicate that the speed of the VExpand operation is no longer the primary bottleneck influencing the performance of VLGPM query execution in VertexSurge.

6.3 Detailed Analysis of VExpand

This section discusses the breakdown of the optimizations implemented in the VExpand operator. The experiments outlined here utilized a single VExpand operator with 20,480 starting vertices on the LDBC SF 1000 dataset.

Optimized Memory Usage: VExpand achieves significant memory efficiency through two primary strategies. Firstly, it replaces the traditional join operation with an expand operation, substantially reducing superfluous intermediate results (§3). Secondly, VExpand employs a bit matrix based compressed representation (§4.1). Table 2 illustrates that when comparing join there is an 8-fold difference in the volume of intermediate results when $k_{\max}$ set to 3. This reduction translates to a remarkable 66-times decrease in memory usage when utilizing the bit matrix format, compared to the original flat data representation. The benefits of this approach are expected to be even more pronounced with larger $k_{\max}$ or more intricate pattern matching scenarios.

	$k_{\max}=1$	$k_{\max}=2$	$k_{\max}=3$
Join	2.5×10^6	1.1×10^9	3.0×10^{11}
Expand	2.5×10^6	7.0×10^8	3.6×10^{10}
Join/Expand	1	1.52	8.51

Table 2. Intermediate results count of two methods and their ratio, on LDBC SF 1000.

Achieving High Performance: Figure 9 presents the speedup of various versions of VExpand compared to the initial straw-man solution, with different optimizations discussed in §4.2. Specifically, *Column-Major* optimization uses or_column on a column-major matrix. *SIMD* optimization employs AVX512 instructions, such as VPORD, to execute VExpand and Mintersect. *Hilbert* optimization sorted the COO edge list according to the Hilbert space-filling curve order. *Prefetch* involves prefetching matrix columns that will be used by $(x+20)$th edge when processing xth edge. The co-use of SIMD and Hilbert optimizations led to a huge performance increase. Also, prefetch optimization significantly reduced L1 cache misses, dropping from 19.8% to a mere 1.3%. These collective optimizations have markedly improved the efficiency of VExpand, leading to a 20× speedup over the straw-man solution and utilizing nearly 80% of the available memory bandwidth.

7 Related Work

Graph Database: Graph databases have received significant attention in recent years, with open-source options such as Neo4j [5], JanusGraph [3], Nebula [4], and GraphScope [27], as well as commercial databases like TigerGraph [23], TuGraph [7], AtlasGraph [1], GalaxyBase [2], and Ultipa [54]. Among these, Ultipa employs techniques such as pruning

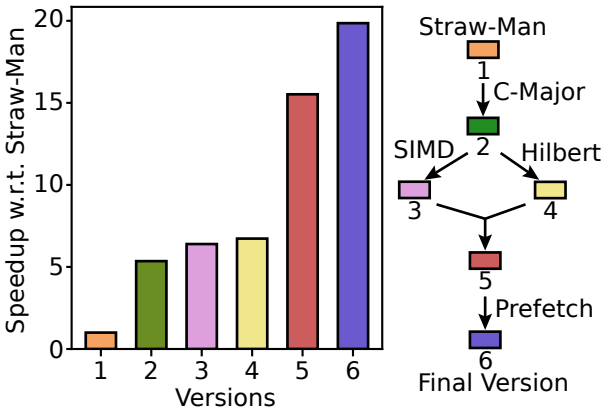


Figure 9. Execution time of different versions of VExpand. The diagram on the right indicates the optimizations added to each version.

to enhance its ability to find deep neighbors quickly. GraphScope has modified CSR into a writable Gart [52] to balance OLTP queries with OLAP queries.

In academia, initiatives like Kuzu [28], A1 [16], and GraphFlowDB [35] have explored graph database processing. These databases employ various methods to minimize intermediate results. Although they still generate superfluous intermediate results when processing VLGPM queries, their orthogonal techniques can be applied to optimize the execution of VLGPM query. Kuzu designed a factorized query processor to achieve greater space-efficiently when processing multi-join queries. It also collects the vertices whose properties need to be read, and then scans them all at once to avoid random disk reads. The optimized join proposed by Kuzu can be applied to MIntersect operator. A1, leveraging FaRM [26], has implemented graph storage on economical memory resources using RDMA. GraphFlowDB has designed an incremental generic join algorithm to manage streaming graph queries.

Significant efforts have also been dedicated to scheduling computational tasks in distributed graph queries to achieve higher performance. Systems like GAIA [46] and Banyan [53] use dataflow abstractions for graph processing across distributed clusters, while aDFS [56] collects cross-node DFS queries in buffers before dispatching them collectively to the next node, balancing the size of intermediate results with the utilization of parallelism. These orthogonal optimizations in graph query systems would be useful when dealing VLGPM queries in distributed scenarios.

Graph Mining Systems: In recent years, many studies have focused on graph pattern mining problems, such as subgraph matching and frequent subgraph mining. Arabesque [55]is the first distributed GPM system that provides higher level programming interface of based on MapReduce [22]. Fractal [24] is the first depth-first search GPM system and introduced "factoids" as the API. Tesseract [12] is the first GPM system supporting evolving graphs. Peregrine [32] reduces the need for canonicality checks and isomorphism computations by extracting semantics of input pattern to guide graph matching exploration. DecoMine [17] utilized a compiler-based approach to automatically decompose patterns to integrate user defined functions and generate execution plan based on a cost model. Moreover, there are systems that utilize GPUs to accelerate graph mining tasks. Pangolin [19] implements GPM on GPUs, employing a structure of arrays data structure to facilitate parallel processing on GPUs. G^2Miner [18] takes this further by addressing GPM issues on multiple GPUs.

Although GPM systems are not designed for VLGPM queries, there is still an opportunity to apply their orthogonal optimizations to improve VLGPM queries. By leveraging the symmetry of input patterns and performing canonicality checks [32, 55], the redundant paths in the VLP search can be skipped. For the second stage of VLGPM queries, the optimization of pattern match, such as guided pattern exploration [32], pattern decomposition [17], can be used to reduce intersection of MIntersect. The insights supporting updates for GPM queries and design of multi-version graph store [12] provide valuable references for the future update support of VLGPM queries.

8 Conclusion

This paper introduces VertexSurge, a system meticulously designed for the efficient execution of VLGPM queries. VLGPM is a crucial type of graph pattern matching widely utilized in a variety of real-world scenarios. The cornerstone of VertexSurge is its microarchitecture-friendly implementation of the VExpand operator. This operator effectively eliminates superfluous intermediate results and achieves remarkable hardware utilization. Evaluations have demonstrated that VertexSurge is able to answer complex VLGPM queries even in billion-scale graphs, up to three orders of magnitude faster than previous work.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Dr. Laurent Bindschaedler, for their valuable comments and helpful suggestions. The authors from Tsinghua University are all in the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China. This work is supported by National Key Research & Development Program of China (2022YFB2404200), Natural Science Foundation of China (61877035) and Tsinghua University Initiative Scientific Research Program, Young Elite Scientists Sponsorship Program by CAST (2022QNRC001), and Beijing HaiZhi XingTu Technology Co., Ltd. Correspondence to: Mingxing Zhang (zhang_mingxing@mail.tsinghua.edu.cn), Jinlei Jiang (jjlei @tsinghua.edu.cn), and Yongwei Wu (wuyw @tsinghua.edu.cn).

References

[1] Atlasgraph, https://atlasgraph.io/.

[2] Galaxybase, https://www.galaxybase.com/galaxyproduct.

[3] Janusgraph, https://janusgraph.org/.

[4] Nebula graph, https://www.nebula-graph.io/.

[5] Neo4j graph data platform, https://neo4j.com.

[6] Tiger graph aml. https://www.tigergraph.com/solutions/anti-money-laundering-aml/.

[7] Tugraph, https://www.tugraph.org/.

[8] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S Cenk Sahinalp. Biomolecular network motif counting and discovery by color coding. 24(13):i241–i249. 00000 Publisher: Oxford University Press.

[9] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. 11(6):691–704.

[10] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasić, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. The LDBC social network benchmark. 00012.

[11] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1199–1214. ACM. 00000.

[12] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. Tesseract: distributed, general graph pattern mining on evolving graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 458–473. ACM.

[13] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[14] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[15] Erik G Boman, Karen D Devine, and Sivasankaran Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning.

[16] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, Matthew Renzelmann, Alex Shamis, Timothy Tan, and Shuheng Zheng. A1: A distributed in-memory graph database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 329–344. ACM. 00000.

[17] Jingji Chen and Xuehai Qian. DecoMine: A compilation-based graph pattern mining system with pattern decomposition. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 47–61. ACM.

[18] Xuhao Chen. Efficient and scalable graph pattern mining on GPUs.

[19] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: an efficient and flexible graph mining system on CPU and GPU. 13(8):1190–1205.

[20] Martin Darling. Delivering deep-link analysis. https://m.digitalisationworld.com/blogs/56699/delivering-deep-link-analysis.

[21] Claire David, Nadime Francis, and Victor Marsault. Run-based semantics for RPQs. In *Proceedings of the Twentieth International Conference on Principles of Knowledge Representation and Reasoning*, pages 178–187. International Joint Conferences on Artificial Intelligence Organization.

[22] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. 51(1):107–113. 00000.

[23] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. TigerGraph: A native MPP graph database. 00015.

[24] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1357–1374. ACM.

[25] Buket Doğan. The importance of graph databases in detection of organized financial crimes.

[26] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414. USENIX Association.

[27] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Jingren Zhou, Diwen Zhu, and Rong Zhu. GraphScope: a unified engine for big graph processing. 14(12):2879–2892.

[28] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. KÙZU^* graph database management system.

[29] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445. ACM.

[30] Richard Henderson. Using graph databases to detect financial fraud. 2020(7):6–10. Publisher: MA Business London.

[31] Connor C J Hryhoruk and Carson K Leung. Compressing and mining social network data.

[32] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16. ACM.

[33] Fan Jiang, Carson K Leung, and Adam G M Pazdor. Big data mining of social networks for friend recommendation.

[34] Bernard Kamsu-Foguem and Daniel Noyes. Graph-based reasoning in collaborative knowledge management for industrial maintenance. 64(8):998–1013.

[35] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1695–1698. ACM.

[36] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *Proceedings of the 2021 International Conference on Management of Data*, pages 925–937. ACM.

[37] Jure Leskovec and Andrej Krevl. SNAP datasets: Stanford large network dataset collection. 00000.

[38] Nikolaos Livieratos and Vasilios T Tampakas. << GRAPH DATABASES AND THEIR APPLICATION TO FINANCIAL PROBLEMS >>.

[39] Daniel Mawhirter and Bo Wu. AutoMine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 509–523. ACM.

[40] B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. 13(1):124–141.

[41] Hung Q. Ngo, Christopher Re, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms.

[42] André Petermann, Martin Junghanns, Robert Müller, and Erhard Rahm. Graph-based data integration and business intelligence with BIIIG. 7(13):1577–1580.

[43] Harald Prokop. Cache-oblivious algorithms.

[44] Debachudamani Prusti, Daisy Das, and Santanu Kumar Rath. Credit card fraud detection technique by applying graph database model. 46(9):1–20.

[45] Shipeng Qi, Heng Lin, Zhihui Guo, Gábor Szárnyas, Bing Tong, Yan Zhou, Bin Yang, Jiansong Zhang, Zheng Wang, Youren Shen, Changyuan Wang, Parviz Peiravi, Henry Gabb, and Ben Steer. The LDBC financial benchmark.

[46] Zhengping Qian, Youyang Yao, Chenqiang Min, Bingqing Lyu, Longbin Lai, Xiaoli Zhou, Yong Fang, Zhimin Chen, Gaofeng Li, and Jingren Zhou. GAIA: A system for interactive analysis on distributed graphs using a high-level language.

[47] Benedek Rozemberczki and Rik Sarkar. Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*, page 1325–1334. ACM, 2020.

[48] Nick Russel. Fraud detection using knowledge graph: How to detect and visualize fraudulent activities. https://www.nebula-graph.io/posts/fraud-detection-using-knowledge-and-graph-database.

[49] Gorka Sadowksi and Philip Rathle. Fraud detection: Discovering connections with graph databases.

[50] Akrati Saxena, Yulong Pei, Jan Veldsink, Werner van Ipenburg, George Fletcher, and Mykola Pechenizkiy. The banking transactions dataset and its comparative analysis with scale-free networks. In *Proceedings of the 2021 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 283–296, 2021.

[51] Bilin Shao, Xiaojun Li, and Genqing Bian. A survey of research hotspots and frontier trends of recommendation systems from the perspective of knowledge graph. 165:113764.

[52] Sijie Shen, Zihang Yao, Lin Shi, Lei Wang, Longbin Lai, Qian Tao, Li Su, Rong Chen, Wenyuan Yu, Haibo Chen, Binyu Zang, and Jingren Zhou. Bridging the gap between relational OLTP and graph-based OLAP.

[53] Li Su, Xiaoming Qin, Zichao Zhang, Rui Yang, Le Xu, Indranil Gupta, Wenyuan Yu, Kai Zeng, and Jingren Zhou. Banyan: a scoped dataflow engine for graph query service. 15(10):2045–2057.

[54] Ricky Sun and Jamie Chen. Design of highly scalable graph database systems without exponential performance degradation. In *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments*, pages 1–6. ACM.

[55] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 425–440. ACM.

[56] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, and Hassan Chafi. aDFS: An almost depth-first-search distributed graph-querying system. page 17.

[57] Joerg Kurt Wegner, Aaron Sterling, Rajarshi Guha, Andreas Bender, Jean-Loup Faulon, Janna Hastings, Noel O'Boyle, John Overington, Herman Van Vlijmen, and Egon Willighagen. Cheminformatics.

[58] Wey. How i cracked chinese wordle using knowledge graph. https://www.nebula-graph.io/posts/crack-chinese-wordle-using-knowledge-graph.

[59] Qiong Wu, Xiaoqi Huang, Adam Culbreth, James Waltz, Elliot Hong, and Shuo Chen. Extracting brain disease-related connectome subgraphs by adaptive dense subgraph discovery.

[60] Natalia Yerashenia and Alexander Bolotov. Computational modelling for bankruptcy prediction: Semantic data analysis integrating graph database and financial ontology.

[61] Gaurav Deshpande Yu Xu. Machine learning and deep link graph analytics: A powerful combination. https://www.kdnuggets.com/2019/04/machine-learning-graph-analytics.html.

[62] Ümt V. Çatalyürek, Cevdet Aykanat, and Bora Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. 32(2):656–683.