An Efficient Memoization Engine for Concurrent Graph Query Processing

Sen Gao[†], Shengliang Lu[†], Shixuan Sun[‡], Yuchen Li[§], Bingsheng He[†]
[†]*National University of Singapore*, Singapore
[‡]*Shanghai Jiao Tong University*, Shanghai, China
[§]*Singapore Management University*, Singapore
sen@u.nus.edu; lusl@nus.edu.sg; sunshixuan@sjtu.edu.cn; yuchenli@smu.edu.sg; hebs@comp.nus.edu.sg

2025 IEEE 41st International Conference on Data Engineering (ICDE) | 979-8-3315-3603-9/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICDE65448.2025.00081

***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 $4.2\times$ over the state-of-the-art CGQ systems.**

***Index Terms*—memoization, concurrent graph query, query processing**

I. 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 [35], Personalized PageRank (PPR) [37], and Ant Colony Optimization (ACO) [13]. Separately, in a more practical context, services like the Grab 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. [2] 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 [40]. This has spurred increasing interest in enhancing CGQ processing efficiency [31], [38], [43], with a primary focus on exploiting the *sharing* opportunities among CGQs. The state-of-the-art, ForkGraph [31], 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.

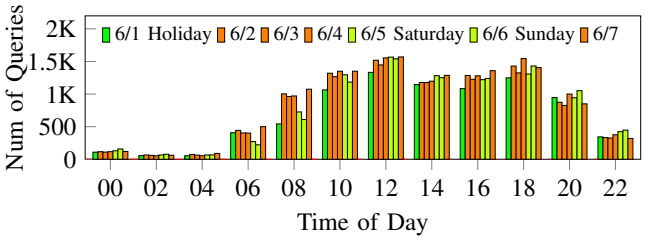


Fig. 1: Hourly distribution of taxi hailing in Jakarta during the first week of June 2021 (average number per minute).

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 [8] 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 Grab, the Grab 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 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

2375-026X/25/$31.00 ©2025 IEEE
DOI 10.1109/ICDE65448.2025.00081

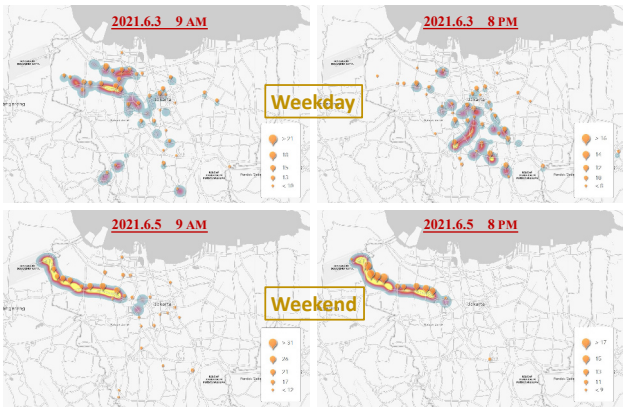


Fig. 2: Top intersections visited by Grab taxis in the first ten seconds during morning and evening rush in Jakarta.

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 [5], [23].

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 $2.3\times$ 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 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 [31]. We evaluate KGraph using the five applications mentioned above as well as a case study with Grab real workloads. The experimental results show that KGraph achieves up to $12.5\times$ speedup with an average speedup at $4.1\times$ over ForkGraph.

II. Background

A. Preliminaries

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

Concurrent graph queries (CGQs). CGQs are essential in many graph applications such as Betweenness Centrality(BC) [16], [42], PLL [2], Ant Colony Optimization(ACO) [13], single-source k-shortest path [49], and machine learning on graph using random walks [19], [35]. These applications launch finite independent queries from different source vertices on the same graph [2], [13], [19], [22], [35], [49]. 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 V.

B. 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 [4], [31], [55]. 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 [10], [25], [46]. 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 SSSP or a specific

[1]The source code is available at *https://github.com/Gawssin/KGraph*.

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 during the query execution, typically measured as the total number of vertex updates.

Definition 2.1: A graph query is considered **memoizable** 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 Triangle Counting [45] which do not start from specific vertices.
2) *Overlap Computation*: For any two memoizable 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. As far as we know, there are a significant number of concurrent graph query problems that have proven memoizable and can generally be categorized into three types:

- **Queries satisfying the graph triangle inequality** [23]. For any three vertices u, v, and w, the graph triangle inequality is expressed as $\texttt{property}(u, v) \oplus \texttt{property}(v, w) \succeq \texttt{property}(u, w)$, 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 [27], and Radii estimation [36].
- **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) [18] and Steiner Tree [7].
- **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 [29], [39].

C. Examples of Memoizable Queries

Here we discuss two examples of memoizable queries.

Minimum Spanning Tree (MST). In the MST problem, the objective is to determine a subset of edges that connects all vertices in the graph, ensuring there are no cycles, while minimizing the total weight of the edges. When multiple MST queries are initiated concurrently from different parts of the graph, there is a high likelihood of overlapping subgraphs. For instance, consider two MST queries q_1 and q_2 starting from different vertices but covering overlapping regions U_1 and U_2 of the graph. The application of memoization can be perfomed as follow:

- Memoizing Subtrees: As each MST query progresses, it computes optimal subtrees for subsets of vertices. By storing these subtrees in the memoization storage, subsequent queries that require these subtrees can retrieve them directly, avoiding redundant computations.
- Reducing Workload: The workloads W_1 and W_2 for queries q_1 and q_2 can be significantly reduced by reusing memoized subtrees corresponding to $U_1 \cap U_2$.
- Dynamic Programming Integration: Since MST algorithms like Kruskal's and Prim's inherently use greedy strategies that build upon optimal substructures, memoization aligns well with their computation models.

Subgraph Matching. Subgraph matching involves finding all occurrences of a query graph within a larger target graph. Algorithms like TurboISO [20] utilize indices to accelerate the matching process. The application of memoization is as follow:

- Index Memoization: When processing subgraph matching queries, indices such as adjacency lists, degree sequences, or more complex neighborhood signatures are computed. By memoizing these indices, subsequent queries that require the same or similar indices can reuse them.
- Filtering Candidates: Memoized indices can help quickly eliminate vertices that cannot be part of the match, reducing the search space and computational overhead.
- Concurrent Queries: In scenarios where multiple subgraph matching queries are executed concurrently, memoization ensures that common computations are not redundantly performed.

III. 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 [31], [47]. 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 [31], [47], 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 $\texttt{dist}(u_q, u_{q'}) + \texttt{dist}(u_{q'}, u) < \texttt{dist}(u_q, u)$ (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, while the overhead lies in scanning the result of q', whose size 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 Grab 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 [8], 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.

A. Profiling Results

Memoization speedup. We set $|Q|$ (i.e., the number of CGQs) to 1000 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.

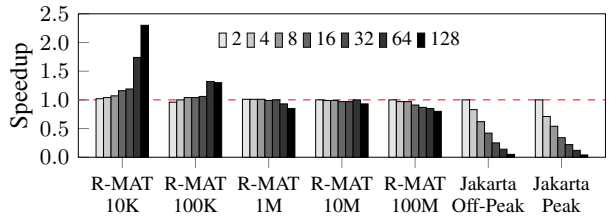


Fig. 3: The speedup of the memoization method with the number of memoized queries varying from 2 to 128.

[2]https://www.openstreetmap.org/

[3]$a = 0.5, b = c = 0.1$

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 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.

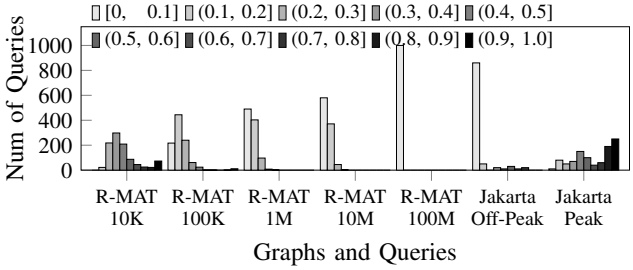


Fig. 4: The operation reduction ratio distribution of 1000 CGQs with 32 memorized queries.

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%.

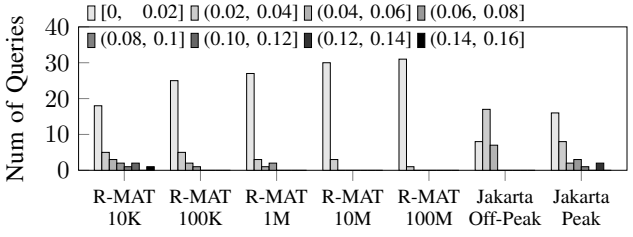


Fig. 5: The total operation reduction ratio distribution of 1000 concurrent queries with 32 memorized queries.

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* [14], 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 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.

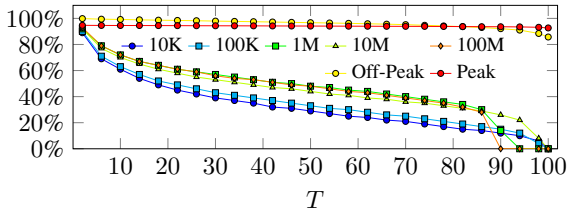


Fig. 6: The percentage of the number of edges in x that belong to at least t distinct shortest path trees.

B. 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.

IV. Overview of KGraph

As shown in Section III, 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 V-A), which imitates the memoization on small-scale graphs to reduce the overhead of scanning the memoized results. The second optimization is *on effective pivot query selection* (Section V-B), 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 V.

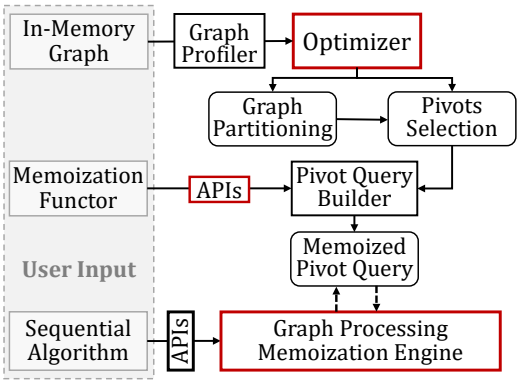


Fig. 7: Overview of KGraph.

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 [31] to ease the programming for CGQ processing. There are two key API functions, InitMemoizationStorage and MemoizationFunctor, that enable users to define the logic of the system. The InitMemoizationStorage(Pivots P, ...) function initializes memoization storage for each pivot in P, precomputing and storing reusable data for future computations. Note that within this function, the computation of distinct pivot queries can be naturally performed in parallel, as building pivot query results is independent for each pivot vertex. Meanwhile, MemoizationFunctor(Pivot p) leverages the memoized data associated with pivot p to efficiently update query results. Sample API implementations for SSSP-based and random walk-based applications are provided in Algorithm 2.

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 ∈ Q do
6:    processor.run(q)
7: procedure run(q)
8:    while q's activated vertices set F ≠ ∅ do
9:       for_each f ∈ F do
10:         if f ∈ Pivots then
11:            Perform MemoizationFunctor(f) to update q
12:         else
13:            Compute(f) to update q
```

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 automatically 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, average degree, 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.

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 5-6.

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.

Comparison with Landmark-Based Algorithms. Landmark-based algorithms are a class of methods used to estimate shortest path distances by precomputing distances from selected landmark nodes to all other nodes in the graph [17]. These algorithms are effective for answering individual shortest path queries in static graphs. Our memoization approach in KGraph differs in several key ways:1) Concurrent Query Optimization: KGraph is designed to optimize the processing of multiple concurrent queries by sharing computations, whereas landmark-based algorithms focus on individual queries. 2) Partition-Based Processing: KGraph partitions the graph to localize memoization, reducing overhead and improving scalability for large graphs. 3) Strategic Pivot Selection: We select pivot queries based on their potential to benefit other queries, enhancing the effectiveness of memoization compared to the fixed landmarks in traditional methods.

V. 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.

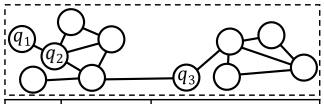
A. Partition-Based Memoization

As we observe in Section III, 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 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 III, performing memoization shows degraded performance on large graphs because the overhead of scanning the results offsets the benefits.

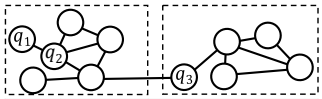
We use an example of three SSSP queries on a graph with 11 vertices in Figure 8 as an example to demonstrate the benefits brought (operations reduced) and overhead involved. In the example, the source vertices of q_1 and q_2 are close, and the source vertex of q_3 is a bit far from them, located near 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 [23] by simply adding a



Query	Memoized	Memoization Operations	
		Useful	Wasted
q_1	q_2	9	1
q_1	q_3	4	6
q_2	q_1	0	10
q_2	q_3	4	6
q_3	q_1	0	10
q_3	q_2	2	8

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



Query	Memoized	Memoization Operations	
		Useful	Wasted
q_1	q_2, q_3	8	1
q_2	q_1, q_3	4	5
q_3	q_1, q_2	2	7

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

unit weight to the results of q_2. Specifically, given the shortest path and the distance from q_2 to any other vertex, denoted as $\text{dis}_{q_2}(v)$, the shortest path starting from q_1 can be computed as: $\texttt{dist}(q_1, v) = \texttt{dist}(q_2, v) + 1 \quad \text{if} \quad v \neq q_1$, which implies that only the distance $\texttt{dist}(q_2, q_1)$ is wasted. 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 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 10 and 6 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 5 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 [31], 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:

$$\frac{|E| + |V|}{|P|} + \frac{k}{|P|} \times \mathsf{Mem_{max}} \leq \mathsf{Size_{LLC}}$$

where $(|E| + |V|)/|P|$ represents the size of the partitioned raw graph data, such as memory for CSR storage in a partition. The term $k/|P| \times \mathsf{Mem_{max}}$ 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

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

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 $\mathsf{Scheduling\_Overhead}$ includes the

costs associated with inter-partition processing, such as coordination and status updates across partitions. In the worst case, these costs can be bounded by W_q, which represents the total workload involved in inter-partition processing. The factor $\mathsf{Pivot\_Hits}$, defined as $\mathsf{Hit\_Ratio} \times |U_q|$, quantifies the number of times a pivot query is utilized, where $\mathsf{Hit\_Ratio}$ represents the likelihood of hitting a pivot. This ratio is influenced by the number and in-degrees of selected pivot vertices, formally given by $\mathsf{Hit\_Ratio} = \frac{\sum_{h \in H} \mathsf{deg_{in}}(x_h)}{\sum_{v \in V} \mathsf{deg_{in}}(v)} = \frac{\sum_{h \in H} \mathsf{deg_{in}}(x_h)}{|E|}$, since a higher in-degree increases the likelihood that a pivot will be hit during query evaluation. Here, $\mathsf{deg_{in}}(x_h)$ is the in-degree of pivot h. The $\mathsf{Work\_Reduction_{avg}} = W_{\mathsf{OpCut}} - W_{\mathsf{LookUp}}$ 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, $\mathsf{Mem_{max}}$, 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, one method to memoize data is to memoize 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 $\mathsf{Mem_{avg}}$ 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.

B. 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 [44]. 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 III, 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 incoming 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 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

TABLE I: Input graphs ($\overline{d}$: average degree).

Graph	Source	$\#V$	$\#E$	$\overline{d}$	Memory	$\lvert P\rvert$
Ca	California [11]	1.9M	4.6M	2.4	0.03GB	4
Us	USA [11]	23.9M	57.7M	2.4	0.39GB	7
Eu	Europe [3]	50.9M	108.1M	2.1	0.78GB	9
Lj	LiveJournal [28]	4.8M	87.5M	18.0	0.36GB	11
Or	Orkut [28]	3.0M	117.1M	38.1	0.46GB	12
Tw	Twitter [26]	61.6M	1.5B	23.8	5.93GB	64
Fr	Friendster [28]	124.8M	1.8B	14.5	7.66GB	72
Uk	UK-Union [6]	133.6M	5.5B	41.0	21.39GB	128

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 automatically 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 $10\%-50\%$ 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.

VI. Evaluation

A. 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. All implementations are compiled using `g++ 7.5.0` with the `-O3` optimization flag and OpenMP enabled.

Comparisons. Our study involves a comparative analysis of KGraph against ForkGraph [31] and Glign [50]. ForkGraph is the state-of-the-art graph processing systems for handling CGQs, and it has been previously shown in [31] to significantly outshine counterparts such as Ligra [36], Gemini [56], and GraphIt [52]. 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 [53] and Krill [9]. Since Glign does not support random walk-based applications, we have excluded it from the corresponding comparison.

Algorithm 2 Example Interface Implementation for Two Types of Applications

```
# SSSP
procedure InitMemoizationStorage(Pivots P, Memoized-Size k)
  parallel_for_each p ∈ P do
    D ← Find k nearest vertices from p within the partition of p
    for_each v_i ∈ D do
      M_p.push(< v_i, distance_p[v_i]>)
procedure MemoizationFunctor(Pivot p)
  for_each < v_i, distance_p[v_i]> ∈ M_p do
    distance[v_i] ← min(distance_p[v_i] + distance[p], distance[v_i])

# RandomWalk
procedure InitMemoizationStorage(Pivots P, Length l)
  parallel_for_each p ∈ P do
    M_p ← 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 ← walkLength − l
```

We measure the time required for each system to complete the processing of CGQs. The measurement excludes the time spent on partitioning the graph or reading data from the disk, as the focus is on optimizing in-memory computation. Note that the time spent on memoization is included for KGraph, as it is an integral part of the runtime execution.

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. To better illustrate the APIs, we use the implementation for random walks as an example. The InitMemoizationStorage function takes pivot vertices and an integer l as inputs. For each pivot, it computes and caches all paths starting from that pivot with lengths up to l, along with their corresponding probabilities. The MemoizationFunctor function then utilizes these cached paths by performing weighted sampling based on the stored probabilities to select and return a path. This approach allows for avoiding numerous random number generations and cache misses that typically occur when accessing graph structures step-by-step during random walks.

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. For the synthetic graphs, we use R-MAT to generate

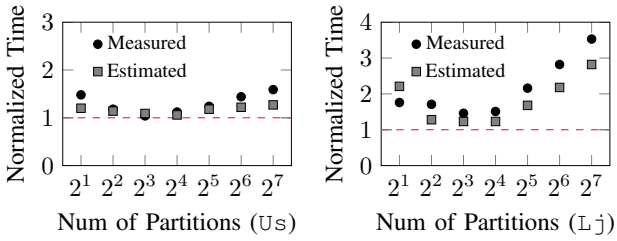


Fig. 10: The actual execution time and estimated time of solving concurrent SSSP queries using different partitions sizes. The results are all normalized to the shortest execution/estimated time.

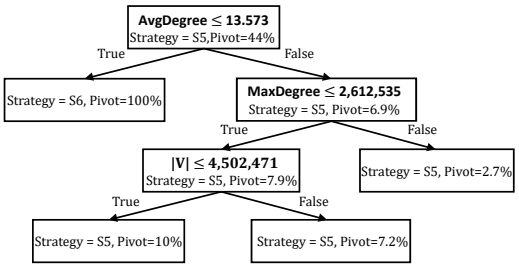


Fig. 11: The visualization of the hybrid decision tree trained in KGraph.

30 graphs of varying scales and densities, as detailed in the complete version [15]. Besides, we select 8 real-world graphs as listed in Table I. These datasets are publicly available, and they are widely used in the existing works [36], [52], [56] to benchmark algorithms and frameworks. Following the experimental methodologies in [36] and [12], weighted graphs are generated by assigning edge weights uniformly at random from the range $[1, \log|V|)$. The datasets are classified into three categories: `Ca`, `Us`, and `Eu` represent road networks; `Lj`, `Or`, `Tw`, and `Fr` denote social networks; and `Uk` corresponds to the web graph. By default, graph partitioning is conducted using METIS [24]. However, for `Tw` and `Uk`, chunk partitioning was employed because METIS ran out of memory during runtime.

Supplement Experiments. Due to space limitations, we have moved the implementation details (see Section VI-A), along with the experimental analyses of optimizer evaluation (Figures 10 and 11), graph partitioning effects (Figure 12), pivot selection effects (Figures 15, 16, and 17), and experiments on synthetic graphs to the complete version [15].

B. Optimizer Evaluation

The optimizer contains two components, the cost model for graph partitioning and the decision tree for pivot selection strategies. The results are shown in Figures 10 and 11.

C. 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 II. First, KGraph demonstrates superior performance in traversal-based queries such as SSSP, SSWP, and SSR, outshining ForkGraph in all cases with up to $12.5\times$ speedup and an average of $5.9\times$. 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 [50].

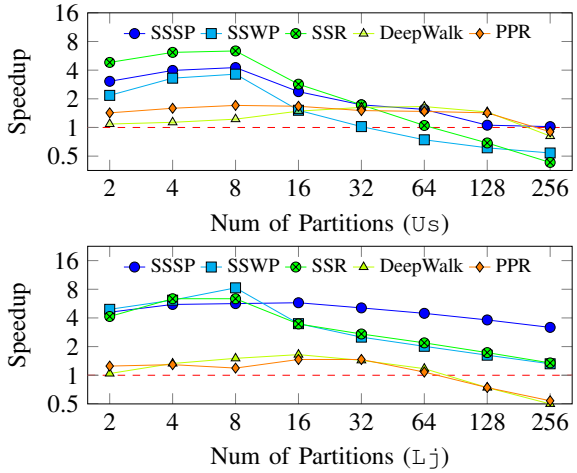


Fig. 12: Speedups of KGraph over Forkgraph using different partition sizes.

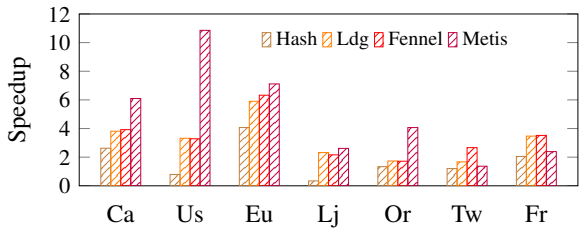


Fig. 13: Speedups using different partitioning methods.

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 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.

TABLE II: Overall speedups of KGraph over ForkGraph and Glign (OOT / OOM: Ran out of 3 hours / 256GB memory).

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×
Fr	1363.9 s	1.70×	9.35×	1819.1 s	2.21×	13.69×	479.9 s	1.21×	4.50×	88.1 s	1.41×	91.6 s	2.34×
Uk	OOT	OOM	1731.6 s	OOT	OOM	1929.8 s	6032.1 s	OOM	8.29×	125.0 s	2.75×	110.7 s	2.55×

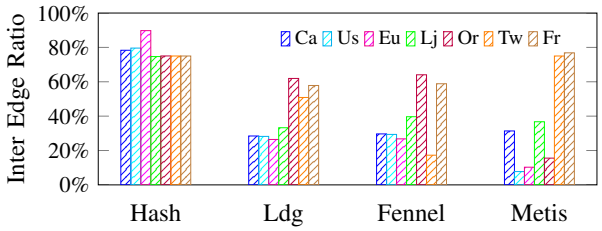


Fig. 14: Inter-partiton edge ratio (% of E).

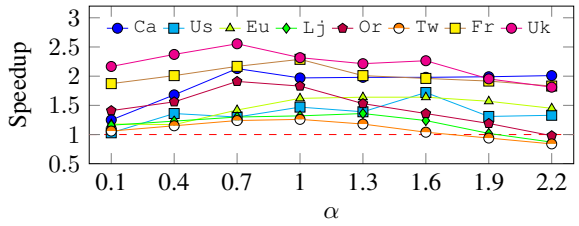


Fig. 15: Speedups on handling PPRs with different numbers of pivots selected.

D. 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 as shown in Figure 12, 13 and 14.

In this part, we analyze the effects of partitioning the graph into varied sizes and adopting different partition methods. We present the effects of different graph partitioning methods in Figure 13 and Figure 14. We ran SSWP on all graphs except Uk, as it exceeded the 3-hour runtime limit. The number of partitions was fixed at 8, and we compared four popular partitioning algorithms: Hash [1], LDG [1], Fennel [1], and METIS. The first three are streaming algorithms. As shown in Figure 13, METIS performs well on smaller graphs compared to streaming algorithms. However, for large graphs, since we use chunk partitioning to replace METIS due to its slower runtime, the chunk-based partitioning performs worse than the otherss, producing partitions with the highest number of inter-partition edges, as shown in Figure 14. Additionally, the hash method consistently shows the lowest speedups.

E. Effects of Pivot Selection

Finding (3) : The optimizer picks the effective configuration for pivot selection as shown in Figure 15, 16 and 17.

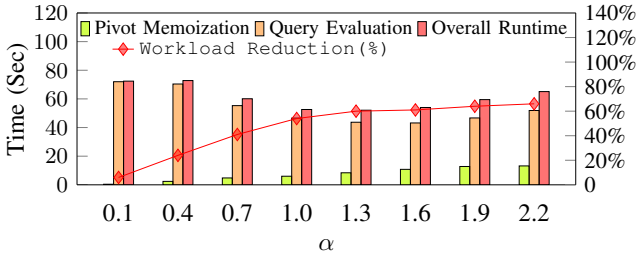


Fig. 16: Runtime breakdown. Additional memory usage and workload reduction for evaluating PPRs on Eu by varying the number of pivots.

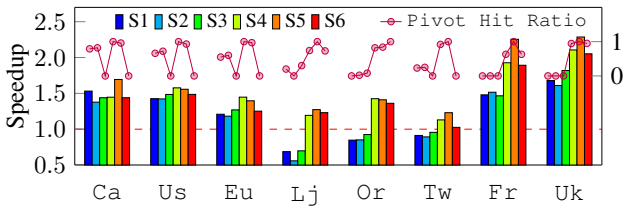
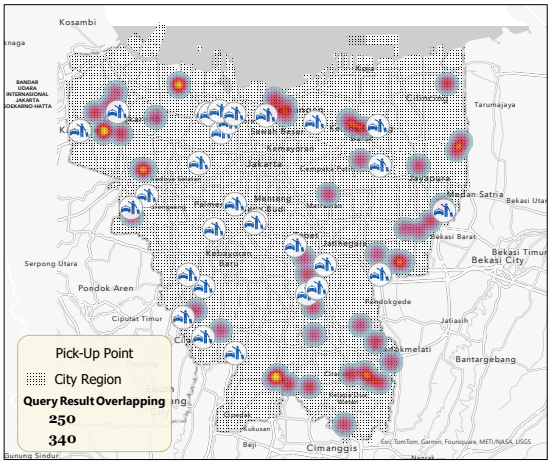


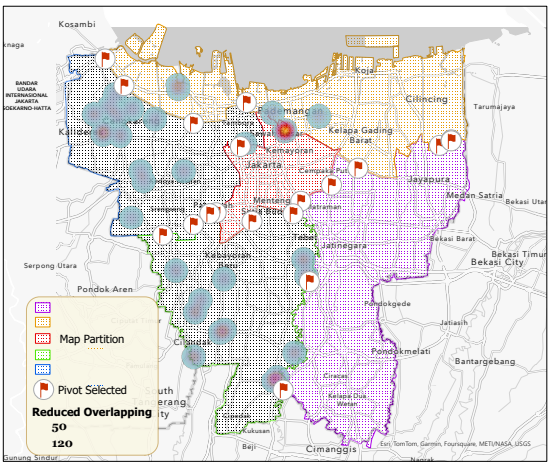
Fig. 17: Speedups and normalized pivot hit ratio in evaluating Deepwalk using different pivot selection strategies.

F. Case Study

In our case study, we applied KGraph to manage concurrent Grab 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



(a) Concurrent Grab taxi hailing in Jakarta at 12 PM on June 10, 2021.



(b) Significant overlap reduction in query evaluation with KGraph.

Fig. 18: Comparative analysis of taxi hailing patterns and query overlap reduction in Jakarta with KGraph.

(8ms for memoization), which is a duration well within the acceptable range for real-time taxi hailing services.

VII. Related Work

CGQ processing has been used to solve a wide range of graph applications, including BC [16], [42], PLL [2], DeepWalk [35], Node2Vec [19], ACO [13], and many others. With the need to handle CGQs efficiently, we have witnessed a few works proposed recently. Hauck et al. [21] perform experimental studies on executing CGQs. Their research explores the inter- and intra-parallelism in handling diverse types of graph queries by assigning different threads to separate instances of Galois [32], 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 [31], [38], [43], [48]. 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. [51] propose a disk-based graph processing system called CGraph to execute multiple and heterogeneous queries simultaneously. Zhao et al. [54] 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. [38] 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. [33], [34] profile graph queries offline, collecting their memory bandwidth consumption and atomic operation characteristics. Based on this profiling, they schedule the utilization of computational resources to mitigate hardware blocking caused by heavy resource contention during the processing of CGQs.

Computation Sharing. Then et al. [41] 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. [30] propose iBFS to efficiently process concurrent BFS queries using GPUs. Rather than visiting vertices individually for each BFS, iBFS employs a joint frontier queue and utilizes bitwise operations to optimize GPU memory access. Xu et al. [47] propose SimGQ adopting the similar idea of finding the shareable subqueries during the processing of concurrent queries. SimGQ Unfortunately, SimGQ specifically serve only BFS or SSSP-like queries, losing the generalities of supporting different algorithms.

VIII. 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 on real-world graphs demonstrate that KGraph achieves significant performance improvements over state-of-the-art concurrent graph query processing systems.

IX. Acknowledgment

This work is supported by National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG2-TC-2021-002) and the Ministry of Education AcRF Tier 2 grant, Singapore (No. MOE-T2EP20121-0016)

REFERENCES

[1] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. Streaming graph partitioning: An experimental study. *Proc. VLDB Endow.*, 11(11):1590–1603, 2018.

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

[3] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 10th dimacs implementation challenge-graph partitioning and graph clustering, 2011.

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

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

[6] 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.

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

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

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

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

[11] Camil Demetrescu, Andrew V Goldberg, and David Johnson. 9th dimacs implementation challenge–shortest paths (2006). 2008.

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

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

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

[15] Sen Gao, Shengliang Lu, Shixuan Sun, Yuchen Li, and Bingsheng He. An efficient memoization engine for concurrent graph query processing, 2025. https://github.com/Gawssin/KGraph/blob/main/KGraph-Complete-Version.pdf.

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

[17] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: *A* search meets graph theory. In *SODA*, pages 156–165. SIAM, 2005.

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

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

[20] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD Conference*, pages 337–348. ACM, 2013.

[21] Matthias Hauck, Marcus Paradies, and Holger Fröning. Can modern graph processing engines run concurrent queries efficiently? In *GRADES (2017)*, pages 1–6, 2017.

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

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

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

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

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

[27] Jüri Lember, Dario Gasbarra, Alexey Koloydenko, and Kristi Kuljus. Estimation of viterbi path in bayesian hidden markov models. *Metron*, 77:137–169, 2019.

[28] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

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

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

[31] Shengliang Lu, Shixuan Sun, Johns Paul, Yuchen Li, and Bingsheng He. Cache-efficient fork-processing patterns on large graphs, 2021.

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

[33] Peitian Pan and Chao Li. Congra: Towards efficient processing of concurrent graph queries on shared-memory machines. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 217–224. IEEE, 2017.

[34] Peitian Pan, Chao Li, and Minyi Guo. Congraplus: Towards efficient processing of concurrent graph queries on numa machines. *IEEE Transactions on Parallel and Distributed Systems*, 2019.

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

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

[37] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W Mahoney. Parallel local graph clustering. *Proceedings of the VLDB Endowment*, 9(12), 2016.

[38] Shixuan Sun, Yuhang Chen, Shengliang Lu, Bingsheng He, and Yuchen Li. Thunderrw: An in-memory graph random walk engine (complete version). *arXiv preprint arXiv:2107.11983*, 2021.

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

[40] Hongshi Tan, Xinyu Chen, Yao Chen, Bingsheng He, and Weng-Fai Wong. Lightrw: Fpga accelerated graph dynamic random walks. *Proc. ACM Manag. Data*, 1(1), may 2023.

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

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

[43] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. 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)*, pages 559–571, 2020.

[44] Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. Fast and concurrent rdf queries using rdma-assisted gpu graph exploration. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 651–664, 2018.

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

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

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

[48] Ke Yang, Xiaosong Ma, Saravanan Thirumuruganathan, Kang Chen, and Yongwei Wu. Random walks on huge graphs at cache efficiency. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 311–326, 2021.

[49] Jin Y Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of Applied Mathematics*, 27(4):526–530, 1970.

[50] Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. Glign: Taming misaligned graph traversals in concurrent graph processing. In *ASPLOS (1)*, pages 78–92. ACM, 2023.

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

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

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

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

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

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