An Efficient Memoization Engine for Concurrent Graph Query Processing (complete version)

Anonymous Author(s)

ABSTRACT

Concurrent graph query (CGQ) processing has been used to solve a wide range of graph applications. By analyzing real-world workloads of CGQs, we observe significant repeated computations among the queries. In this work, we present *KGraph*, a novel graph processing memoization engine to efficiently handle CGQs on large graphs by performing *memoization on graphs*. However, the efficacy of memoization in optimizing CGQs on large graphs is constrained by substantial computational and memory overheads, coupled with the potential amount of sharing opportunities. Thus, we develop two novel approaches in KGraph to address the memoization overhead. First, we develop a fine-grained memoization method, which only maintains query results within their associated graph partitions. This approach not only reduces the overhead but also enhances the potential for sharing. Secondly, we selectively perform memoization on pivotal queries, those with a high likelihood of promoting substantial computation sharing among CGQs, while avoiding the excessive overhead associated with managing unnecessary memoization across a large number of queries. We comprehensively analyze KGraph's performance using five popular CGQ applications. Experimental results show that our system achieves an average speedup of 4.2× over the state-of-the-art CGQ systems.

1 INTRODUCTION

Concurrent Graph Query (CGQ) processing is an emerging processing paradigm in graph analysis, particularly when applications necessitate the simultaneous processing of multiple graph queries. For example, a large number of random walks need to be generated to feed downstream applications such as DeepWalk [32], Personalized PageRank (PPR) [34], and Ant Colony Optimization (ACO) [12]. Separately, in a more practical context, services like taxi-hailing in Jakarta launches over 1.5K Single-Source Shortest Path (SSSP) queries per minute during peak hours, as shown in Figure 1.

Processing CGQs presents significant computational challenges, often becoming the bottleneck in graph analysis. For instance, a study by Akiba et al.[1] reported that executing a few batches of 1,024 concurrent SSSP queries, as part of processing Pruned Landmark Labelling, accounted for over 99% of the total execution time. Similarly, the execution of a large number of random walks represents a significant bottleneck in numerous graph neural network training applications, as evidenced in various studies [37]. This has spurred increasing interest in enhancing CGQ processing efficiency [27, 35, 40], with a primary focus on exploiting the *sharing* opportunities among CGQs. The state-of-the-art, ForkGraph [27], optimizes spatial sharing of cache. It does this by employing graph partitions that fit into the last-level cache (LLC), thus reducing cache misses. Graph operations are executed in a coordinated fashion, grouping memory accesses according to the partition currently in the LLC.

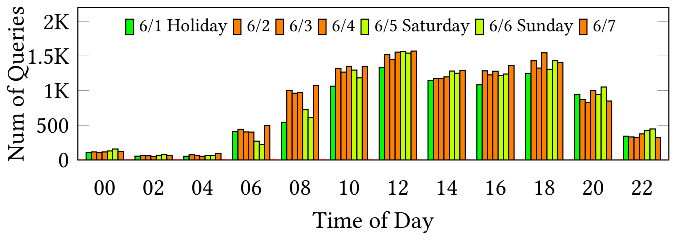


Figure 1: Hourly Distribution of Taxi Hailing in Jakarta During the First Week of June 2021 (Average Number per Minute).

While existing works exploit *resource sharing* opportunities such as cache, our experimental study reveals that a substantial portion of the computations during the processing of CGQs are repeated. We experimentally execute 100 SSSP queries from randomly selected source vertices on five scale-free graphs generated using R-MAT [6] and we have the following observations in terms of the potential sharing opportunities. First, given an edge e that belongs to the results of an arbitrary query, the chance of e belonging to the results of another query is around 90%. Second, more than 10% edges in the results of CGQs belong to the results of at least 90 queries. In our collaboration with a large superapp company ABC in this region, the taxi hailing service, as illustrated in Figure 2, demonstrates analogous findings. Here, we observed that numerous queries share identical pickup points, and an even greater number of queries have pickup points in close proximity to each other. Both observations render significant sharing opportunities for the computation results.

Based on our observations, we propose a novel memoization engine named KGraph to exploit the *computation sharing* among CGQs. Memoization is a widely-used technique to avoid repeated computation. The fundamental idea applied in this paper is to process some of the queries, memoize the results that can be used by other queries, and then process the rest. While processing a query, we check if the query can reach any memoized results and then reuse them if so. In such cases, the memoized queries are taken as the subproblems of the query to process. The application of memoization thus prunes repeated graph operations and reduces random memory accesses in the processing. For example, if a query q_a visits a vertex that is the source vertex of a query q_b that has been processed with its results memoized, the processing of q_a can potentially benefit from the results of q_b [4, 19].

However, we observe that memoization can sometimes result in significant overhead and lead to degraded performance on large graphs because the operation reduction ratio is low and the overhead of scanning the memoized results offsets the reduction benefit. Specifically, a hand-tuned program achieves 2.3× speedups by memoization when handling 1,000 SSSP queries on a small-scale graph with 10K edges, whereas the same program exhibits a 20% performance degradation on a larger graph with 100M edges compared to

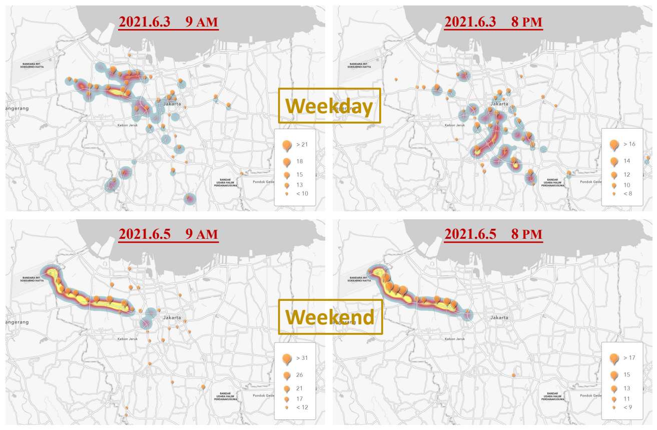


Figure 2: Top Intersections Visited by ABC Taxi in the First Ten Seconds During Morning and Evening Rush in Jakarta.

the processing without memorization. Achieving the same level of memoization benefits becomes more challenging for an equivalent number of queries when dealing with larger graphs.

We propose two novel approaches to improve the efficiency of KGraph. *First, we develop a fine-grained memoization on the partitioned graph and a cost model to determine the granularity of graph partitioning.* To improve the memoization effectiveness and achieve better locality within a partition, KGraph only processes and memoizes query results within the partition where they are located. Consequently, the size of the graph partition emerges as a crucial tuning parameter, prompting us to develop a cost model specifically designed to configure this parameter. *Second, we propose to select pivot queries to memoize since the results of these queries have a higher chance of being reused by CGQs, which thus helps to reduce more graph operations.* Instead of directly finding the sharing opportunities among CGQs, we leverage the sharing opportunities between CGQs and memoized pivots with higher sharing opportunities. We have evaluated multiple pivot selection strategies to improve the chance of sharing. We further devise a decision tree-based model that leverages graph features to find the best configuration for pivot selection.

KGraph is designed to ease the implementation of memoization for CGQs by providing a few high-level application programming interfaces (APIs). Users only need to program the sequential algorithm and KGraph automatically parallelizes it with memoization enabled. We use KGraph to implement two types of graph algorithms: (1) The traversal-based algorithms, which include SSSP, SSWP (single-source widest path), and SSR (single-source reachability; (2) The random walk-based algorithms, such as DeepWalk and PPR (Personalized PageRank).

We experimentally compare KGraph[1] with the state-of-the-art concurrent graph processing framework, ForkGraph [27]. We evaluate KGraph using the five applications mentioned above as well as a case study with ABC real workloads. The experimental results show that KGraph achieves up to 12.5× speedup with an average speedup at 4.1× over ForkGraph.

[1]The source code is available at *https://anonymous.4open.science/r/KGraph.*

The remainder of this paper is organized as follows. Section 2 introduces the background of CGQ and memoization. In Section 3, we present the motivations, followed by an overview of KGraph in Section 4. We introduce the memoization on graphs in Section 5. We present the experimental results, and a case study with ABC real workloads in Section 6. Finally, we review the related work in Section 7 and conclude in Section 8.

2 BACKGROUND

2.1 Preliminaries

We define a graph $G = (V, E)$ to be a directed graph, where V and E are the sets of vertices and edges, respectively. An undirected graph can be represented as a directed graph by replacing each undirected edge with two edges from both directions. The vertex partitions of a graph is denoted by P and the number of partitions is denoted as $|P|$.

Concurrent graph queries (CGQs). CGQs are essential in many graph applications such as BC, PLL, ACO, single-source k-shortest path [46], and machine learning on graph using random walks [16, 32]. These applications launch finite independent queries from different source vertices on the same graph [1, 12, 16, 18, 32, 46]. We use set $Q = \{q_1, q_2, ..., q_{|Q|}\}$ to denote CGQs, a set of homogeneous graph queries that are *simultaneously* launched from $|Q|$ source vertices on the graph G. Besides, we also define a set of pivot queries $H = \{h_1, h_2, ..., h_k\}$ launched from k vertices on the same graph G. The pivot queries are selected to assist the application of memoization. We will give more details of them in Section 5.

2.2 Memoization

Memoization is an optimization concept used primarily to speed up computer programs by storing the results of expensive processes. Although memoization also stores useful results for reuse, it is fundamentally different from the caching techniques such as buffering [3, 27, 52]. A caching technique tends to buffer raw data that would be frequently accessed, while memoization emphasizes storing the processed results.

In previous studies, the memoization technique is mainly used for processing queries where the intermediate results of the query are memoized in the scope of the entire graph [8, 21, 43]. In contrast, KGraph is the first graph processing memoization engine for designing memoization within graph partitions to support CGQs on large graphs.

A graph query q can be represented as a tuple $\langle S, U, W\rangle$. S(*Source Vertices*) denotes the set of vertices from which the query is initiated. These vertices are akin to the "starting points" in various graph algorithms such as Single Source Shortest Path (SSSP) or a specific vertex in a Random Walk. U(*Visited Vertices*) is the set of all vertices that are visited or processed during the execution of the query. This includes all vertices that potentially influence or are influenced by the computation initiated from S. W(*Workload*) is a quantifiable measure representing the total computational effort or resource utilization during the query execution, typically measured as the total number of vertex updates.

Definition 2.1. A graph query is considered **memorizable** if it adheres to the following criteria:

(1) *Initiation*: $S \neq \varnothing$. The set of source vertices S must be non-empty. This criterion ensures that the query is initiated from a well-defined subset of vertices in the graph, distinguishing it from global graph computations like PageRank [29] or Triangle Counting [42] which do not start from specific vertices.
(2) *Overlap Computation*: For any two memorizable queries $q_1 = \langle S_1, U_1, W_1 \rangle$ and $q_2 = \langle S_2, U_2, W_2 \rangle$, the intersection of their visited vertices $U_1 \cap U_2 \neq \varnothing$. This overlap signifies that there is shared computation between the queries, making the memoization of their results potentially beneficial.
(3) *Workload Reduction*: The combined workloads of q_1 and q_2 (i.e., $W_1 + W_2$) should be reducible by leveraging memoized data from the intersecting set $U_1 \cap U_2$. This implies that reusing previously computed results for the overlapping vertices should decrease the overall computational effort required when these queries are evaluated concurrently.

Applicability. A substantial number of concurrent graph query problems exhibit properties that can be generally categorized into three types:

- **Queries satisfying the graph triangle inequality** [19]. For any three vertices u, v, and w, the graph triangle inequality is expressed as $\texttt{property}(u,v) \oplus \texttt{property}(v,w) \geq \texttt{property}(u,w)$, where `property` could be any metric between two vertices. An example is the Single Source Shortest Path (SSSP), where $\texttt{dist}(u,v) + \texttt{dist}(v,w) \geq \texttt{dist}(u,w)$ holds, with `dist` representing the shortest path distance. For such queries, memoization can be employed to precompute $\texttt{property}(v,*)$, thereby reducing the computations for $\texttt{property}(u,w)$. Representative applications include SSSP, SSWP, SSR, Viterbi [23], and Radii estimation [33].
- **Queries with optimal substructures**. Many graph problems, akin to dynamic programming, exhibit optimal substructures, implying that an optimal solution to a graph problem can be constructed from optimal solutions to its subproblems. These subproblem solutions can be reused across different queries. Notable algorithms displaying this property include Minimum Spanning Tree (MST) [15] and Steiner Tree [5].
- **Vertex-centric index-based query optimization**. Several methodologies construct sophisticated indices capturing multiple layers of neighborhood information around a vertex, which are extensively used in graph databases such as JanusGraph, Neo4j, ArangoDB, and Aerospike for query optimization. In graph analysis tasks, such as subgraph matching, these indices facilitate the rapid and accurate identification of subgraph structures by filtering out non-matching parts of the target graph [25, 36].

3 MOTIVATION

CGQs can perform repetitive operations since they are homogeneous and execute on the same graph. Motivated by the intuition, we conduct extensive experiments to study whether we can accelerate CGQ processing by memoizing results of completed queries to facilitate unprocessed queries. We select concurrent SSSP queries as representative workloads because SSSP is the typical traversal-based query and concurrent SSSP is one of the most used benchmarks for CGQs [27, 44]. We obtain similar observations from other CGQ workloads such as SSWP, SSR, and random walks.

Settings. We design and implement two approaches to examine the impact of memoization. Following existing CGQ processing methods [27, 44], a hand-tuned baseline assigns each query in a set Q of SSSP queries to a CPU core and executes them simultaneously without memoization. In contrast, the approach with memoization executes a set Q' of queries, called *memoized queries*, and stores their results before executing Q. The result of query $q' \in Q'$ is an array recording the path distances from the source $u_{q'}$ to other vertices. When the processing of query $q \in Q$ reaches $u_{q'}$, we reuse the memoized results of q' as follows: update the path distances from u_q to another vertex u if $dist(u_q, u_{q'}) + dist(u_{q'}, u) < dist(u_q, u)$ (i.e., the current distance from u_q to u is greater than that via $u_{q'}$). Reusing q' prunes invalid distance update operations and reduces random memory accesses in the processing, while the overhead lies in scanning the result of q', the size of which is $|V|$.

We execute those queries on a set of real-world and synthetic graphs to assess their performance on graphs with variant properties. Particularly, for the real-world graph, we extract the `Jakarta` city map of 3.9M vertices and 4.2M edges from *OpenStreetMap* [2], and we use the pick-up locations of the actual ABC taxi hailing service during peak and off-peak hours as the source vertices for queries. For synthetic graphs, we use a widely used graph generator, R-MAT [6], to generate five scale-free graphs [3]. The number of edges scales from 10K, 100K, 1M, 10M, to 100M. Given an edge, we generate a value from 1 to 1,000 uniformly at random and set it to the edge weight. The source vertex for synthetic graphs in both Q and Q' is selected uniformly at random from the graphs.

3.1 Profiling Results

Memoization speedup. We set $|Q|$ (i.e., the number of CGQs) to 1,000 and vary $|Q'|$ (i.e., the number of queries whose results are memoized) from 2 to 128. Figure 3 presents the speedup of the hand-tuned implementation with memoization over the baseline in terms of the execution time of Q. We can see that the speedup grows with the number of memorized queries increasing on graphs with 10K and 100K edges. However, the performance of the approach with memoization degrades on `Jakarta` and graphs with the size scaling from 1M to 100M, and more memoized queries result in worse performance.

Micro-benchmarks. To thoroughly understand the degradation of memoization on large graphs, we design three micro-benchmarks to further profile the workload.

Operation Reduction Ratio Distribution. SSSP updates the distance from the source vertex to other vertices in a greedy manner. As such, we use the number of distance update operations to measure the workload of a query. Let ω and ω' represent the workload of a query without and with memoization. The *operation reduction ratio* r is equal to $\frac{\omega - \omega'}{\omega}$. Figure 4 illustrates the operation reduction ratio distribution of the 1,000 queries with 32 memoized queries. As

[2]https://www.openstreetmap.org/
[3]$a = 0.5, b = c = 0.1$

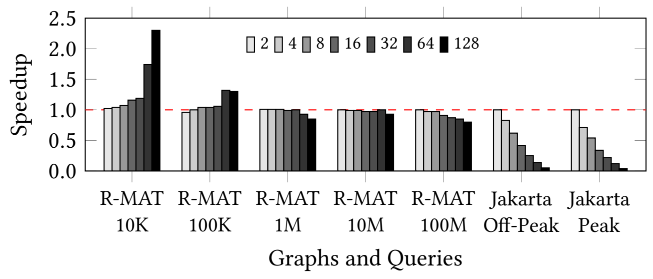


Figure 3: The Speedup of the Memoization Method With the Number of Memoized Queries Varied From 2 to 128.

shown in the figure, the ratio on small graphs is much higher than that on larger graphs. Specifically, the ratio of around 100 queries on the graph with 10K edges is above 0.9, whereas the ratios of all queries on the graph with 100M edges and `Jakarta` are both below 0.1. Consequently, the overhead of memoization on large graphs offsets its benefit, and results in the degraded performance in Figure 3.

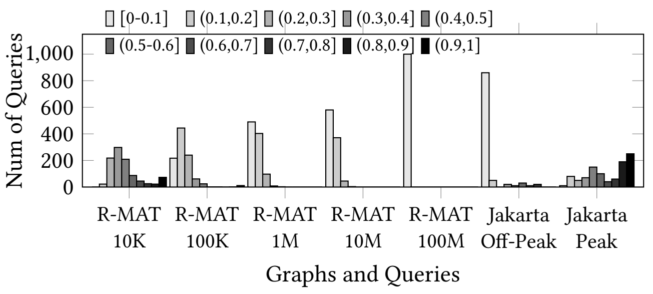


Figure 4: The operation reduction ratio distribution of 1,000 CGQs with 32 memorized queries.

Total Operation Reduction Ratio Distribution. We further examine the impact of each individual memorized query q' in terms of the *total operation reduction ratio*: $s = \frac{\sum_{q \in Q}(\omega_q - \omega'_q)}{\sum_{q \in Q}\omega_q}$ where ω_q and ω'_q represent the operations of q without and with memoized query q', respectively. Figure 5 presents the total operation reduction ratio distribution of the 32 memoized queries when executing 1,000 CGQs. The impact of memorized queries varies greatly. While many queries have a reduction ratio of less than 2%, there are queries with remarkably high reduction ratios between 4% to 16%.

Query Result Overlapping Distribution. SSSP finds the shortest path from the source vertex to all vertices. If the results of queries in Q are highly overlapped, then those queries incur a substantial number of repetitive operations. Therefore, we examine the *query result overlapping distribution* to exploit the problem. Specifically, given $q \in Q$, we generate the *shortest path tree* [13], which consists of the shortest paths from the source vertex to each vertex. Given a set Q of CGQs, X is the set of edges in at least one shortest path tree of queries in Q. We measure the query result overlapping distribution as the percentage of edges in X that belong to at least T distinct shortest path trees with $2 \leq T \leq |Q|$. As constructing shortest path trees for 1,000 queries on large graphs is time-consuming, we randomly select 100 queries from the 1,000 queries as representatives.

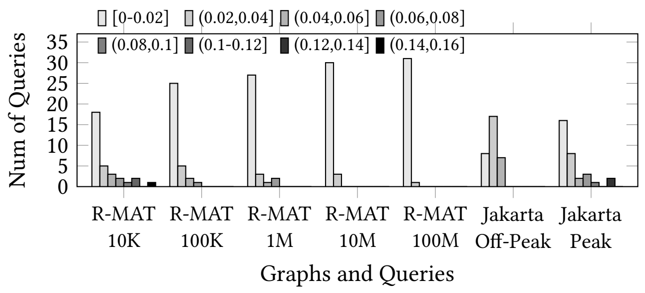


Figure 5: The Total Operation Reduction Ratio Distribution of 1,000 Concurrent Queries With 32 Memorized Queries.

Figure 6 presents the query result overlapping distribution with T varied from 2 to 100. Around 80% of edges in X appear in at least two distinct shortest path trees. The value is higher on large graphs than that on small graphs. On the real-world graph `Jakarta`, the ratio even reaches 99% and remains stable as T increases. In other words, the queries on large graphs have a strong locality though the overall operation reduction ratio is low.

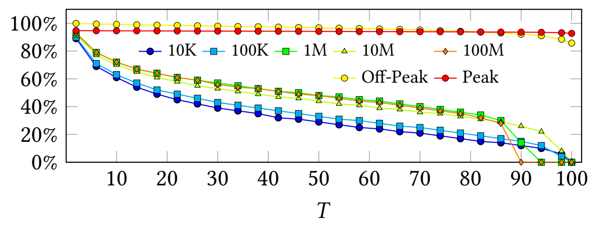


Figure 6: The Percentage of the Number of Edges in x That Belong to at Least t Distinct Shortest Path Trees.

3.2 Observations and Implications

We make three important observations from the profiling results:

Observation 1. *Memoization shows degraded performance on large graphs because the operation reduction ratio is low and the overhead of scanning the memoized results offsets the reduction benefit.*

Observation 2. *The impact in reducing redundant operations varies significantly among different memoized queries.*

Observation 3. *The query results on large graphs are still heavily overlapped among CGQs.*

The observations lead to the following implications for the design and implementation of an efficient memoization engine for CGQs. First, the memoization technique is non-trivial for large graphs but there are still substantial computation sharing opportunities. Second, storing the complete result of a "memoized" query leads to prohibitive overhead on large graphs. Third, the memoized queries should be carefully selected to improve the reduction ratio.

4 OVERVIEW OF KGRAPH

As shown in Section 3, the baseline approach of memoization slows down the CGQ processing on large graphs. To this end, we propose KGraph, an efficient memoization engine for CGQs, with two major optimizations for efficiency. The first optimization is *partition-based memoization with adaptation* (Section 5.1), which imitates the memoization on small-scale graphs to reduce the overhead of scanning

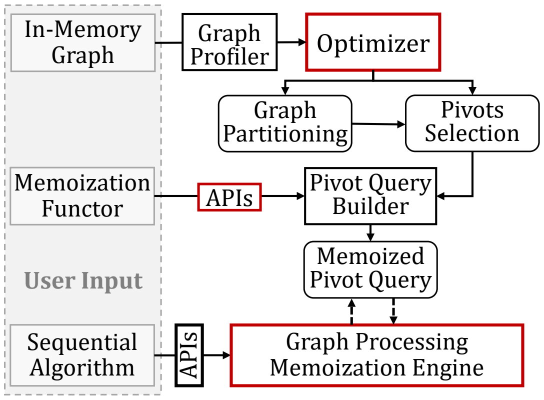


Figure 7: Overview of KGraph.

the memoized results. The second optimization is *on effective pivot query selection* (Section 5.2), which pre-selects pivot queries that have a high operation reduction ratio to other CGQs. We provide the technical details of these optimizations in Section 5.

Graph processing system. Figure 7 shows the system architecture of KGraph. KGraph takes the in-memory graph data, a user-defined memoization function and a sequential algorithm as inputs. The system exploits the embarrassingly parallelizable workloads among CGQs by allocating one CPU thread per query and automatically parallelizes the CGQ processing efficiently with memoization enabled. The three major components are listed as follows.

Memoization engine. To process every single query, the sequential program provided by users through the given APIs will be executed. While processing, the memoization engine checks if any pivot vertex is reached that could potentially reduce the workload, reusing its results if applicable. The process includes efficiently scanning the memoized results and calling the update functor of memoization APIs. This update adjusts the query's context and switches to another operational state for continued processing.

APIs. We put our efforts into easing the programming of different applications on KGraph. However, we recognize that the implementations of memoization vary significantly for different CGQs. To this end, KGraph first extends the APIs from ForkGraph [27] to ease the programming for CGQ processing. Based on that, we introduce the memoization functor that allows the users to program the logic to update the results of a query given the memoized results of other queries provided.

Optimizer. Many factors affect the performance of KGraph in handling CGQs. These factors include the partition size and pivot query selection. We propose an optimizer in KGraph to atomically tune the graph processing engine for high performance.

The optimizer contains a cost model to estimate the performance of CGQ processing on different graph partition sizes and thus to guide the graph partitioning adaptively for different graph inputs. Besides the cost model, we also train a decision tree to generate high-performance configurations for pivot query selection. The features used by the decision tree include the number of vertices, edges, the average degree, the maximum degree, and a few others. The configurations generated by the decision tree include two factors: the criteria of pivot query selection and the number of pivot queries to memorize.

Algorithm 1 Execution Flow of KGraph on CGQ Processing.

```
1:  LoadGraphPartitions()
2:  Pivots ← PivotQuerySelection(StrategyConfig)
3:  InitMemoizationStorage(Pivots)
4:  processor ← KGraph(MemoizationFunctor())
5:  parallel_for_each $q \in Q$ do
6:      processor.run($q$)
7:  procedure run($q$)
8:      while $q$'s activated vertices set $F \neq \emptyset$ do
9:          for_each $f \in F$ do
10:             if $f \in$ Pivots then
11:                 Perform MemoizationFunctor($f$) to update $q$
12:             else
13:                 Compute($f$) to update $q$
```

Overall execution flow. Algorithm 1 shows the overall execution flow of KGraph on CGQ processing. To prepare the application of memoization on graphs, we first load the partitioned graph into KGraph (Line 1). Within each partition, we select a few distinct vertices in each partition as the source vertices of pivot queries as shown in Lines 2 and 3. After preparation, KGraph initializes a processor instance by integrating the memoization functor programmed by users in Line 4 and executes the queries concurrently in Lines 5 and 6.

Procedure Run in Line 7 shows the process of how KGraph executes a query. Basically, it checks if the query reaches any pivot queries (Line 10) and reuses the memoization if so (Line 11) or processes it directly if not. Note that we hide some details in Algorithm 1: KGraph will first process the pivot query if it has not yet been processed and memoizes the results of the pivot query.

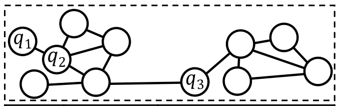
5 MEMOIZATION ON GRAPH

In this section, we present the partition-based memoization optimization that is used to reduce the overhead of performing memoization on large graphs. We then talk about the strategic pivot query selection, which lets KGraph memoize queries that could contribute more to the processing of other queries. Finally, we will discuss the applicability of memoization on graphs for CGQ processing.

5.1 Partition-Based Memoization

As we observe in Section 3, performing memoization on small-scale graphs exhibits higher ratios of operations reduced than that on large-scale graphs. It is because the computation sharing is sensitive to *locality*. Except if a memoized query starts from the same source vertex of a query q, which is rare, the memoized results can only be partially used by q. Usually, the closer the queries are, the more computation sharing they have. As the queries are close to the memoized queries on small-scale graphs, the ratios of operations reduced are high.

Benefits and overheads of memoization. The application of memoized results consists of many memoization operations. A memoization operation first reads one value from the memoized result and updates the corresponding value of the query that is being processed. We consider a memoization operation useful when another query benefits from the result and reduces its computation workload; the operation is wasted when it fails to update the results



Query	Memoized	Memoization Operations	
		Useful	Wasted
q_1	q_2	9	0
q_1	q_3	4	5
q_2	q_1	0	9
q_2	q_3	4	5
q_3	q_1	0	9
q_3	q_2	1	8

Figure 8: The numbers of useful and wasted memoization operations when solving a single query.

of another query. It is wasted because storing and checking the memoized results bring overhead to the processing. For example, it takes $O(|V|)$ memoization operations when reusing the memoized results of one query for a traversal-based query, such as SSSP, SSWP, and SSR. As discussed in Section 3, performing memoization shows degraded performance on large graphs because the overhead of scanning the results offsets the benefits.

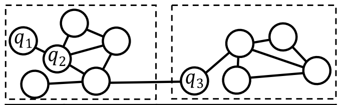
We use an example of three SSSP queries on a graph with 11 vertices in Figure 8 as an example to demonstrate the benefits brought (operations reduced) and overhead involved. In the example, the source vertices of q_1 and q_2 are close, and the source vertex of q_3 is a bit far from them, located near the center of the graph. In our setting, we analyze the condition of executing one query given that another query has been executed and memoized. We collect two factors in the table, the numbers of useful and wasted memoization operations. A higher number of useful memoization operations means higher memoization effectiveness, which is computed as the number of useful operations divided by the total number of operations. On the contrary, a higher number of wasted operations means a higher overhead involved.

When executing q_1, as the source vertex of q_2 is the only way q_1 reaches other vertices, we reuse the memoized results of q_2 using graph triangle inequality [19] by simply adding a unit weight to the results of q_2. As shown in the first row of the table provided in Figure 8, q_1 could use nine out of ten of the memoized results, which exhibits high effectiveness.

However, if we memoize q_3 instead, the effectiveness is lower during the processing of q_1 and q_2 as the source vertex of q_3 is "far away" from the others. Nevertheless, the memoized results of q_3 exhibit a good locality that can help the processing of q_1 and q_2 on the four vertices located on the right side of the source vertex of q_3 in the graph.

Fine-grained memoization on graph partitions. To improve the benefits and reduce the overhead of memoization on large graphs, we proposed to imitate memoization on small-scale graphs. We first divide the large-scale graph into partitions and only perform memoization within partitions to make the application of memoized results fine-grained. Instead of memoizing the results of a query on the whole graph, we memoize its partial results within the partition where its source vertex is located.

We use the previous example given in Figure 8 to demonstrate the effectiveness of the partition-based approach. We divide the graph into two parts, as shown in Figure 9. Note that we pick two



Query	Memoized	Memoization Operations	
		Useful	Wasted
q_1	q_2, q_3	9	0
q_2	q_1, q_3	4	4
q_3	q_1, q_2	1	4
q_1	q_2, q_3	9	0

Figure 9: Partition the graph given in Figure 8 to reduce the number of wasted memoization operations and maintain high memoization effectiveness.

queries to memoize in this example as we only memoize the results within the partition. The two queries take the same amount of memory space as picking one in Figure 8.

We observe that the wasted memoization operations can be significantly reduced and the effectiveness has then been improved. Specifically, the numbers of wasted operations for q_2 are 9 and 5 when memoizing q_1 and q_3, respectively. From another perspective, the memoized results of q_3 can be fully reused by the other two queries, which shows high effectiveness. The partition-based approach reduces the number of wasted operations to 4 while maintaining higher memoization effectiveness than the baseline approach on average.

Adaptation for different graphs. We can expect that the smaller the partition is, the higher the memoization effectiveness would be because it is more fine-grained. However, we also notice that smaller partitions mean more partitions to manage in real-time. Essentially, there are more workloads shifted from intra-partition processing to inter-partition processing. As a result, the fine-grained partition-based memoization brings more workloads in practice, which could offset the benefits of the high effectiveness achieved.

To address this issue, we present an adaptation solution with a cost model in the optimizer to configure the partition sizes for different graphs. Generally, the cost model estimates the cost of intra-partition processing and the workload of inter-partition processing via cross-partition edges. Given a graph, the cost model takes the features of the graph (including $|V|$ and $|E|$) as inputs and then outputs a reasonable partition size.

Given that CGQs are processed independently, we analyze the cost associated with a single query $q\langle S_q, U_q, W_q\rangle$ considering a set of pivot queries $H = \{h_1, h_2, \ldots, h_k\}$. The overall cost is then a multiple of the cost for this individual query.

Each pivot query h within H is characterized by three metrics: $\langle x_h, \mathsf{Work\_Reduction}_h, \mathsf{Mem}_h\rangle$, where x_h denotes the source vertex. The metric $\mathsf{Work\_Reduction}_h$ quantifies the computational reduction achieved by hitting this pivot and utilizing the memoized results, while Mem_h represents the memory required to store these intermediate results. The values for $\mathsf{Work\_Reduction}_h$ and Mem_h are specified by the application and will be further elucidated through two detailed examples subsequently.

Partition Memory Constraint. In cache-efficient partition-based graph processing systems [27], the graph data is divided into partitions sized to fit LLC, with only one partition processed at a time. When memoization is performed, results from additional pivot queries are stored and frequently accessed. To ensure cache-resident graph processing, the following constraint must be met:

$$\frac{|E|+|V|}{|P|}+\frac{k}{|P|}\times \mathsf{Mem}_{\mathsf{avg}} \leq \mathsf{Size}_{\mathsf{LLC}}$$

where $(|E|+|V|)/|P|$ represents the size of the partitioned raw graph data, such as memory for CSR storage in a partition. The term $k/|P|\times \mathsf{Mem}_{\mathsf{avg}}$ accounts for the additional memory overhead caused by memoization within that partition.

Memoization-Based Cost Function. The cost of evaluating a query q in KGraph can be estimated as

$$W_q + \mathsf{Scheduling\_Overhead} - \mathsf{Pivot\_Hits} \times \mathsf{Work\_Reduction}_{\mathsf{avg}}$$

subject to the partition memory constraint.

In this cost function, W_q represents the inherent workload of the application when evaluated using the fastest known sequential algorithm. The term Scheduling_Overhead encompasses the costs associated with inter-partition processing, including the coordination and status updates required across partitions, which can be modeled as $O(|P|^2)$ [27]. The factor Pivot_Hits, defined as $\mathsf{Hit\_Ratio}\times|U_q|$, quantifies the number of times a pivot query is utilized, where Hit_Ratio represents the likelihood of hitting a pivot. This ratio is influenced by the number and in-degrees of selected pivot vertices, formally given by $\mathsf{Hit\_Ratio} = \frac{\sum_{h\in H}\deg_{\text{in}}(x_h)}{\sum_{v\in V}\deg_{\text{in}}(v)} = \frac{\sum_{h\in H}\deg_{\text{in}}(x_h)}{|E|}$. Here, $\deg_{\text{in}}(x_h)$ is the in-degree of pivot h. The $\mathsf{Work\_Reduction}_{\mathsf{avg}} = W_{\mathsf{OpCut}} - W_{\mathsf{LookUp}}$ calculates the average computational reduction from utilizing memoized results, where W_{OpCut} denotes the number of operations actually saved by the algorithm. Memoization incurs costs, as the pivot stores all intermediate results within its partition, which are not tailored for specific queries. Upon hitting a pivot, the system looks up these results to perform useful memoization operations to update the ongoing query. Typically, the size of these stored results, $\mathsf{Mem}_{\mathsf{avg}}$, significantly impacts W_{LookUp}, and this impact varies across different applications.

We categorize traversal-based algorithms like SSSP, SSWP, and SSR, to have a complexity similar to SSSP queries. The implementation for these and walk-based algorithms is depicted in Algorithm 2. For these two types of algorithms, the memoized data will be the shortest paths of k'-nearest neighbors of a pivot vertex and all fix-length paths starting from the vertex respectively, which can be used to measure $\mathsf{Mem}_{\mathsf{avg}}$ and W_{LookUp}. W_{OpCut} for SSSP is derived from the number of vertices within the memoized results that are also children of the pivot in the shortest path tree of the input query. Walk-based algorithms calculate W_{OpCut} using reduced walk lengths. To enhance adaptability to various graph structures, we employ a supervised machine learning model trained on extensive data from different partition sizes and applications within KGraph. This model guides the partitioning strategy for new graphs.

5.2 Strategic Pivot Query Selection

Based on the observation that some vertices contribute more to the processing of other queries, we propose to involve these *important* vertices and create a bunch of pivot queries starting from them. Instead of exploiting the computation sharing among CGQs, we are looking into whether a query can leverage any results of the pivot queries. The pivot query selection has the following two questions to answer. The first question is what criteria should be used to select vertices as source vertices for pivot queries. The second question is how many pivots queries we should select to memoize.

Challenges for finding the optimal solution. The optimal strategy for the pivot query selection is to find a set of pivot queries that will minimize the workload for CGQ execution. However, achieving the optimal solution is challenging due to the complexities involved. These complexities arise from various factors, including the input graph's structure, the graph partition strategy, the algorithmic logic of user-defined applications, and the distribution of source vertices across the launched queries on the graph. Moreover, the effort to identify the optimal strategy can adversely impact system performance, resulting in increased latency and reduced throughput which are both critical metrics in CGQ processing systems [41]. Given these challenges, we opt for designing selection strategies based on a series of heuristics. In practice, these heuristics have proven effective, as evidenced by our experimental results.

Heuristics for pivot query selection. As we see that some queries contribute significantly more than others in Section 3, we intend to memoize these queries to improve the efficiency of memoization on graphs. Empirically, high-degree vertices are usually more important than others. However, as we propose to use the partition-based memoization, we find that selecting pivots according to the graph partitions could also be effective. Therefore, in KGraph we use two criteria to select pivot queries. One is the degree of a vertex (number of in-coming edges), and the other one is whether the vertex is a boundary vertex. Boundary vertices are defined as vertices that have edges incoming from other partitions. Based on these two criteria, we propose the following six strategies, *S1*-*S6*, which cover most of the pivot selection approaches.

- *S1*: randomly select k vertices in each partition.
- *S2*: randomly select $|P|k$ vertices on the graph.
- *S3*: randomly select k boundary vertices in each partition.
- *S4*: sort vertices by degree and select the top k vertices in each partition.
- *S5*: sort vertices by degree and select the top $|P|k$ vertices on the graph.
- *S6*: sort vertices by degree and select the top k boundary vertices in each partition.

Number of pivot queries. In the strategies listed above, we use k to denote the number of pivot queries selected for each partition on average. We find that configurations of the number of pivot queries that bring high performance greatly depend on the graph structures. Specifically, we tend to select more vertices on sparse graphs like road networks and select fewer vertices when the graphs become denser.

We find that sparse graphs like road networks tend to perform better when selecting boundary vertices as the source vertices of pivot queries. The reason is that as boundary vertices work like the gateways for partitions, all kinds of graph processing must visit a boundary vertex at the time they reach a partition. On a sparse graph, the number of boundary vertices is small, and it does not bring huge memory cost even if we select all the boundary vertices as pivots. For example, around 10GB memoization space is required

Table 1: Input Graphs ($\overline{d}$ is the Average Degree).

Graph	Source	#V	#E	$\overline{d}$	Memory	$\|P\|$
`Ca`	California [9]	1.9M	4.6M	2.4	0.07GB	4
`Us`	USA [9]	23.9M	57.7M	2.4	0.82GB	7
`Eu`	Europe [2]	50.9M	108.1M	2.1	1.65GB	9
`Lj`	LiveJournal [24]	4.8M	87.5M	18.0	1.04GB	11
`Or`	Orkut [24]	3.0M	117.1M	38.1	1.37GB	12
`Tw`	Twitter [22]	61.6M	1.5B	23.8	17.27GB	15

for the whole US road network when memoization the results of queries at boundary vertices, which is quite easy to be satisfied by a compute node.

Differently, we tend to select high degree vertices as the source vertices of pivot queries for dense graphs like social networks. The reasons are listed as follows. First, memoizing all queries at boundary vertices is not realistic as the total number of boundary vertices is significantly larger than that on a sparse graph. Second, the diameter of a dense graph is generally small, and the boundary vertices are almost always just one hop away from the large-degree vertices. Large-degree vertices are more frequently visited than other vertices, including the boundary ones. As it is not realistic to memoize too many pivot queries, KGraph is tuned to select the top 2 – 10% of the vertices in each partition to be the source vertices of pivot queries. In particular, the denser the graph is, the lower percent of vertices are preferred.

Decision tree-based pivot selection. The graph structures, including the numbers of vertices and edges, the maximum degree of vertices, and other factors, affect the performance of KGraph on CGQ processing. Instead of configuring the system empirically, we propose a trained decision tree model in the optimizer to atomically configure pivot query selection for high performance in KGraph. The decision tree is trained on a large space of data points collected by executing KGraph on various graphs with different strategies for pivot query selection. Notably, we prepare a few approaches, listed as follows, to generate various graph data to avoid overfitting during training.

- Vary the degree and *abc* parameters in R-MAT, with the numbers of edges varying from 10K to 100M.
- Sparsify a real-world graph by sampling 10% – 90% of the original edges randomly.
- Densify a real-world graph by creating 50 – 100% edges between vertices randomly.

For each graph data generated, we thoroughly run KGraph with various configurations of pivot selection strategies and different numbers of pivot queries. We extract the features mentioned above from every graph data and we take the configurations that generate the best performance on every graph to be the training targets.

6 EVALUATION

6.1 Experimental Setup

Hardware configuration. We conduct experiments on a Linux server with a 32-core AMD EPYC™ 7543 CPU at 2.8GHz with hyperthreading disabled (LLC size: 256MB). The main memory is 256GB. We compile all the implementations using g++ 7.5.0 with -O3 flag and OpenMP enabled.

Algorithm 2 Example Interface Implementation for Two Types of Applications.

```
# SSSP Interface
procedure InitMemoizationStorage(Pivots $P$, MemoizedSize $k$)
  parallel_for_each $p \in P$ do
    $D \leftarrow$ Find $k$ nearest vertices from $p$ within the partition of $p$
    for_each $v_i \in D$ do
      $M_p$.push(<$v_i$, distance$_p[v_i]$>)
procedure MemoizationFunctor(Pivot $p$)
  for_each <$v_i$, distance$_p[v_i]$> $\in M_p$ do
    distance[$v_i$] $\leftarrow$ min(distance$_p[v_i]$ + distance[$p$], distance[$v_i$])

# RandomWalk Interface
procedure InitMemoizationStorage(Pivots $P$, Length $l$)
  parallel_for_each $p \in P$ do
    $M_p \leftarrow$ All paths with the length $l$ starting from $p$ and the corresponding probabilities.
procedure MemoizationFunctor(Pivot $p$)
  Perform a weighted sampling of a path $w$ from $M_p$
  Append the path $w$ to the walk path
  walkLength $\leftarrow$ walkLength $- l$
```

Implementation details. KGraph uses the codebase from ForkGraph [27] because it is the state-of-the-art programming system targets processing CGQs. ForkGraph proposes a cache-efficient buffer execution model to manage operations of graph queries. KGraph reuses all the features designed for handling CGQs provided by ForkGraph and injects the memoization features into the intra-partition processing part. First, KGraph manages the memoization storage by using a pre-allocated space to store memoized results and uses a hash-based lookup table to locate the corresponding data quickly. Second, KGraph re-designs the functors in ForkGraph to allow them to include the functors to check the availability of memoized results and to reuse them.

The optimizer in KGraph consists of the decision tree-based model for pivot query selection (Section 5.1) and a cost model for adaptive partition (Section 5.2). Specifically, given different query types and the input graph, the cost model generates the suitable partition size; the decision tree takes the highest degree of vertices, the average degree, the numbers of partitions, vertices, and edges as the input features to generate the following configurations: 1) pivot selection strategies (S1 to S6), and 2) the total number of pivot queries. We use the numbers of partitions $|P|$ provided by the cost model, as shown in Table 1, for each graph data.

Comparisons. Our study involves a comparative analysis of KGraph against ForkGraph [27] and Glign [47]. ForkGraph is the state-of-the-art graph processing systems for handling CGQs, and it has been previously shown in [27] to significantly outshine counterparts such as Ligra [33], Gemini [53], and GraphIt [49]. On the other hand, Glign stands as a leading method specifically optimized for concurrent traversal-based queries on power-law graphs. Glign demonstrates remarkable performance advantages over other concurrent graph processing systems, including GraphM [50] and Krill [7]. Since Glign does not support random walk-based applications, we have excluded it from the corresponding comparison.

We measure the time for each system to complete the processing of CGQs. The measurement excludes the time spent partitioning

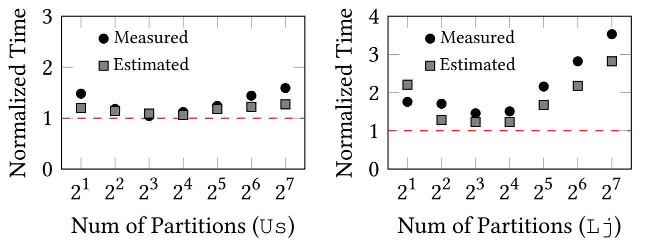


Figure 10: The actual execution time and estimated time of solving concurrent SSSP queries on `Us` and `Lj` using different partitions sizes. The results are all normalized to the shortest execution/estimated time.

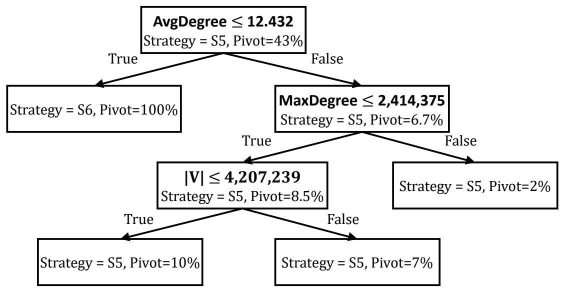


Figure 11: The Visualization of the Decision Tree Trained in KGraph.

the graph or reading data from the disk because we focus on optimizing the in-memory computation. Note that the time spent on memoization is included for KGraph as it is part of the execution in runtime.

Applications. We evaluate the performance of KGraph and ForkGraph on five CGQ applications, SSSP, SSWP, SSR, DeepWalk, and PPR. The example interface implementation can be found in Algorithm 2. We configure the five applications as follows.

- SSSP, SSWP, and SSR: we randomly sample a batch of 512 vertices as the source vertices for each test case. All these queries are submitted together.
- DeepWalk and PPR: we randomly sample a batch of 1,000,000 vertices as the source vertices for each test case. The walk lengths for both are set at 1000, with a damping factor of 0.85 for PPR.

Datasets. Besides the five graphs generated using R-MAT used in Section 3, we select six real-world graphs as listed in Table 1. These datasets are publicly available, and they are widely used in the existing works [33, 49, 53] to benchmark algorithms and frameworks. Following the experimental studies in [33] and [10], we create weighted graphs by selecting edge weights between $[1, \log |V|)$ uniformly at random. The datasets are categorized into two types: `Ca`, `Us` and `Eu` represent road networks, while `Lj`, `Or` and `Tw` denote social networks. By default, graph partitioning is performed using METIS [20], except that `Tw` is randomly partitioned because METIS runs out of memory at runtime.

6.2 Optimizer Evaluation

The optimizer contains two major components, the cost model for graph partitioning and the decision tree for pivot selection strategies. To demonstrate that our optimizer is accurate for picking high-performance configurations for KGraph, we first compare the estimated execution time generated by the model with the actual execution time of KGraph. We then give the visualization of the decision tree model trained in this work to show that it is able to capture the graph structure for generating high-performance configurations.

Figure 10 plots both the normalized actual execution time and estimated execution time of solving concurrent SSSP on two representative graphs. We can see that the model can capture the trends of the changes in execution time though it cannot find the optimal number of partitions for every graph. Therefore, we can leverage the trend estimated by the model to configure KGraph for high performance.

We draw the decision tree trained in KGraph in Figure 11 based on the visualization exported from the model. We can observe from the figure that the decision tree firstly captures the density of a graph in the first level. Specifically, for graphs with low average degrees (`AvgDegree` ≤ 12.432 in this case), we can get the configuration of *Strategy = S6, Pivot*=100%, which represents the strategy of selecting all boundary vertices as pivots for sparse networks. Road networks are regressed in this branch. On the right child of the root node, the data sets are further divided according to the maximum degrees. If the graph is highly skewed like `Tw`, the pivot selection tends to only select those high-degree vertices. The rest nodes are used to fine-tune graphs that are dense but not so skewed. From the model we can see that it can exhibit the accurate modeling of the most important performance factors among distinctive features of graph data. Note that due to the limited space, we slightly prune some leave nodes to better visualize the tree without affecting the structure significantly.

6.3 Overall Performance Comparison

Finding (1) : KGraph significantly outperforms the state-of-the-art concurrent graph processing frameworks.

We summarize the overall speedups of KGraph over ForkGraph in Table 2. First, KGraph demonstrates superior performance in traversal-based queries such as SSSP, SSWP, and SSR, outshining ForkGraph in all cases with up to 12.5× speedup and an average of 5.9×. While KGraph exhibits comparable results to Glign, which is tailored for traversal-based queries on power-law graphs, it significantly outperforms Glign in road network scenarios, aligning with findings reported in [47].

Second, we notice that KGraph almost doubles the speeds when processing random walk queries on road networks but only shows around 20% performance improvement on social networks. The reasons are listed as follows. 1) The strategy of selecting boundary vertices as the source vertices of pivot queries on road networks can cover 100% of the paths that route from outside to the inside of the partitions. However, it is not realistic to use the same strategy on social networks because the number of boundary vertices is large. 2) The random walk queries are relatively simple. In the partition-based processing of KGraph, the cost of processing random walk is remarkably close to the memory access of reusing memoized results. Memoizing additional boundary vertices results in increased memory usage, and therefore, which in turn leads to more LLC

Table 2: Overall Speedups of KGraph Over ForkGraph and Glign.

Dataset	SSSP			SSWP			SSR			DeepWalk		PPR	
	ForkGraph	Glign	KGraph	ForkGraph	Glign	KGraph	ForkGraph	Glign	KGraph	ForkGraph	KGraph	ForkGraph	KGraph
`Ca`	76.8 s	0.31×	6.45×	45.6 s	0.46×	7.08×	40.9 s	0.75×	4.00×	73.8 s	1.69×	87.1 s	2.06×
`Us`	1115.8 s	0.27×	3.47×	690.5 s	0.57×	3.89×	1001.8 s	1.63×	4.51×	73.6 s	1.56×	83.3 s	1.63×
`Eu`	2307.1 s	0.38×	5.51×	1896.7 s	0.93×	7.05×	2030.2 s	0.93×	7.44×	72.8 s	1.45×	71.4 s	1.65×
`Lj`	233.3 s	5.53×	6.48×	260.9 s	8.57×	12.59×	121.8 s	7.30×	8.15×	91.9 s	1.23×	74.3 s	1.24×
`Or`	85.4 s	1.28×	5.29×	115.2 s	1.54×	6.46×	42.4 s	1.05×	4.74×	98.9 s	1.41×	86.1 s	1.83×
`Tw`	2474.3 s	3.51×	3.54×	3130.8 s	5.39×	6.99×	1178.1 s	4.14×	3.55×	109.4 s	1.23×	95.5 s	1.26×

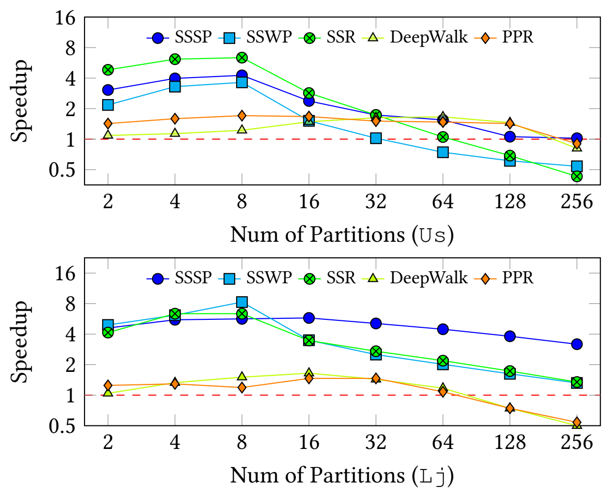


Figure 12: Speedups of KGraph Over Forkgraph Using Different Partition Sizes.

misses due to the random visiting of pivots. Fortunately, social networks typically have much smaller diameters compared to road networks. High-degree vertices that are memoized are revisited more frequently than others, making KGraph perform better than ForkGraph in such scenarios.

6.4 Effects of Graph Partitioning

Finding (2) : The configurations of graph partitioning generated by the optimizer in KGraph generally make it run at a good performance.

In this part, we analyze the effects of partitioning the graph into varied sizes. We select a road network case and a social network case to present the results for simplicity. The numbers of partitions used for these two graphs in Table 2 are 7 and 11, respectively. In our testing, we vary the number of partitions from 2 to 256 to analyze the trend of performance changes. The graph partitioning is done using METIS [20].

First of all, from the figures we can see a trend that KGraph has a good performance when there are not many partitions (< 8) and the performance becomes poor when there are too many partitions (≥ 64). There is a trade-off between the partition-based processing and memoization effectiveness. When the graph is cut into small partitions, the memoization effectiveness is high but the underlying partition-based processing provided by ForkGraph would be slow because there are significantly more graph operations to handle during the processing of a graph data with more partitions than those with fewer partitions.

Second, we can see that selecting the wrong partition size could result in a severe performance drop. For example, when dividing the `Us` graph into 256 partitions, the system could consume nearly 2× time than the case where the performance is almost as good as ForkGraph. This is the major motivation to have the optimizer in KGraph. It does not guarantee the configurations of the best performance, but it helps the users to avoid those poor cases.

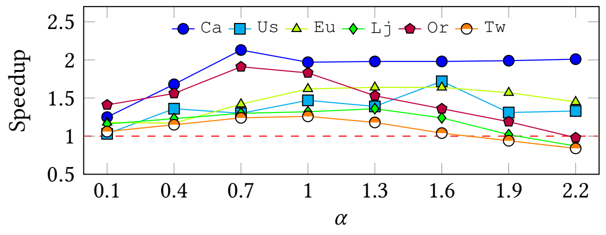


Figure 13: Speedups on Handling PPRs With Different Numbers of Pivots Selected.

6.5 Effects of Pivot Selection

Finding (3) : The optimizer picks the effective configuration for pivot selection in KGraph.

There are two major configurations in the pivot selection of KGraph. One is the number of pivot queries selected and the other one is the strategies of pivot selection. In this part, we first evaluate the effects of performance by adjusting the number of pivot queries while keeping the same strategies used in Table 2. We then study the performance of KGraph with different pivot selection strategies while fixing the number of pivot queries.

Number of pivots queries. We use concurrent PPR queries as an example to discuss the effects of the number of pivot queries selected. We use α to denote the normalized number of pivots. For example, when $\alpha = 1$, we use the same setting as the experiments in Table 2; when $\alpha = 0.1$, we use 10% of the pivots. Figure 13 presents the speedups of KGraph over ForkGraph with different numbers of pivots selected.

We can make several insights from the figure. First, the query processing on road networks benefits greatly from memoization, which is also sensitive to configurations as they can be influenced a

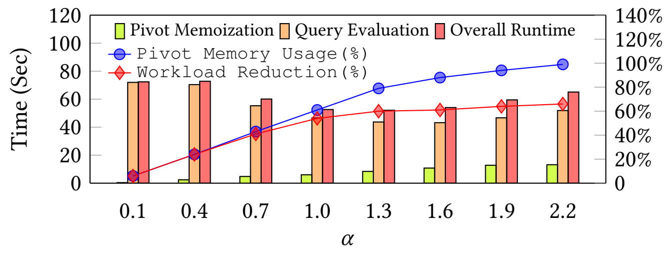


Figure 14: Runtime Breakdown, Additional Memory Usage and Workload Reduction for Evaluating PPRs on Eu by Varying the Number of Pivots.

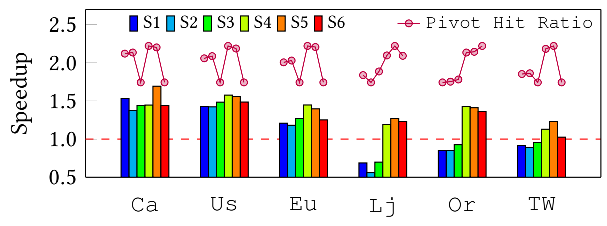


Figure 15: Speedups and Normalized Pivot Hit Ratio in Evaluating Deepwalk Using Different Pivot Selection Strategies.

lot compared to the processing on social networks. Basically, selecting all the boundary vertices as pivots ($\alpha = 1$) significantly reduces the intra-partition traversal, which thus exhibits the efficiency compared to other settings.

Second, we observe that not all configurations bring positive performance improvements over ForkGraph. For example, we can see that when we only select 10% of the boundary vertices (the leftmost dot), the performance of processing is poor. The reasons are twofold: 1) having fewer pivot queries means there are fewer chances to hit a pivot vertex, and 2) in order to improve memory efficiency, KGraph operates on a graph with smaller partitions. However, this is not favorable for partition-based processing due to the high scheduling overhead.

Third, memoizing more pivot queries can bring higher overheads. This observation is valid on both road and social networks. The major reason is that the benefits of performing more memoization are lower than the overhead it brings. For example, once we have already selected all the boundary vertices as pivots, getting more other pivots would be redundant; when processing queries on social networks, the benefits brought by the top high-degree vertices are much better than the other high-degree vertices at the second tier. Therefore, the benefits could be marginal. We can see that although the optimizer may not provide configurations to achieve the best-case performance, the configurations are usually sufficiently good in practice.

Microbenchmark. Our findings can be further corroborated through a detailed microbenchmark, as shown in Figure 14. This figure breaks down the overall running time into memoization of pivot queries and evaluation of concurrent queries. It also details additional memory overhead (a ratio to graph size) for pivot query memoization and the workload reduction ratio (in the PPR case, calculated as the ratio of the total saved walk length upon hitting a

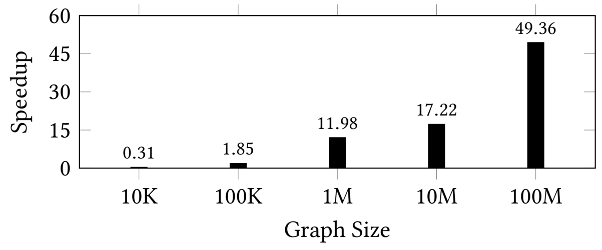


Figure 16: The speedups of KGraph over the hand-tuned baseline on five synthetic graphs generated using R-MAT.

pivot to the total required walk length). The figure illustrates that with a small number of selected pivots ($\alpha < 1$), both the runtime for memoizing pivots and the saved walk length increase linearly with α. However, for larger α values ($\alpha > 1$), the lower degree of newly memoized pivots leads to a smaller increase in memory usage. Nevertheless, the benefits are negated due to the introduction of more redundant pivots and a higher LLC miss ratio.

Pivots selection strategies. Figure 15 presents the speedups of KGraph over ForkGraph and the normalized pivot hit ratio using different strategies to select pivot queries. To avoid the case where both $S3$ and $S6$ select all the boundary vertices on road networks, we keep the numbers of the pivot queries in these cases at 80% of the settings in Table 2. Thus, we can make a fair comparison of the differences between the randomly selected boundary pivots and the boundary pivots with high degrees. We also find some interesting findings from the results.

First, selecting high-degree vertices ($S4$ and $S5$) always shows a much better performance than other strategies. This can be observed especially on social networks which have a power-law degree distribution. The reason is that the chances of paths and walks visiting high-degree vertices are much higher than other vertices. Second, selecting boundary vertices ($S3$ and $S6$) demonstrates better performance compared to randomly selecting pivots. This can be observed in social networks where $S3$ and $S6$ have the second highest hit ratios compared to $S4$ and $S5$. The hit ratios on road networks are relatively low. This is because road networks usually have large graph diameters, and in PPR, the walk may teleport back to the source vertex, making it rare for the walk to cross partitions and visit boundary vertices.

6.6 Performance on Synthetic Graphs

Finding (4) : KGraph shows a better scalability on large graphs than the hand-tuned baseline on synthetic graphs.

In this part, we use KGraph to run on the five synthetic graphs, which are generated by R-MAT, used in Section 3. We compare the performance of KGraph over the hand-tuned baseline implementation, which directly applies memoization using the 32 randomly selected queries. The hand-tuned implementation embarrassingly parallelizes the Dijkstra's algorithm [11] to solve SSSP queries.

Figure 16 shows the speedups of KGraph over the hand-tuned baseline. We can have the following observations. First, KGraph outperforms the hand-tuned approach significantly on large-scale graphs and there is a trend that KGraph achieves higher speedups on larger graphs. Second, as the partition-based approach brings extra overhead and the memoization on the small-scale graph is

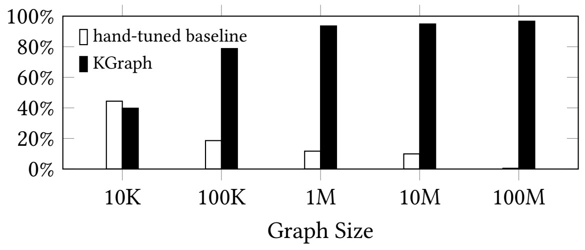


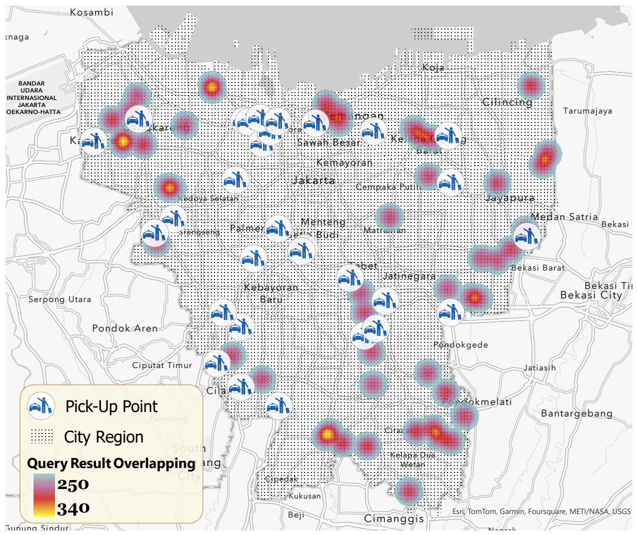
Figure 17: The operation reduction ratios of KGraph and the hand-tuned baseline on five synthetic graphs generated using R-MAT.

remarkable, KGraph takes around 3× more time to solve 512 queries. Nevertheless, both implementations solve 512 SSSP queries within 35ms on the smallest graph.

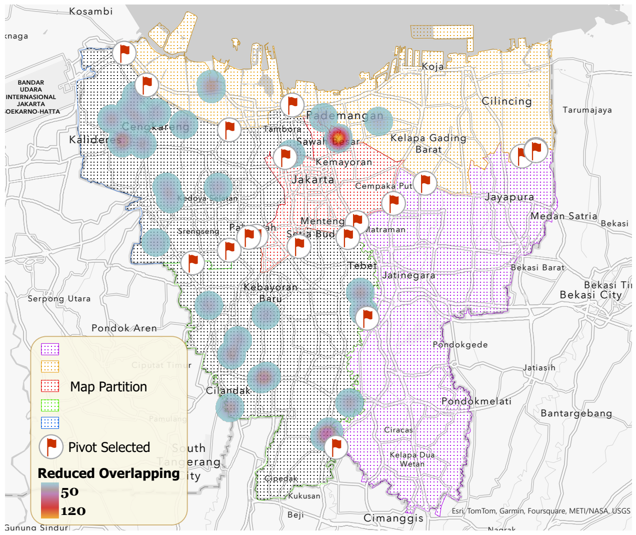
We use the microbenchmark of operation reduction in Section 3 to show the effectiveness of KGraph compared to the hand-tuned baseline. The operation reduction ratios of KGraph are calculated by comparing the numbers of operations between the executions when memoization is enabled and disabled. Figure 17 shows the overall operation reduction ratios of them for solving 512 SSSP queries. We can see from the figure that KGraph shows high reduction ratios (> 95%) on large-scale graphs.

6.7 Case Study

In our case study, we applied KGraph to manage concurrent ABC taxi hailing in Jakarta, showcasing its memoization effectiveness and efficiency. We utilized a detailed map of Jakarta from OpenStreetMap with 3.9 million vertices and 4.2 million edges. Our evaluation focused on SSSP for 113 queries during the first five seconds at 12 PM on June 10, 2021, executed via a sequential algorithm. Figure 18a illustrates the distribution of pick-up points (displayed partially for clarity) and highlights the significant overlap in evaluation results. Here overlaps pertain to the state updates in SSSP, leading to considerable lag (6519ms) in hailing queries, which markedly impacts user experience. Applying KGraph for this case significantly alleviated these issues. As depicted in Figure 18b (where pick-up points are omitted as they do not affect pivot selection), KGraph partitioned the Jakarta map into five sections, aligning closely with the city's five administrative regions. Utilizing the decision tree, KGraph adopted Strategy 6 and memoized all 1,944 boundary vertices as pivots. Each pivot store the shortest distances and paths to a number of nearby locations within its corresponding partition, which can be reused for hailing queries. This approach drastically minimized overlap during query evaluations, reducing the total runtime to 725ms (8ms for memoization), which is a duration well within the acceptable range for real-time taxi hailing services.



(a) Concurrent ABC Taxi Hailing in Jakarta at 12 PM on June 10, 2021.



(b) Significant Overlap Reduction in Query Evaluation with KGraph.

Figure 18: Comparative Analysis of Taxi Hailing Patterns and Query Overlap Reduction in Jakarta with KGraph.

7 RELATED WORK

CGQ processing has been used to solve a wide range of graph applications, including BC [14, 39], PLL [1], DeepWalk [32], Node2Vec [16], ACO [12], and many others. With the need to handle CGQs efficiently, we have witnessed a few works proposed recently. Hauck et al. [17] conduct experimental studies on executing CGQs. They study the inter- and intra- parallelism of handling diverse types of graph queries by assigning different threads to different instances of Galios [28], a graph processing framework. Those systems have overlooked the sharing opportunities among CGQs.

Researchers have presented more specific graph processing frameworks and systems for handling CGQs [27, 35, 40, 45]. Moreover, most of them tend to leverage sharing opportunities among different queries. Most systems fail to exploit the memorization opportunities among CGQs.

Graph Data Sharing. Zhang et al. [48] propose a disk-based graph processing system called CGraph to execute multiple and heterogeneous queries simultaneously. Zhao et al. [51] propose GraphM, a storage system that efficiently handles consolidated, out-of-core CGQs. Both CGraph and GraphM divide the graph into partitions and prioritize the partition to process with the most pending graph operations to maximize the utilization ratio of each partition. As introduced previously, ForkGraph also leverages graph data sharing. Different from disk-based graph processing systems, ForkGraph maximizes the graph data sharing at the cache level. When ForkGraphs handles the graph operations in an LLC-sized graph partition, all the accesses are expected to be limited within the cache.

Hardware Resources Sharing. Sun et al. [35] propose ThunderRW, a random-walk engine. The authors present a novel step-interleaving technique that breaks down the random walk operation into multiple stages. Pan et al. [30, 31] propose to collect the memory bandwidth consumption and atomic operations characteristics of graph queries by profiling them offline and then schedule the usage of computation resources to reduce the hardware blocking caused by heavy resources contention during the processing of CGQs.

Computation Sharing. Then et al. [38] proposes MS-BFS to accelerate concurrent BFS queries using multi-core CPUs. The authors observe the sharing opportunities among the explorations of activated vertices when handling graphs with the small-world property. The MS-BFS shares common computation across queries and mitigates the random memory accesses. Similarly, Liu et al. [26] propose iBFS to efficiently handle concurrent BFS queries using GPUs. Instead of visiting vertices individually for each BFS, iBFS leverages a joint frontier queue and bitwise operations for better GPU memory accesses. Xu et al. [44] propose SimGQ adopting the similar idea of finding the shareable subqueries during the processing of concurrent SSSP-like queries. SimGQ Unfortunately, SimGQ specifically serve only BFS or SSSP-like queries, losing the generalities of supporting different algorithms.

8 CONCLUSIONS

Concurrent graph query processing is fundamental for a wide range of graph applications. In this paper, we propose KGraph, a novel graph processing memoization engine to efficiently handle CGQs on large graphs. KGraph is proposed based on our observation that the results of queries have high overlaps among them. Thus, KGraph first proposes a fine-grained memoization with adaptive graph partitioning to reduce memoization overhead. Second, KGraph uses a decision tree model to strategically select pivot queries for high memoization effectiveness. Our evaluations of the system using real-world graphs show that KGraph significantly outperforms state-of-the-art concurrent graph query processing systems.

REFERENCES

[1] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD (2013)*. 349–360.

[2] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 2011. 10th DIMACS Implementation Challenge-Graph Partitioning and Graph Clustering.

[3] Laurynas Biveinis, Simonas Šaltenis, and Christian S Jensen. 2007. Main-memory operation buffering for efficient R-tree update. In *Proceedings of the 33rd international conference on Very large data bases*. 591–602.

[4] Markus Bläser. 2003. A new approximation algorithm for the asymmetric TSP with triangle inequality. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. 638–645.

[5] Austin Buchanan, Yiming Wang, and Sergiy Butenko. 2018. Algorithms for node-weighted Steiner tree and maximum-weight connected subgraph. *Networks* 72, 2 (2018), 238–248.

[6] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[7] Hongzheng Chen, Minghua Shen, Nong Xiao, and Yutong Lu. 2021. Krill: a compiler and runtime system for concurrent graph processing. In *SC*. ACM, 51.

[8] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. Introduction to algorithms. (2022).

[9] Camil Demetrescu, Andrew V Goldberg, and David Johnson. 2008. 9th DIMACS implementation challenge–Shortest Paths (2006). (2008).

[10] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *SPAA (2017)*. 293–304.

[11] Edsger W Dijkstra et al. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[12] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. 2006. Ant colony optimization. *IEEE computational intelligence magazine* 1, 4 (2006), 28–39.

[13] Giorgio Gallo and Stefano Pallottino. 1988. Shortest path algorithms. *Annals of operations research* 13, 1 (1988), 1–79.

[14] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. 2020. Traversing large graphs on GPUs with unified memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1119–1133.

[15] Ronald L Graham and Pavol Hell. 1985. On the history of the minimum spanning tree problem. *Annals of the History of Computing* 7, 1 (1985), 43–57.

[16] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *SIGKDD (2016)*. 855–864.

[17] Matthias Hauck, Marcus Paradies, and Holger Fröning. 2017. Can Modern Graph Processing Engines Run Concurrent Queries Efficiently?. In *GRADES (2017)*. 1–6.

[18] Fuad Jamour, Spiros Skiadopoulos, and Panos Kalnis. 2017. Parallel algorithm for incremental betweenness centrality on large graphs. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2017), 659–672.

[19] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: generalized incremental graph processing via graph triangle inequality. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 17–32.

[20] George Karypis and Vipin Kumar. 1998. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing* 48, 1 (1998), 96–129.

[21] Farzad Khalvati, Mark D Aagaard, and Hamid R Tizhoosh. 2015. Window memoization: toward high-performance image processing software. *Journal of Real-Time Image Processing* 10, 1 (2015), 5–25.

[22] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW (2010)*. 591–600.

[23] Jüri Lember, Dario Gasbarra, Alexey Koloydenko, and Kristi Kuljus. 2019. Estimation of Viterbi path in Bayesian hidden Markov models. *Metron* 77 (2019), 137–169.

[24] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[25] Zhen Lin and Yijun Bei. 2014. Graph indexing for large networks: A neighborhood tree-based approach. *Knowledge-Based Systems* 72 (2014), 48–59.

[26] Hang Liu, H Howie Huang, and Yang Hu. 2016. ibfs: Concurrent breadth-first search on gpus. In *SIGMOD (2016)*. ACM, 403–416.

[27] Shengliang Lu, Shixuan Sun, Johns Paul, Yuchen Li, and Bingsheng He. 2021. Cache-Efficient Fork-Processing Patterns on Large Graphs. arXiv:2103.14915 [cs.DB]

[28] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP (2013)*. 456–471.

[29] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[30] Peitian Pan and Chao Li. 2017. Congra: Towards Efficient Processing of Concurrent Graph Queries on Shared-Memory Machines. In *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 217–224.

[31] Peitian Pan, Chao Li, and Minyi Guo. 2019. CongraPlus: Towards Efficient Processing of Concurrent Graph Queries on NUMA Machines. *IEEE Transactions on Parallel and Distributed Systems* (2019).

[32] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *SIGKDD (2014)*. 701–710.

[33] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.

[34] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W Mahoney. 2016. Parallel Local Graph Clustering. *Proceedings of the VLDB Endowment* 9, 12 (2016).

[35] Shixuan Sun, Yuhang Chen, Shengliang Lu, Bingsheng He, and Yuchen Li. 2021. ThunderRW: An In-Memory Graph Random Walk Engine (Complete Version). *arXiv preprint arXiv:2107.11983* (2021).

[36] Shixuan Sun and Qiong Luo. 2020. Subgraph matching with effective matching order and indexing. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2020), 491–505.

[37] Hongshi Tan, Xinyu Chen, Yao Chen, Bingsheng He, and Weng-Fai Wong. 2023. LightRW: FPGA Accelerated Graph Dynamic Random Walks. *Proc. ACM Manag. Data* 1, 1, Article 90 (may 2023), 27 pages. https://doi.org/10.1145/3588944

[38] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T Vo. 2014. The more the merrier: Efficient multi-source graph traversal. *Proceedings of the VLDB Endowment* 8, 4 (2014), 449–460.

[39] Joseph Wang and D Eppstein. 2001. Fast approximation of centrality. In *SODA (2001)*. 228–229.

[40] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. 2020. {GraphWalker}: An {I/O-Efficient} and {Resource-Friendly} Graph Analytic System for Fast and Scalable Random Walks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 559–571.

[41] Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. 2018. Fast and concurrent {RDF} queries using {RDMA-assisted}{GPU} graph exploration. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 651–664.

[42] Emo Welzl. 1988. Partition trees for triangle counting and other range searching problems. In *Proceedings of the fourth annual symposium on Computational geometry*. 23–33.

[43] Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. 2021. MD-HM: memoization-based molecular dynamics simulations on big memory system. In *Proceedings of the ACM International Conference on Supercomputing*. 215–226.

[44] Chengshuo Xu, Abbas Mazloumi, Xiaolin Jiang, and Rajiv Gupta. 2020. SimGQ: Simultaneously evaluating iterative graph queries. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 1–10.

[45] Ke Yang, Xiaosong Ma, Saravanan Thirumuruganathan, Kang Chen, and Yongwei Wu. 2021. Random Walks on Huge Graphs at Cache Efficiency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 311–326.

[46] Jin Y Yen. 1970. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quart. Appl. Math.* 27, 4 (1970), 526–530.

[47] Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2023. Glign: Taming Misaligned Graph Traversals in Concurrent Graph Processing. In *ASPLOS (1)*. ACM, 78–92.

[48] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. 2018. CGraph: A correlations-aware approach for efficient concurrent iterative graph processing. In *ATC (2018)*. 441–452.

[49] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *OOPSLA (2018)* 2 (2018), 121.

[50] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. 2019. GraphM: an efficient storage system for high throughput of concurrent graph processing. In *SC*. ACM, 3:1–3:14.

[51] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. 2019. GraphM: an efficient storage system for high throughput of concurrent graph processing. In *SC (2019)*. ACM, 3.

[52] Jingren Zhou and Kenneth A Ross. 2003. Buffering accesses to memory-resident index structures. *Proceedings of the VLDB Endowment* (2003).

[53] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *OSDI (2016)*. 301–316.