Scaling Asynchronous Graph Query Processing via Partitioned Stateful Traversal Machines

Shaoyuan Chen*, Hongtao Chen*, Shaonan Ma†, Yajie Qin†, Zheng Wang*, Weiyu Xie*, Mingxing Zhang*†, Kang Chen*, Xia Liao*, Yingdi Shan*, Jinlei Jiang*†, Yongwei Wu*†
*Tsinghua University, †9#AISoft

***Abstract*—Due to the escalating demand to analyze large graphs, many organizations are now collecting billion-level property graph datasets, concurrently executing many complex graph queries against them, and expecting interactive-level response latency. However, such requirements are particularly challenging because of the notoriously irregular data access pattern and complex dependencies between heterogeneous subtasks. Despite the widespread availability of many-core CPUs and high-speed networking in modern datacenters, existing distributed graph query systems struggle with their inherent inefficiencies, resulting in low hardware utilization and poor query performance on these state-of-the-art hardware. To address these challenges, we introduce the Partitioned Stateful Traversal Machine (PSTM), which extends the Gremlin graph traversal machine. PSTM retains the expressive power of the Gremlin query language, enabling it to accommodate a wide range of graph query tasks, including traversal, pattern matching, filtering, and result aggregation. It additionally introduces query memoranda, allowing for more efficient implementation and execution of numerous graph queries in distributed environments. Moreover, PSTM facilitates various system-level optimizations, such as massively parallel execution, overlapping computation with communication, locality-aware data access, and lightweight progress tracking. Building upon PSTM, we develop GraphDance, a distributed graph database featuring an efficient asynchronous PSTM runtime. Our evaluations, conducted on an 8-node cluster, show that GraphDance achieves millisecond-level query latency for complex queries on terabyte-scale graphs, with an average latency reduction of 89.2% across all interactive complex queries in the LDBC SNB benchmark compared to existing distributed graph query systems.**

I. Introduction

A. Motivation

As the scale of graph datasets continues to rise, there is a growing necessity for the efficient management, analysis, and serving of graph data. This surge has spurred the development of specialized systems capable of handling a wide range of graph-related workloads. Among different kinds of workloads, analytical complex graph queries are commonly used in real-time applications such as information retrieval, recommendation, and fraud detection [1], [2]. For instance, a social networking application may suggest new friends to a user by selecting the 10 most influential individuals reachable within k steps of the "knows" relationship from that user. Figure 1a presents the Gremlin [3] code for implementing such an example query. Such queries access a large proportion of vertices and edges in the graph, involving complex query operations like graph traversal, deduplication, filtering, and aggregation, and requires millisecond-level latency.

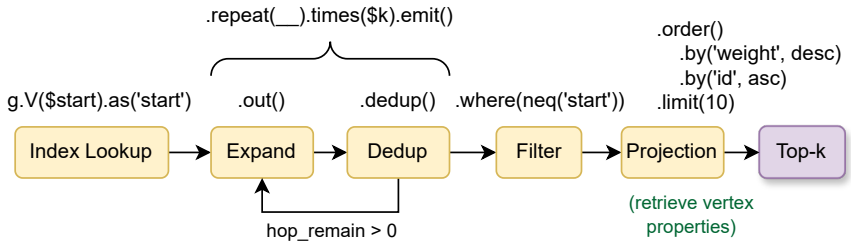
One of the primary challenges in efficiently performing complex graph queries lies in the size of graph datasets. Contemporary graphs can scale up to billions of edges and terabytes of property data, necessitating distributed environments to process. Moreover, graph queries often contain a multitude of query operators, which establish complex dependencies between subtasks. These operators also contain in sparse and irregular acesses to graph data. Furthermore, the trend in hardware technology has shifted towards the widespread utilization of multi-core CPUs and high-speed networks. Modern servers routinely incorporates processors with tens of cores per socket, presenting enhanced opportunities for the parallel execution of various tasks. Additionally, networking bandwidth has surpassed 100Gbps, allowing swift data transfer and communication among nodes. However, these hardware advancements also present considerable challenges in designing efficient systems capable of handling diverse graph queries and fully leveraging the capabilities of modern computing devices.

To express and handle complex graph queries, many graph query languages and execution models have been developed to handle complex graph queries on large graphs. One notable example is Gremlin [3], a popular graph traversal language adopted by systems such as Apache TinkerPop [4] and JanusGraph [5]. In Gremlin, a set of traversers navigates the graph to execute query tasks. Each traverser maintains its own state, performs different traversal steps, and may spawn new traversers. Users can combine various steps to achieve complex graph traversals and queries. However, the Gremlin traverser model tightly couples all execution states to individual traversers, hindering the implementation of certain algorithms and optimizations. For instance, during a multi-hop graph traversal shown in Figure 1, it is often necessary to remove duplicated traversers to avoid redundant computation. However, Gremlin lacks the mechanism for a traverser to determine whether another traverser has already visited the current vertex. Although Gremlin includes a Dedup step to address this issue, common implementations tend to execute it within a single thread, leading to performance bottlenecks in parallel systems. Similar challenges arise with general operators such as join and aggregation, which are not well-suited to the Gremlin traverser model.

Despite the expressive power of the query language, an efficient implementation of the execution engine is also critical to a high-performing graph query system. The recommended

```
g.V($start).as('start').
  repeat(out()).times($k).
  emit().dedup().
  where(neq('start')).
  order().by('weight', desc).
    by('id', asc).
  limit(10)
```

(a) Gremlin code



(b) Optimized traversal program

Fig. 1: An example k-hop neighborhood query. The query requires finding all vertices within k hops from `$start` and returning the 10 most weighted (influential) ones, with ties broken by vertex id.

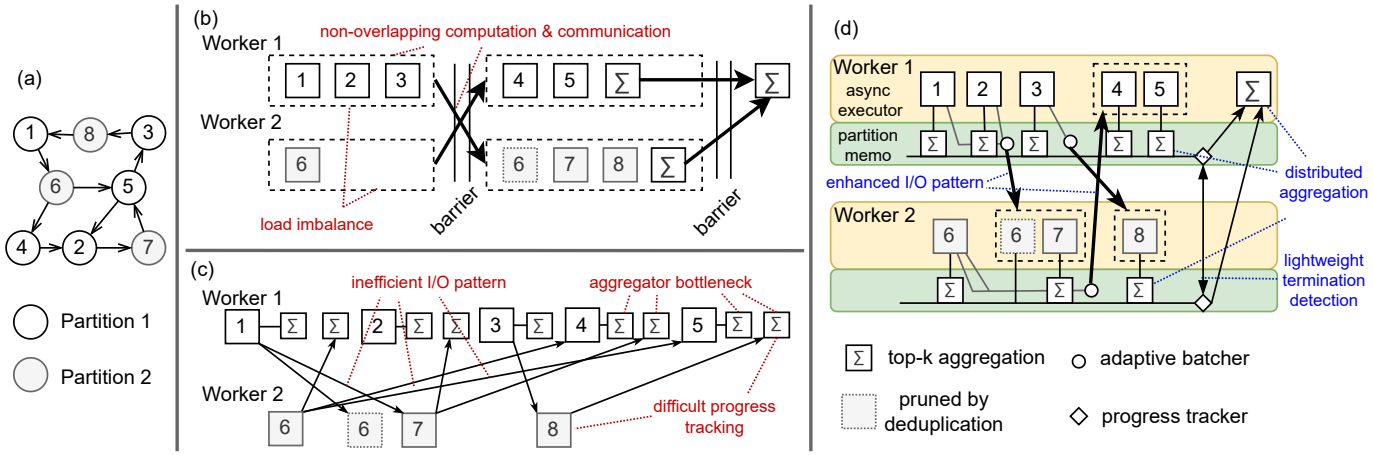


Fig. 2: Comparison of different execution models when executing the plan in Figure 1b. Only Expand, Dedup, and Top-k are plotted for ease of demonstration. (a) The example partitioned graph. (b) Execution with the BSP model. Traversers shall wait for global barriers before moving to adjacent vertices, causing load imbalance and hardware underutilization. (c) Execution with the asynchronous model. Traversers are not synchronized at superstep boundaries; however, spawning remote asynchronous traversers incurs inefficient I/O patterns. Also, centralized result aggregation and progress tracking deteriorate the query latency. (d) Execution with PSTM. Communications across partitions are organized as mini-batches. Aggregation and termination detection are implemented in a distributed and efficient manner.

implementation of Gremlin [3] adopts the bulk synchronous parallel (BSP) protocol [6]. In this protocol, the query execution is organized into distinct phases, or *supersteps*. Each superstep is dedicated to processing a specific type of operation in parallel, followed by a global barrier to synchronize and exchange data between different workers. BSP-based systems often face practical challenges that lead to suboptimal performance [7], [8] in graph workloads. A notable issue is the straggler problem, particularly prevalent in complex queries where each superstep only accesses a dynamic and sparse subset of the graph. Moreover, the inherent separation of computation from communication in the BSP model results in non-overlapping utilization of different hardware resources.

Recent research has also explored asynchronous systems for graph processing [9], [10] to avoid the shortcomings of the BSP model. These frameworks enable worker threads to execute tasks and send messages independently without global barriers. This advancement improves hardware utilization, increases flexibility, enhances the chance for concurrent task execution, and effectively addresses the straggler problem. However, transitioning to asynchronous execution for complex graph queries also presents its own challenges on modern hardware. Most importantly, as asynchronous systems decompose the query into fine-grained subtasks, the additional overhead of task scheduling and progress tracking may introduce significant CPU overhead. We find that simplistic methods for progress tracking may inadvertently introduce significant overhead. Also, the execution of fine-grained tasks may generate an excessive number of small and scrappy messages over the network, which are limited by the message rate of the networking stack. These challenges must be addressed to develop an efficient asynchronous graph query system.

B. Our Contributions

To enhance the expressive power of the Gremlin language and support the development of an efficient distributed graph query engine, we propose an extension to the original Gremlin graph traversal machine model. This extension is based on the

observation that many query operators exhibit the partitionable property (§III-A). Following this observation, we evolve the Gremlin machine to the *partitioned stateful traversal machine* (PSTM) by integrating the partition-aware query memoranda. This innovative design allows different traversers to coordinate with each other without excessive communication and synchronization, supporting efficient expression and implementation of operations like Dedup and Join. Also, this partitioned model facilitates a high-performance implementation of distributed asynchronous graph query engine. By providing partition awareness in the graph traversal model, we unlock a range of system-level optimizations previously challenging to achieve in previous graph traversal systems. Moreover, our tailored implementation of PSTM bridges the gap between the flexibility and expressiveness of the upper-layer programming model and the efficiency of the underlying execution engine which requires fast and lightweight progress tracking, optimized data placement, and high-throughput message passing.

As an example, our analysis reveals that progress tracking and termination detection significantly impact the performance of asynchronous graph query systems, though these aspects are often neglected. Our experiments have shown that simple progress tracking can substantially prolong the query latency by up to $4.46\times$. To this end, we integrate a novel weight-based technique for query progress tracking into PSTM, which uses scalar values to represent task progressions and requires only a single integer addition per traverser for progress tracking. We will provide detailed explanations of this tailored progress tracking mechanism in §IV-A. Moreover, in PSTM, progress tracking can be parallelized and distributed among all workers. Each worker accumulates the total progress made by local traversers and only reports its combined progress to the central query progress tracker when necessary. By aggregating the progress of multiple traversers within the same worker, we significantly reduce the workload of the progress tracker, thereby enhancing overall query latency.

Based on this innovative model, we developed GraphDance, an asynchronous distributed graph database system meticulously engineered to achieve both high throughput and low latency for complex graph queries. We conducted comprehensive experiments to assess the performance and scalability of GraphDance, benchmarking it against leading open-source and commercial graph query systems. The results demonstrate that GraphDance is capable of processing complex queries on billion-scale property graphs with single-digit millisecond latency, marking a significant improvement compared to the baseline systems. An average of 89.2% lower latency is observed across all the 14 complex queries in LDBC Social Network Benchmark (LDBC SNB) [11]. We also include further microbenchmarking that sheds light on the performance characteristics of GraphDance and the impact of each implemented optimization. Our code for reproducing these results is shared in an anonymous code repository, available at https://anonymous.4open.science/r/GraphDance.

II. Background

A. Graph Workloads

To further understand the characteristics of different types of graph workloads, we categorize them into *transactional queries*, *interactive complex queries*, and *offline analytics*, based on their complexity, data access size, and performance requirements as listed in Table I:

- **Transactional queries** primarily involve updating vertex/edge data and retrieving vertex/edge properties and neighborhood information. These queries contain simple query logic and require minimal data access from the query. Transactional graph processing systems are expected to deliver high throughput and very low latency, and provide ACID guarantees.
- **Interactive complex queries** are commonly used in applications like social network recommendations, real-time fraud detection, and knowledge graph searching. These queries often access process a significant subset of the graph via multi-hop traversals, exhibit irregular data access patterns, and involve complex query logic such as join, deduplication, and aggregation. These queries also have strict requirements for query latency to ensure quick response. For example, a search engine has only about $50\,\mathrm{ms}$ time budget to render the page, and any queries from knowledge graphs that fail to complete within this time limit will simply be aborted [12].
- **Offline analytics** focus on scanning and analyzing the whole graph, with typical applications including PageRank [13], community detection, and graph coloring. Graph analytical systems often execute iterative algorithms, with each iteration running a user-defined program over the entire graph [14], [15].

Among all these workload types, interactive complex queries pose unique challenges due to their extensive and dynamic data processing needs within tight time constraints. This paper concentrates on designing graph query systems that balance their demands for both interactive-level latency and high throughput.

B. Gremlin Graph Traversal Machine

To address the challenges posed by complex queries, the *Gremlin graph traversal machine* [3] is introduced as a general framework for defining graph query logic. This model, widely embraced across modern graph databases [5], [9], [10], consists of three primary components: a property graph G, a traversal program Ψ, and a set of traversers T.

The property graph model represents graph data G as a triplet (V, E, λ), where V is the set of vertices, E is the set of directed edges, and $\lambda : (V \uplus E) \times \mathsf{Key} \to \mathsf{Value}$ is a function that assigns properties to vertices and edges as key-value pairs. Two special edge property keys `_src` and `_dest` $\in V$ are used to indicate the endpoints of each edge.

The traversal program Ψ, written in Gremlin code (like Figure 1a), composes a tree of *traversal steps*. The traversal steps allow the traversers to move around the graph, perform

TABLE I: Characteristics of different graph workloads

	Transactional Queries	**Interactive Complex Queries**	**Offline Analytics**
Examples	vertex/edge retrieval	friend discovery, job referral	PageRank, community detection
Typical Benchmark	LDBC SNB Short Reads	LDBC SNB Complex Reads	LDBC Graph Analytics
Accessed Graph Data	$< 0.01\%$	$0.1\% \sim 10\%$	$\sim 100\%$
Data Access Pattern	very sparse	sparse	dense
Number of Compute Stages	$1 \sim 3$	$3 \sim 10$	N/A
Potential Parallelism	limited	massive	massive
Expected Response Time	µs- to ms-level	ms- to sec-level	min- to hour-level
Expected Query Throughput	millions QPS per node	thousands QPS per node	< 1 QPS per node
Theoretical Bottleneck	data access latency	data access bandwidth	data access bandwidth

various query operations, and generate new traversers. Each traverser $t \in T$ is located in a graph vertex $\mu(t)$ and executes a specific traversal step $\psi \in \Psi$.

Sometimes, a sequence of the steps in the traversal program can be expressed and executed in a more efficient way. Hence, the Gremlin compiler may apply a range of *traversal strategies* to optimize the traversal program [16]. Each traversal strategy defines a rewriting rule that converts a section of the traversal into a semantically equivalent yet more efficient form. For example, the IndexLookUpStrategy replaces a full vertex scan followed by a filter with an index look-up operation, reducing the size of accessed data.

As an example, the query presented in Figure 1 will be compiled and optimized into the following steps:

- An IndexLookup step that identifies the start vertex and launches an initial traverser on it;
- k Expand steps, during which each traverser t spawns sub-traversers along vertex $\mu(t)$'s outgoing edges;
- A Filter step that removes start vertices from consideration;
- A Projection step that maps vertex identifiers to their properties;
- An Aggregation step that compiles the top 10 results.

C. Parallel Processing for Interactive Complex Graph Queries

As the demand for real-time processing of large-scale graph data continues to grow, distributed graph processing systems have emerged as a solution to offer various kinds of graph services. Due to the tremendous data access and computation involved in complex graph queries, parallel and distributed graph query engines are essential to meet the stringent time constraints. In these systems, the graph vertex set V is often divided into *partitions* with a hash function $H : V \rightarrow \mathsf{PartId}$, where $\mathsf{PartId} = \{0, 1, \ldots, n_{\text{parts}} - 1\}$ denotes the set of partitions. Each graph partition can thus be handled by a separate worker, allowing parallel processing of large graphs. The traversers in the Gremlin traversal machine are able to independently and concurrently move around the graph as dictated by Ψ, exposing the intrinsic intra-query parallelism of complex graph queries. Such parallelism can be exploited in different ways to accelerate query processing.

1) BSP Systems: For example, many graph processing systems [17]–[22] adopt BSP model for parallel execution due to its implementation simplicity. However, when it comes to complex graph queries, BSP-based systems face notable hurdles such as stragglers and low hardware utilization.

2) Asynchronous Systems: To address the constraints of the BSP model, researchers and industry professionals have explored the potential of *asynchronous* execution models in graph-processing systems. Asynchronous models eliminate unnecessary dependencies dictated by global synchronization, enabling worker threads to communicate or launch tasks independently. This approach is well-suited for the traverser model and has the potential to increase execution flexibility, boost concurrency, and mitigate the impact of stragglers. However, the shift to asynchronous execution in complex graph query processing introduces its own drawbacks.

For example, many operators commonly used in complex graph queries, such as Dedup and Join, require coordination and interaction among multiple traversers. These operators, which operate collectively on a large set of traversers, are not ideal for asynchronous systems. As a result, many existing asynchronous systems resort to executing these operators with a single thread, significantly slowing down query processing.

Moreover, detecting query completion in a distributed asynchronous environment presents a significant challenge. This involves identifying a global quiescent state, where no active traversers are present in any worker threads, and no tasks are in transit over the network. In an asynchronous model where traversers can dynamically and independently spawn new subtasks, confirming the absence of active traversers on a global scale becomes difficult. Consequently, existing asynchronous graph query and analysis systems either revert to using global barriers or employ more sophisticated techniques for task termination detection, both of which can substantially degrade query performance.

Additional challenges include optimizing network bandwidth and preventing data races and consistency issues [7], [8]. Asynchronous models, while advantageous for CPU-intensive tasks, often struggle with message-intensive workloads requiring extensive communication operations. The small-sized messages generated by asynchronous traversers can often lead to bottlenecks in the NIC's IOPS.

III. Partitioned Stateful Traversal Machines

From the discussion above, it is evident that the existing Gremlin model does not admit a high-performance distributed graph query system implementation. Consequently, we have

extended the original Gremlin model to incorporate a stateful, partition-aware architecture. We identify that the increment step execution, along with the partitionable property observed in many query steps, are two key elements towards efficient asynchronous execution of graph traversers. Following this, we detail our extensions to the query execution model, called partitioned stateful traverser machine (PSTM). We highlight PSTM's capacity to facilitate on-demand synchronizations, demonstrated through an ad-hoc deduplication example. We also discuss the termination detection mechanism adopted in PSTM. Finally, we introduce how PSTM handles aggregation steps and subqueries.

In this paper, we focus on the implementation of a Gremlin-based distributed graph query system. Due to the powerful expressiveness of PSTM, our proposed methods can also be adapted for distributed systems using other graph query languages by translating those queries into graph traversal programs. Furthermore, various specialized graph processing tasks, such as graph pattern matching and graph mining, can also be expressed using the Gremlin steps (e.g., Expand and Join), thereby leveraging the advantages offered by PSTM.

A. Efficient Parallel Execution of Query Operators

As outlined in §II-B, Gremlin processes a query by transforming a set of traversers T according to the traversal instructions Ψ, with each step $\psi \in \Psi$ converting input traversers to output traversers. Many steps, such as Expand that generates an output traverser for each outgoing edge and Filter that selectively removes traversers, independently operate on each traverser. These embarrassingly parallel steps allow asynchronous systems to enhance processing speeds by independently executing each traverser in parallel. However, complex graph queries also include many operators which requires complex interactions between traversers. For example, the Dedup step removes duplicated traversers residing in the same vertex, which is a common operation in graph queries to reduce the result set size and avoid combinatorial explosion. Nevertheless, we find that these steps can be efficiently executed in asychronous and distributed systems with *incremental execution* and *traverser partition* techniques, respectively.

a) Incremental step execution: Instead of collectively processing all traversers of such steps with global barriers, we may handle these operators in an *incremental* fashion. We may maintain the internal states of such operators, update the internal states when inputs become available, and produce new incremental outputs if possible. Take the Dedup step as an example. We can maintain the set of visited vertices as S; whenever a traverser t executing the Dedup step, if its location $\mu(t)$ is already in S, it will terminate directly. Otherwise, $\mu(t)$ is added to S and a new traverser is generated to perform subsequent steps. By processing traversers incrementally, we can generate new traversers asynchronously without waiting for all traversers of the current step to be ready.

b) The partitionable property: Although incremental processing helps avoid global synchronizations, managing the internal states of these traversers in centralized workers will introduce performance bottlenecks. Fortunately, our investigation indicates that many steps adhere to the *partitionable* property, which helps parallel execution of such steps among all partitions. Formally, a step ψ that maps a set of traversers to another is partitionable if all traversers T can be partitioned by a function $h_\psi : T \to \mathsf{PartId}$, such that

$$\psi(T) = \biguplus_{p \in \mathsf{PartId}} \psi(\{t_i | h_\psi(t_i) = p\}),$$

where $\biguplus$ denotes the disjoint union of traverser sets. Intuitively, the partitionable property allows dividing the traversers into parititions and limiting their interactions within each partition. For example, the aforementioned Dedup step can be naturally partitioned by the current partition of the traversers, i.e., $h_{\mathsf{Dedup}}(t) = H(\mu(t))$, because the effect of a Dedup traverser depends only on previous traversers of the same partition. By leveraging this property, traverser steps can be executed in parallel without introducing global synchronizations between different partitions.

Besides Dedup, the Join step useful in optimizing many complex graph pattern matchings, is another important step capable of incremental and parallel execution. To illustrate this, consider the example query in Figure 3. Given a person p and a tag t, the query requires to find all posts created by one- or two-hop friends of p with tag t. With the join operation, we can initiate two traversals concurrently, starting from both endpoints. Specifically, we may break the path into two partial paths (denoted `PathA` and `PathB`), find these two patterns independently, and join them at the creator v. The selection of the join key is facilitated by a cost-based query planner, which chooses the key that minimizes the estimated number of all matched partial paths. This join-centric execution plan typically outperforms approaches that solely expand from either endpoint of the pattern path, as it can significantly reduce the size of intermediate data sets.

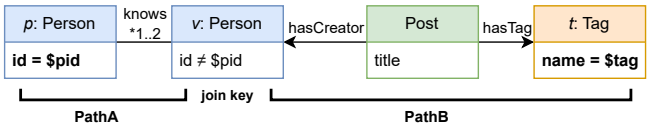


Fig. 3: An example query pattern where bidirectional join outperforms unidirectional traversal from either endpoint.

We use the double-pipelined join algorithm [23] to execute the join step incrementally. Specifically, we maintain the sets of found `PathA` and `PathB` instances at each creator person (i.e., the join key). Upon finding a `PathA` instance π_A at the creator person v, the traverser first inserts π_A to the `PathA` instances at join key v. Then, it probes the hash table of `PathB` to obtain the set of `PathB` instances $\{\pi_B\}$ with creator person v. For each instance π_B in the set, we can immediately spawn a new traverser representing an instance $\pi_A \circ \pi_B$ of the complete query pattern for further processing. The workflow for finding a `PathB` instance is similar. The Join step is also naturally partitionable on join key; hence,

the join computation can be carried out in parallel at different workers for keys in different partitions.

B. Extending the Gremlin Traversal Machine

Based on the above intuition, we have evolved the original property graph model, which solely represented the data graph, to include graph partition information and intermediate execution states. This new model, termed as *partitioned stateful graph* model, is a 5-tuple $G = (V, E, \lambda, H, \mathcal{M})$ that incorporates two new components:

1) $H : V \rightarrow \mathsf{PartId}$ is the graph partitioning function described in §II-C.
2) $\mathcal{M} = \{p \in \mathsf{PartId} \mid M_p\}$ are temporary key-value stores assigned to each partition, called *memoranda* (memos). The memos are primarily used to record some mutable states of various traversal steps, which can be read or written by traversers in the current partition. The memos distinguish themself from the regular graph data in the following aspects:
   - Every query can only access the memo records it creates, and each memo record has its lifetime bound to some specific query. The memo is automatically cleared after the creating query terminates.
   - In transactional graph processing systems, the memo data can be freely read or written by traversers and is not subject to concurrency control. That is, even a read-only graph transaction can modify the memo data.

Correspondingly, a graph traversal query can be represented as a *partitioned stateful traversal machine* (PSTM) program, in which a traverser can be formally defined as a 4-tuple (v, ψ, π, w), where

1) $v \in V$ denotes the current position of the traverser.
2) $\psi \in \Psi$ denotes the current step of the traverser.
3) π denotes a set of local variables, whose meanings are interpreted based upon step specifications. For example, in a projection step, the expression can refer to the parameters in π.
4) $w \in \mathbb{R}$ is a new component not present in the original Gremlin model, called **progression weight**, which is pivotal in PSTM to implement efficient progress tracking and termination detection. Intuitively, it represents the amount of work the current traverser has to do. The root traverser has a progression weight of 1. If a traverser with weight w spawns n $(n \geq 1)$ new traversers, the progress weight of each new traverser is w/n; otherwise, it ends without spawning any new traverser and its weight w is recorded as finished. This maintains the invariant that the sum of weights of all active traversers, including those being transferred over the network, plus the total finished weights equals 1,

$$\sum_{t \in T} w_t + w_{\text{finished}} = 1.$$

Hence, when the total of finished weights equals 1, it indicates that no active traversers remain and the entire traversal has completed.

The capability of this model can be exemplified through the k-hop graph traversal in Figure 1. In multi-hop traversals, many traversers may access the same vertex multiple times via different paths. Hence, duplicated traversers are often pruned to avoid the combinatorial explosion. In BSP systems, this traversal is implemented as k supersteps, each expanding the current vertex set and removing duplicated vertices, as in Figure 4b. While asynchronous expansion of the vertex set is feasible, the requirement of deduplication poses a challenge.

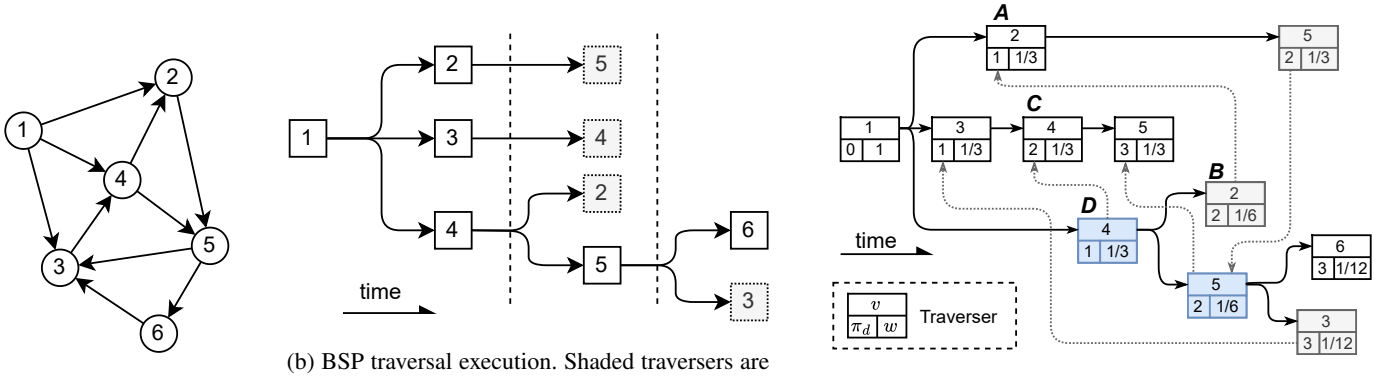
Figure 5 shows the execution plan of the multi-hop traversal in our model. With PSTM, the system can keep track of the shortest distance from the starting vertex to each vertex v in the memo as $M_{H(v)}[\texttt{Distance}, v]$, where `Distance` is a user-defined property label distinguishing it from other key-value pairs in the memo. Thus, a traverser may be pruned if its traversed distance π_d is no less than the known shortest distance to the current vertex. For instance, in Figure 4c, traverser B visiting vertex 2 can identify that traverser A has previously visited this vertex with a distance of 1 from the start vertex, less than B's traversed length. As a result, B will not discover more vertices than A and can be pruned, preventing redundant data access and computation.

Although traverser deduplication is enabled with memos, redundant vertex accesses may still occur. For instance, traverser D visits vertex 4 following traverser C. In this case, D must continue exploring from vertex 4 as it might potentially discover more vertices than C. This redundancy, however, has a negligible effect on the overall performance of asynchronous traversal in practice, as traversers with a shorter history trajectory are generally scheduled to run before those with a lengthier trajectory. Moreover, with memo-assisted deduplication, the time complexity of a k-hop graph traversal is limited to $O(k|E|)$, as each vertex memo will be updated no more than k times. This effectively prevents the combinatorial explosion problem.

C. Supporting Aggregation Steps and Subqueries

Aggregation steps, like Sum, Max, and GroupBy, are commonly used to combine a set of query results. Like partitionable steps, aggregation operations with commutativity and associativity can also handled by all partitions in parallel. Hence, we may store the locally aggregated results in memo. After the termination of all previous traversers, all local results can be extracted and combined to obtain the final results.

Unlike other steps that can produce outputs incrementally, aggregation steps cannot give the final results unless all previous traversers have terminated. As shown in Figure 6, in PSTM, we create a subquery for each set of traversers to aggregate. Each subquery is progress-tracked separately using the weight-based mechanism. After the termination of a subquery, the traverser in the parent query resumes execution with the aggregated results.



(a) Example graph.

(b) BSP traversal execution. Shaded traversers are pruned because of duplication.

(c) Asynchronous execution of PSTM.

Fig. 4: Comparison of BSP and asynchronous model in a 3-hop graph traversal. In the asynchronous execution, gray traversers are pruned because a previous traverser has visited the same vertex with a less-or-equal π_d; however, blue traversers shall continue exploring despite that the current vertex has been visited before.

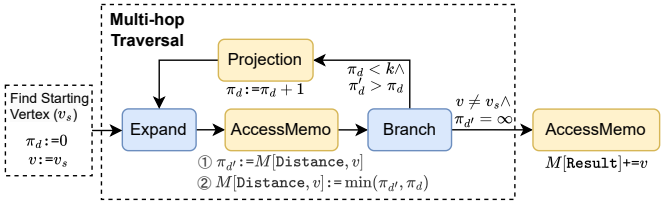


Fig. 5: Execution plan of the multi-hop traversal.

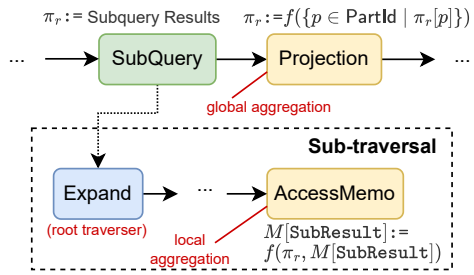


Fig. 6: Execution plan of the aggregation operator with aggregation function f.

IV. The GraphDance System

Based on the PSTM model, we developed GraphDance, a distributed in-memory graph database optimized for interactive complex graph queries. From a bird's-eye perspective, a GraphDance cluster contains a set of nodes for both graph storage and query execution. Each node contains a set of workers managing the graph partitions. GraphDance adopts a shared-nothing design, where each graph partition is managed by a dedicated worker and all traversers on that partition are executed by that worker. Since the traverser can only access the local data partition and memo, this design helps promote cache locality, and create an ideal fit for Non-Uniform Memory Access (NUMA) architectures. As the memo and graph data of each partition are managed by a single-thread worker, accessing the local data and memo does not require expensive inter-thread synchronization primitives like mutexes and semaphores.

A. Efficient Distributed Progress Tracking

During the execution of the graph traversal, most traversers only handle small tasks such as accessing one vertex/edge property. Consequently, it is crucial to keep the amortized progress tracking overhead minimal to avoid impacting the query performance. Also, the system shall promptly detect the end of traversal without introducing excessive latency.

To effectively implement the weight-throwing mechanism without creating a centralized performance bottleneck, we adopt the following key optimizations in our system.

a) Weight coalescing: When there are a large number of traversers terminated, their weights must be actively aggregated to ensure prompt detection of query termination. Using a centralized progress tracker to collect weights of all terminated traversers will clearly lead to a performance bottleneck.

We propose *weight coalescing* to solve this problem. The finshed weights are first temporarily stored and aggregated in the local memo. Whenever we flush the local message buffer as stated in §IV-B, the locally combined weights are also sent to remote progress tracker. This mechanism ensures prompt terminatin detection while also largely reduces the number of messages sent to the progress tracker.

b) Avoiding floating-point arithmetic: When implementing the weight-throwing mechanism with floating-point numbers, it suffers from precision and underflow problems. While this can be solved with arbitrary-precision arithmetic, we use a simpler yet more efficient method. We represent the weight as an element of some finite abelian group G. Whenever we want to spare some weight from w, we choose an element $a \in G$ uniformly and independently at random and split w into a and $w - a$. We prove that the algorithm has a bounded false-positive probability.

Theorem 1. *Let n be the number of coalesced weights sent to the progress tracker. The above algorithm reports a false-positive termination with a probability of at most $(n-1)/|G|$.*

Proof. Let w_i $(1 \leq i \leq n)$ be the total received weight after the ith receipt, where $w_n = 1$ indicates the termination. The algorithm reports a false-positive termination whenever $w_j = 1$ for some $j < n$. For each $j < n$, $1 - w_j$ is the total weight not yet returned to the tracker, which is the sum of several independent random variables uniformly distributed on $|G|$. Hence, $1 - w_j$ is also uniformly distributed over $|G|$, which means $\mathbb{P}(w_j = 1) = 1/|G|$. By Boole's inequality, we have

$$\mathbb{P}\left(\bigcup_{i=1}^{n-1}\{w_j = 1\}\right) \leq \sum_{i=1}^{n-1}\mathbb{P}(w_j = 1) = \frac{n-1}{|G|}. \qquad \square$$

In our implementation, we use 64-bit integers and modular arithmetic to represent and accumulate the weights. Hence, the false-positive rate, $(n-1)/2^{64}$, is negligible in practice.

B. Two-Tier I/O Scheduling

During the execution of a complex graph query with a asynchronous model, millions of messages may be generated asynchronously between the worker threads. The massive amount of small and scrappy I/O operations generated by these asynchronous messages is the root cause of communication inefficiency in asynchronous systems.

We use a two-tier message-passing channel for efficient inter-worker communication. The first tier of the channel performs thread-level message batching. Each worker thread contains a message buffer for each node in the cluster. Messages sent by traversers in this partition are first stashed in the corresponding buffers. Whenever the content size in a buffer exceeds a certain limit (8 KB in our experiments), we flush all the buffered messages to the second tier. Also, if there are no more traversers ready for execution in the current partition, we flush all the buffers before the current thread sleeps. The second tier, managed by separate network threads, combines the messages from different threads. The message pack is then sent over a TCP stream to the remote server. As a shortcut, messages sent to workers on the same node are passed via shared memory instead of networking.

C. Transactional Processing Support

To support transactional updates to the graph data, we use the transactional edge log (TEL) [24] to store the multi-version adjacency lists, which embed the creation and deletion timesteps to the edge data, allowing to find visible edges within a single sequential scan. We use MV2PL for concurrency control, as read-only queries will not be blocked by concurrent update transactions under MV2PL. Specifically, a centralized transaction manager assigns timestamps for update transactions and maintains the *last commit timestamp* (LCT), which means that all transactions before LCT are committed. When the system restarts after a crash, all workers will scan the graph data and remove all versions with timestamps larger than LCT. To reduce the load of the centralized transaction manager, the LCT is broadcast to all worker nodes; a read-only query can fetch the LCT from any worker node as its read timestamp without consulting the transaction manager.

V. Performance Evaluation

To demonstrate the efficiency of GraphDance, we employed the whole LDBC Social Network Benchmark (LDBC SNB) [11] and compare GraphDance with state-of-the-art graph databases in various settings, which simulates a mix of simple and complex query workloads. We also use the k-hop traversal query in Figure 1 on representative real-world graph datasets to study the scalability of GraphDance and the impact of the optimizations we used in the design of GraphDance. As all these graphs are unweighted, we assign a random integer weight to each vertex for aggregation queries. The starting vertex is randomly selected from all vertices for 100 times and the average is reported.

TABLE II: Summaries of graph datasets used in evaluation.

Dataset	# Vertices	# Edges	Raw Size
LDBC SNB SF300	969,958,916	6,729,459,600	256 GB
LDBC SNB SF1000	2,930,667,395	20,718,772,476	862 GB
LiveJournal (LJ) [25]	3,997,962	34,681,189	464 MB
Friendster (FS) [25]	65,608,366	1,806,067,135	31 GB

Unless otherwise mentioned, the experiments are conducted on a cluster of 8 nodes running Ubuntu 20.04 with Linux kernel version 5.4.0. Each node has two Intel Xeon Gold 6240R processors and 384 GB RAM. All nodes are interconnected with the 200 Gbps network. We choose the following graph query engines as baseline systems:

- **TigerGraph** [26], a prominent commercial distributed graph database known for its scalability and superior performance relative to many open-source alternatives in distributed settings. We use its official LDBC SNB query implementations for evaluation.
- **GraphScope** [27], the current top-performing system in LDBC social network benchmarking with officially audited results. While GraphScope supports distributed execution using the Gremlin language, its LDBC queries are implemented with manually optimized C++ procedures tailored for single-node settings only.
- **GAIA** [28], a graph analysis engine on distributed graphs with native Gremlin language support.
- **Banyan** [29], a distributed graph query engine based on scoped dataflows. Because the source code of Banyan is hardcoded for specific queries, we implement Banyan's scoped dataflow mechanism on GraphDance's codebase.

Furthermore, to validate the effectiveness of PSTM model, we have modified GraphDance for a comparative analysis against the following alternative data and execution models:

- **Non-Partitioned Graph Model**: In this scenario, the graph data and query states are not partitioned and are shared by all worker threads.
- **BSP Execution**: Instead of using the traverser model, all queries are executed using BSP model.

A. LDBC SNB Workload

The LDBC Social Network Benchmark is one of the most renowned and popular benchmarks for assessing the

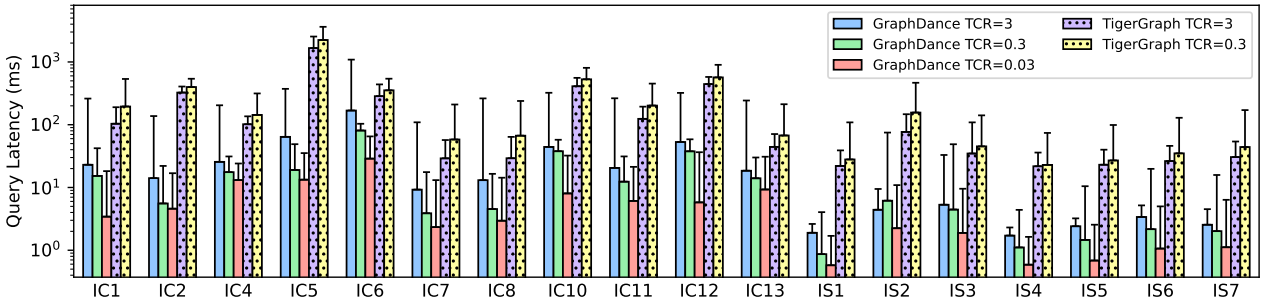


Fig. 7: Average and P99 latency of IC and IS queries using mixed LDBC SNB SF300 Interactive Workload.

performance of graph databases. It comprises a comprehensive suite of 14 representative Interactive Complex queries (ICs), designed to rigorously evaluate a graph database system's ability and performance in various domains, including traversal, join operations, filtering, and data aggregation. Additionally, it incorporates short read and update queries. The detailed definitions of the benchmark dataset and the associated queries are available at https://ldbcouncil.org/ldbc_snb_docs/.

Given the challenges in overcoming communication overhead and achieving high performance in distributed environments, it has been observed that all previously audited graph databases [27], [30], [31] are either limited to single-node or have had their LDBC SNB implementations exclusively tailored for single-node setups. Their distributed implementation, if available, typically exhibit increased latency and reduced throughput, predominantly due to the poor handling of communication, scheduling, and data access for distributed graph traversal executions. Thus, we benchmarked the LDBC SNB performance of GraphDance against TigerGraph, which provides distributed LDBC SNB query implementation. We also compare with GraphScope, which provides the current best single-node implementation.

1) Mixed LDBC SNB Interactive Workload: We first evaluate the performance of GraphDance using the entire mixed LDBC SNB Interactive Workload against TigerGraph, which includes a variety of queries: interactive complex queries (IC), interactive short queries (IS), and transactional updates (UP). Different from stress testing that executes concurrent queries as much as possible, in LDBC, each type of query is issued at a predefined frequency. The frequencies are controlled by a workload parameter known as the Time Compression Ratio (TCR). A lower TCR indicates a higher throughput requirement for systems to achieve. Moreover, due to significant timeouts in TigerGraph's implementation of IC3, IC9, and IC14, we excluded these queries to enable TigerGraph to complete the mixed workload tests successfully. The results of executing these queries will be given in the next section.

Figure 7 compares the latency of all queries between GraphDance and TigerGraph. Notably, TigerGraph fails to complete the test at a TCR of 0.03 because it is unable to keep up with the query issuance rate. Across all queries, GraphDance's average latency is 88.7% and 91.6% less than TigerGraph at a TCR of 3 and 0.3, respectively, reflecting its superior performance as well as its remarkable scalability.

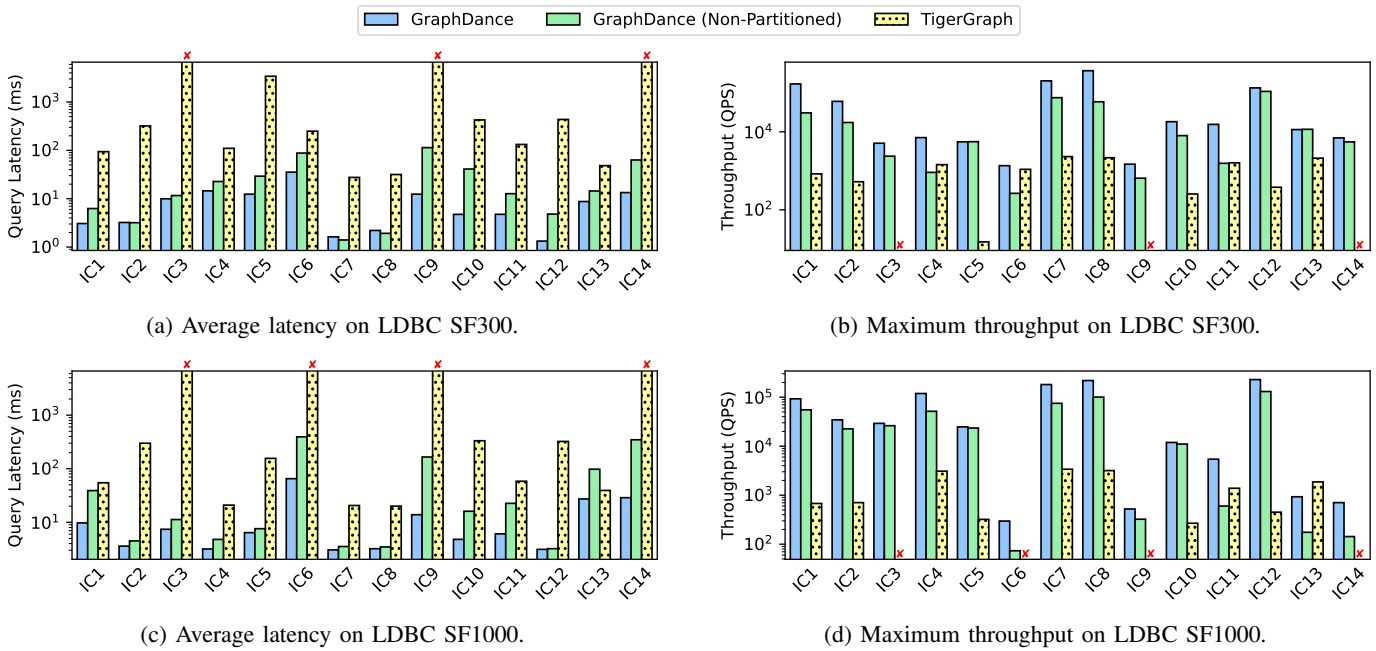
2) Individual Interactive Complex Queries: As reported in Figure 8, we also test the performance of GraphDance and TigerGraph on each interactive complex query individually. Moreover, to evaluate the effectiveness of our partitioned stateful traversal machine model, we also compare against the non-partitioned graph model that shares graph storage and query states across all threads in a node. We measure the minimum query latency by submitting the queries sequentially, and the maximum throughput by using up to 256 concurrent threads to submit queries.

As we can see from the figure, on the LDBC SF300 graph, GraphDance is able to deliver an average of 88.9% lower latencies in interactive complex queries. Meanwhile, GraphDance offers $43.3\times$ higher throughput on average. On the larger LDBC SF1000 graph, GraphDance can still offer 90.3% lower latency and $35.5\times$ higher throughput. This justifies the high efficiency and better resource utilization of asynchronous execution models for interactive complex graph queries, especially under high concurrency.

When compared to a non-partitioned storage, a partitioned graph model achieves an average reduction of 46.5% in latency and a $3.29\times$ increase in throughput. This improvement is primarily attributed to the removal of inter-thread synchronization on the graph storage and query states. In a non-partitioned system, worker threads must employ latches when accessing shared data, which can impose substantial overhead and potentially lead to contention issues. Furthermore, the PSTM design enhances data locality by ensuring that each worker thread accesses only the memory of its local NUMA node and improving the CPU cache hit rate.

3) Comparison against Single-Node System: To delve into the strengths and weaknesses of single-machine versus distributed graph query systems, we conducted a comparative analysis using LDBC SNB graphs and queries. Despite GraphScope's support for distributed graph storage and execution, its LDBC SNB implementation is specifically tailored for single-node deployment. Given that GraphScope currently holds the highest officially audited LDBC SNB results, we selected it as the benchmark for single-node LDBC SNB performance and compared it against GraphDance.

When using the SF300 graph, where the entire dataset can fit into the memory of a single node, GraphScope exhibited



(a) Average latency on LDBC SF300.

(b) Maximum throughput on LDBC SF300.

(c) Average latency on LDBC SF1000.

(d) Maximum throughput on LDBC SF1000.

Fig. 8: Latency and throughput of individual complex read queries from LDBC SNB SF300 and SF1000.

an average query latency that was 58.1% lower than that of GraphDance. This advantage is primarily attributed to its elimination of cross-node communication and scheduling overhead. Also, GraphScope's LDBC implementation uses specialized C++ plugins manually optimized for each query, which lacks generality. On the other hand, distributed systems like GraphDance can achieve higher throughput by scaling across multiple machines. For instance, with eight nodes, GraphDance provided an average throughput that was 2.16× higher than that of GraphScope.

For the larger SF1000 graph, GraphScope was unable to complete 9 out of 14 IC queries within the time limit due to the graph's size exceeding the memory capacity, resulting in frequent memory swapping. This highlights a fundamental limitation of single-node graph query systems, which are only efficient when processing small graphs that can fit within the DRAM of a single node.

B. System Scalability

We also use the example k-hop query described in Figure 1 to study the scalability of GraphDance and compare it against other distributed graph query systems. We both evaluate the vertical scalability by changing the number of worker threads within one node, as well as the horizontal scalability by varying the number of nodes.

From the results in Figure 9, it is evident that GraphDance is capable to achieve almost linear speedup for medium- and large-sized queries. In contrast, dataflow systems such as Banyan and GAIA show limited scalability. This limitation primarily arises because these systems instantiate each dataflow operator in every worker thread, leading to a linear increase in the overhead associated with scheduling and process tracking. On the other hand, the overhead of the weight-based process tracking used in GraphDance is independent of the number of worker threads, enabling it to scale efficiently even with a large number of threads. Moreover, GAIA executes the final aggregation step in a centralized worker, inherently limiting its scalability in both vertical and horizontal cases.

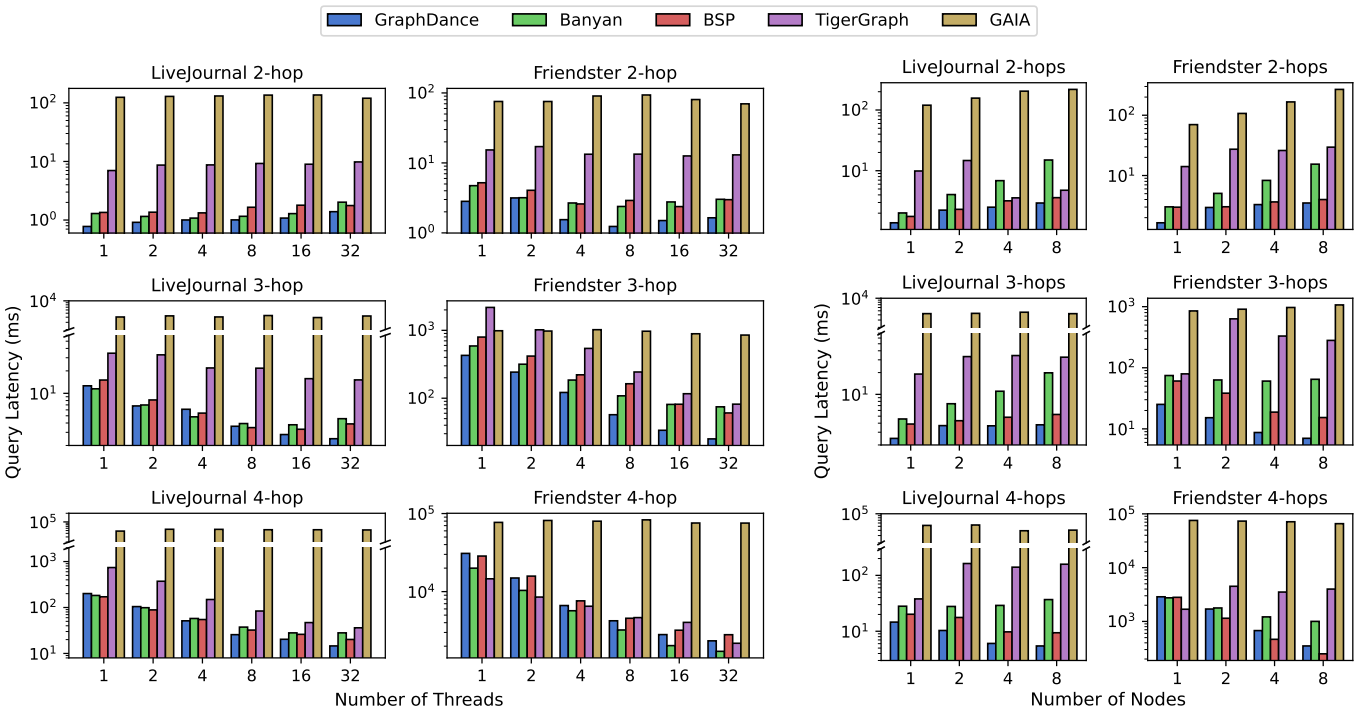
For small numbers of threads, GraphDance might have higher latency than Banyan on 4-hop queries. This is due to the slightly higher per-traverser progress tracking overhead of GraphDance. However, with more worker threads added, the latency of GraphDance quickly decreases. For longer queries, e.g., Friendster 4-hops, the BSP model performs best as it can amortize the scheduling and synchronization costs among a large number of traversers during each iteration.

C. Performance Breakdown

Finally, we assess the impact of some key optimizations of GraphDance. All the experiments are running the same benchmark as §V-B unless otherwise specified.

1) Lightweight Progress Tracking: We evaluate the performance impact of lightweight progress tracking by disabling the weight coalescing (WC) optimization.

The results in Figure 10 indicate that the weight coalescing optimization can save up to 77.6% query execution time. By distributing the weight merging computation over the entire cluster, WC removes the performance bottleneck on the progress tracking of a massive number of traversers. This can be confirmed by the number of messages with or without WC enabled, presented in Figure 11. Without WC, the number of progress tracking messages is comparable to other message types. However, unlike other messages, progress tracking messages must be processed by a centralized worker, creating



(a) Vertical scalability.

(b) Horizontal scalability.

Fig. 9: Scalablity of GraphDance and other graph query systems.

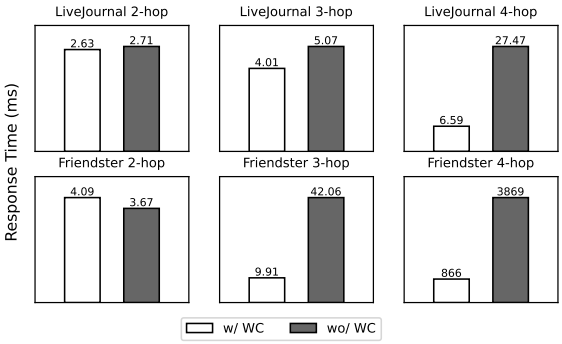


Fig. 10: Impact of weight coalescing in progress tracking.

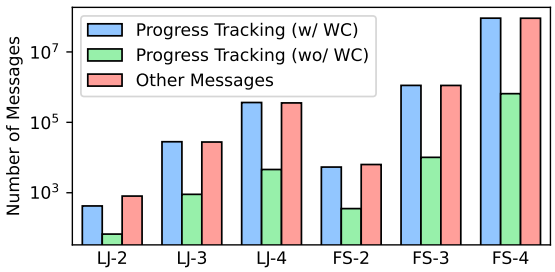


Fig. 11: Number of progress tracking messages and other messages.

a significant performance bottleneck. With WC enabled, the number of progress tracking messages is reduced by 91.2% to 99.3%. It's worth noting that for simpler queries like LiveJournal 2- and 3-hop, the benefits of weight coalescing may not outweigh the associated increase in latency, leading to a higher response time for these particular queries.

2) Two-Tiered I/O Scheduler: To measure the effectiveness of our two-tiered I/O scheduler design described in §IV-B, we first evaluate the performance of a baseline implementation that synchronously sends every message to the underlying TCP stream. Then, we examine the performance change by enabling thread-level message combining (TLC). Finally, we compare them with the full GraphDance implementation by further enabling node-level message combining (NLC).

As we can see in Figure 12, thread-level combining significantly improves the performance for all test queries. TLC is particularly effective in large queries (e.g., 15.9× speedup in Friendster 4-hop), which involves many message passings. By combining many smaller messages within a thread, we not only avoid frequent inter-thread synchronizations but also save many expensive system calls, which amortizes the additional overhead of massive fine-grained tasks.

On the other hand, node-level combining has a minor improvement in the overall response time in large queries. This is because the number of messages combined in NLC is generally no more than the number of threads in the local node, significantly smaller than the number of concurrent traversers within a region. In smaller, latency-bounded queries, NLC might slightly slow the execution time because NLC will increase the message-passing latency.

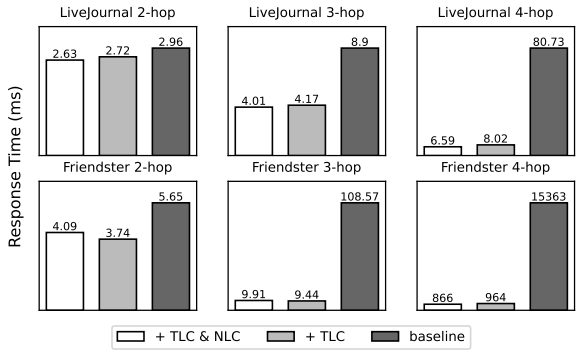


Fig. 12: Impact of the two-tiered I/O scheduler.

3) Hardware Impact: We also study the performance impact of modern hardware by experimenting using legacy hardware configurations with reduced networking bandwidth and CPU core count.

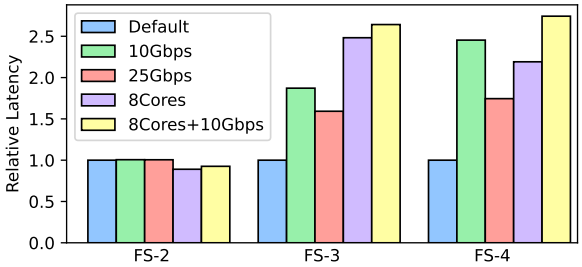


Fig. 13: Relative query latency with reduced networking bandwidth and CPU core count.

As shown in Figure 13, the hardware configuration significantly influences the performance of longer 3- and 4-hop queries, yielding up to a 2.74× performance boost when using modern hardware. In contrast, for shorter 2-hop queries, increasing networking bandwidth and CPU core count does not enhance performance, as these queries are primarily latency-bound. Additionally, we find that increasing both networking bandwidth and computational power is crucial for achieving optimal query latency; either resource can become a potential bottleneck for longer queries.

VI. Related Work

a) Graph databases and graph queries: With the increasing demand for graph data management in recent years, graph databases and graph query engines have attracted great interest in the industry. Neo4j [32] is one of the earliest graph databases that adopt a native graph storage engine. GraphX [33] is an analytical graph engine based on Spark. It stores vertices and edges using resilient distributed datasets (RDD) and maps graph APIs to dataflow operators. Banyan [29] is a graph query engine using scoped dataflow, an extension of timely dataflow [34], to achieve fine-grained task control and performance isolation. Besides the graph databases discussed above, ByteGraph [10] and aDFS [35] are also recently proposed distributed graph databases that are aware of the different characteristics of different graph workloads.

b) Parallel and distributed query execution: In relational graph databases, many studies have proposed techniques to optimize distributed parallel execution by using local computation at the partition level. For example, Dremel [36] and Apache Arrow [37] apply local aggregation before global aggregation, while Spark SQL [38] uses local deduplication to reduce redundancy. These optimizations, compared to centralized query execution, alleviate performance bottlenecks and minimize data transfer. However, existing parallel graph query engines either convert graph queries to relational queries [33], [39], or only optimize specific graph operations such as graph traversal and pattern matching [20], [40], [41]. In contrast, GraphDance seamlessly integrates advanced optimizations commonly found in relational systems with a native distributed graph query execution engine.

c) Sync and Async execution in distributed systems: In distributed and parallel computation, synchronous (*Sync*) and asynchronous (*Async*) execution modes each have unique strengths and limitations. Given their distinct advantages, some graph processing systems, such as PowerGraph [19], Trinity [42], and PowerLyra [18], allow users to select between *Sync* and *Async* execution depending on the workload. PowerSwitch [8] introduces an adaptive approach, dynamically switching between modes during different phases of query execution to optimize performance. Although GraphDance adopts the *Async* model for low-latency, complex graph queries, integrating *Sync* mode or PowerSwitch's hybrid approach in GraphDance could further improve the performance of long-running queries.

VII. Conclusion

This paper introduces PSTM, a stateful asynchronous graph traversal model that enhances the expressive power of Gremlin and bridges the gap between the flexibility of the upper-layer programming model and the architectural preferences of the lower physical layers, thereby enhancing both CPU and network utilization. Utilizing PSTM, we have developed GraphDance and compared its performance against leading systems using the LDBC SNB benchmark. The results demonstrate that GraphDance achieves an average of 89.2% lower latency than state-of-the-art systems across all interactive complex queries.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. The authors affiliated with Tsinghua University are all in the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China. This work is supported by National Key Research & Development Program of China (2022YFB2404200), Natural Science Foundation of China (62141216) and Tsinghua University Initiative Scientific Research Program, Young Elite Scientists Sponsorship Program by CAST (2022QNRC001), and Beijing HaiZhi XingTu Technology Co., Ltd. Correspondence to: Mingxing Zhang (zhang_mingxing@mail.tsinghua.edu.cn).

REFERENCES

[1] V. S. Tseng, J.-C. Ying, C.-W. Huang, Y. Kao, and K.-T. Chen, "Fraudetector: A graph-mining-based framework for fraudulent phone call detection," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 2157–2166.

[2] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing: extended survey," *The VLDB journal*, vol. 29, no. 2, pp. 595–618, 2020.

[3] M. A. Rodriguez, "The gremlin graph traversal machine and language (invited talk)," in *Proceedings of the 15th Symposium on Database Programming Languages*, ser. DBPL 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–10. [Online]. Available: https://doi.org/10.1145/2815072.2815073

[4] T. A. S. Foundation, "Apache TinkerPop," https://tinkerpop.apache.org/, 2024.

[5] JanusGraph Authors, "JanusGraph," https://janusgraph.org/, 2023.

[6] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[7] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, pp. 1–39, 2015.

[8] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "Sync or async: Time to fuse for distributed graph-parallel computation," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 194–204, 2015.

[9] H. Chen, C. Li, J. Fang, C. Huang, J. Cheng, J. Zhang, Y. Hou, and X. Yan, "Grasper: A high performance distributed system for olap on property graphs," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 87–100. [Online]. Available: https://doi.org/10.1145/3357223.3362715

[10] C. Li, H. Chen, S. Zhang, Y. Hu, C. Chen, Z. Zhang, M. Li, X. Li, D. Han, X. Chen, X. Wang, H. Zhu, X. Fu, T. Wu, H. Tan, H. Ding, M. Liu, K. Wang, T. Ye, L. Li, X. Li, Y. Wang, C. Zheng, H. Yang, and J. Cheng, "Bytegraph: A high-performance distributed graph database in bytedance," *Proc. VLDB Endow.*, vol. 15, no. 12, p. 3306–3318, sep 2022. [Online]. Available: https://doi.org/10.14778/3554821.3554824

[11] R. Angles, J. B. Antal, A. Averbuch, A. Birler, P. Boncz, M. Búr, O. Erling, A. Gubichev, V. Haprian, M. Kaufmann, J. L. L. Pey, N. Martínez, J. Marton, M. Paradies, M.-D. Pham, A. Prat-Pérez, M. Spasić, B. A. Steer, D. Szakállas, G. Szárnyas, J. Waudby, M. Wu, and Y. Zhang, "The ldbc social network benchmark," 2020. [Online]. Available: https://arxiv.org/abs/2001.02299

[12] C. Buragohain, K. M. Risvik, P. Brett, M. Castro, W. Cho, J. Cowhig, N. Gloy, K. Kalyanaraman, R. Khanna, J. Pao, M. Renzelmann, A. Shamis, T. Tan, and S. Zheng, "A1: A distributed in-memory graph database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 329–344. [Online]. Available: https://doi.org/10.1145/3318464.3386135

[13] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[14] O. Batarfi, R. E. Shawi, A. G. Fayoumi, R. Nouri, S.-M.-R. Beheshti, A. Barnawi, and S. Sakr, "Large scale graph processing systems: survey and an experimental evaluation," *Cluster Computing*, vol. 18, pp. 1189–1213, 2015.

[15] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya, "Scalable graph processing frameworks: A taxonomy and open challenges," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–53, 2018.

[16] M. A. Rodriguez and J. Shinavier, "Exposing multi-relational networks to single-relational network analysis algorithms," *Journal of Informetrics*, vol. 4, no. 1, pp. 29–41, 2010.

[17] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 301–316.

[18] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 3, pp. 1–39, 2019.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 17–30.

[20] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[21] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: a system for distributed graph mining," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 425–440.

[22] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang, "Knightking: a fast distributed graph random walk engine," in *Proceedings of the 27th ACM symposium on operating systems principles*, 2019, pp. 524–537.

[23] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld, "An adaptive query execution system for data integration," *ACM SIGMOD Record*, vol. 28, no. 2, pp. 299–310, 1999.

[24] X. Zhu, G. Feng, M. Serafini, X. Ma, J. Yu, L. Xie, A. Aboulnaga, and W. Chen, "Livegraph: a transactional graph storage system with purely sequential adjacency list scans," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1020–1034, 2020.

[25] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, 2012, pp. 1–8.

[26] TigerGraph, Inc, "TigerGraph," https://www.tigergraph.com/, 2023.

[27] W. Fan, T. He, L. Lai, X. Li, Y. Li, Z. Li, Z. Qian, C. Tian, L. Wang, J. Xu *et al.*, "Graphscope: a unified engine for big graph processing," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, pp. 2879–2892, 2021.

[28] Z. Qian, C. Min, L. Lai, Y. Fang, G. Li, Y. Yao, B. Lyu, X. Zhou, Z. Chen, and J. Zhou, "GAIA: A system for interactive analysis on distributed graphs using a high-level language," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 321–335.

[29] L. Su, X. Qin, Z. Zhang, R. Yang, L. Xu, I. Gupta, W. Yu, K. Zeng, and J. Zhou, "Banyan: A scoped dataflow engine for graph query service," *Proc. VLDB Endow.*, vol. 15, no. 10, p. 2045–2057, jun 2022. [Online]. Available: https://doi.org/10.14778/3547305.3547311

[30] A. Group, "Tugraph," https://www.tugraph.org/, 2023.

[31] CreateLink, "Galaxybase," https://www.createlink.com/galaxyProduct, 2024.

[32] Neo4j, Inc, "Neo4j graph data platform," https://neo4j.com/, 2023.

[33] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, 2014, pp. 599–613.

[34] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 439–455.

[35] V. Trigonakis, J.-P. Lozi, T. Faltín, N. P. Roth, I. Psaroudakis, A. Delamare, V. Haprian, C. Iorgulescu, P. Koupy, J. Lee, S. Hong, and H. Chafi, "aDFS: An almost Depth-First-Search distributed Graph-Querying system," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 209–224. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/trigonakis

[36] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, 2010.

[37] A. S. Foundation, "Apache arrow: A cross-language development platform for in-memory data," https://arrow.apache.org/, 2016, version 1.0.0, Accessed: 2024-10-24.

[38] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1383–1394.

[39] M. Hammoud, S. Sakr, K. Zoumpatianos, F. Abuzaid, and A. Mhedhbi, "Tegra: Efficient ad-hoc analytics on evolving graphs using a relational query processor," in *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*, 2018, pp. 465–480.

[40] X. Sun, Y. Zhu, X. Qin, and H. Wang, "Scalable in-memory graph pattern matching on symmetric multiprocessor systems," in *Proceedings*

of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2012, pp. 184–194.

[41] W. Han, J.-H. Lee, S. Kasif, L. Wang, and J. Kim, "Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 2013, pp. 337–348.

[42] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 505–516.