Banyan: A Scoped Dataflow Engine for Graph Query Service

Li Su[†], Xiaoming Qin[†], Zichao Zhang[†], Rui Yang[‡], Le Xu[★], Indranil Gupta[‡], Wenyuan Yu[†], Kai Zeng[◇], Jingren Zhou[†]

Alibaba Group[†], University of Illinois Urbana-Champaign[‡], The University of Texas at Austin[★], UESTC[◇]

{lisu.sl, xiaoming.qxm, houbai.zzc, wenyuan.ywy, jingren.zhou}@alibaba-inc.com[†]
{ry2, indy}@illinois.edu[‡], lexu@cs.utexas.edu[★], kaizeng.zk@gmail.com[◇]

ABSTRACT

Graph query services (GQS) are widely used today to interactively answer graph traversal queries on large-scale graph data. Existing graph query engines focus largely on optimizing the latency of a single query. This ignores significant challenges posed by GQS, including fine-grained control and scheduling during query execution, as well as performance isolation and load balancing in various levels from across user to intra-query. To tackle these control and scheduling challenges, we propose a novel *scoped* dataflow for modeling graph traversal queries, which explicitly exposes concurrent execution and control of any subquery to the finest granularity. We implemented Banyan, an engine based on the scoped dataflow model for GQS. Banyan focuses on scaling up the performance on a single machine, and provides the ability to easily scale out. Extensive experiments on multiple benchmarks show that Banyan improves performance by up to three orders of magnitude over state-of-the-art graph query engines, while providing performance isolation and load balancing.

PVLDB Reference Format:
Li Su, Xiaoming Qin, Zichao Zhang, Rui Yang, Le Xu, Indranil Gupta, Wenyuan Yu, Kai Zeng, Jingren Zhou. Banyan: A Scoped Dataflow Engine for Graph Query Service. PVLDB, 15(10): 2045 - 2057, 2022.
doi:10.14778/3547305.3547311

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://graphscope.oss-cn-beijing.aliyuncs.com/vldb/banyan_code.zip.

1 INTRODUCTION

Graph query services (GQS) are widely used in many Internet applications ranging from search engines, recommendation systems, to financial risk management. The global GQS market is estimated to reach 2.9 billion USD by 2024, with an annual growth of 22.2% [13]. In these applications, data are represented as graphs such as knowledge graphs and social networks. Explorations on the data are usually expressed as graph traversal queries. GQS serves these large-scale graphs for interactive query access, allowing many users to submit graph traversal queries and obtain results in real-time.

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547311

Previous graph query engines [15, 26, 34, 35, 43, 46] mainly focus on optimizing the query latency by improving computation efficiency, i.e., traversing more vertices/edges in a time unit. However, we observe that optimizing for computation efficiency *alone* is not sufficient to achieve short query latency. A multi-tenant GQS needs to address two key goals to fulfill stringent latency requirements: (**O1**) fine-grained control and scheduling, as well as (**O2**) performance isolation and load balancing during the query execution.

Fine-Grained Control and Scheduling (O1). A graph query usually performs many traversals starting from different source vertices in the graph. We observe two key techniques that can reduce query latencies, and in turn boost the system throughput: (**O1-1**) concurrently executing these traversals, and controlling them at a fine granularity; (**O1-2**) carefully choosing the traversal strategy.

Example 1 (A Graph Traversal Query in Gremlin)**.**

```
g.V(123).repeat(out('knows'))
    .until(out('worksAt').is(eq('XYZ')))
    .or().loops().is(gt(5)))
 .where(in('tweets').out('hasTag').is(eq('#ABC')))
 .limit(20)
```

Example 1 shows a graph traversal query intended to find 20 users who are within the 5-hop neighborhood of user 123, work at company 'XYZ', and have tweeted with hashtag '#ABC'. When executing this query, every user entering the *where* subquery starts a new traversal, which can be canceled immediately if any tweet of the user is found to have the desired hashtag. Canceling this traversal should not affect the traversals of other users. More importantly, it should not be blocked by other traversals, e.g., by users who tweet a lot but without hashtag '#ABC'. This implies that a GQS needs to support concurrent execution and fine-grained control of subquery traversals (**O1-1**).

Eagerly checking the hashtags tweeted by one user requires a DFS scheduling policy for the *where* subquery. The same scheduling policy holds within each loop iteration in the *repeat* subquery, as we would like to eagerly check if a neighbor works at company 'XYZ'. However, it would be preferable to use BFS when scheduling across loop iterations of the *repeat* subquery. This is because we do not want to blindly explore all neighbors within 5 hops if 20 candidates can be found in a much smaller neighborhood. This implies that GQS systems need to support customized traversal policies in subqueries (**O1-2**).

Performance Isolation and Load Balancing (O2). It is challenging to fulfill the stringent latency requirement in a production environment. Due to the intrinsic skewness in graph data, graph traversal queries can vary dramatically in terms of the amount of computation. Therefore, GQS should be capable of enforcing

Table 1: Comparing graph query engines on O1 and O2.

	Neo4j	JanusGraph	Timely	GAIA	Banyan
O1-1	✗	✗	✗	✓	✓
O1-2	✗	query-level only	✗	query-level only	✓
O2	✗	✗	✗	✗	✓

performance isolation across queries to guarantee the latency SLO. In addition, as illustrated in Example 1, for better performance, the isolation granularity needs to be as small as subquery-level traversals. Existing graph databases [15, 26, 27] map concurrent queries to system threads and rely on the operating system for scheduling. This mechanism does not expose the internal query complexity and thus cannot support subquery-level isolation. The skewness in graph data can lead to dynamic workload skewness at run time, requiring GQS to provide dynamic load balancing.

Table 1 compares existing graph query engines on their supports for **O1** and **O2**. Many graph query engines [8, 22, 25, 32, 36, 39] use the dataflow model to execute graph traversal queries in languages such as Cypher [9], Gremlin [40] and GSQL [39]. Existing dataflow models either a) support only static topologies (e.g., Timely [23] and GAIA [32]) or b) can only dynamically spawn tasks at coarse granularity (e.g., CIEL [24]). Specifically, Timely and GAIA attach a metadata in every message to identify which subquery traversal it belongs to, but then these systems process messages belonging to different traversals of a subquery in the same static execution pipeline determined at compile time. Without physically isolating the traversals inside a subquery, these systems cannot efficiently support both **O1** and **O2**. We discuss the limitations of existing dataflow models in Section 2, and evaluate this issue in Section 5.

In this paper, we propose a novel *scoped dataflow* to model graph traversal queries, which supports fine-grained control and scheduling during query execution. The scoped dataflow model introduces a key concept called *scope*. A scope marks a subgraph in the dataflow, which corresponds to a subquery. The scope can be dynamically replicated at runtime into physically isolated scope instances which correspond to independent traversals of the subquery. Scope instances can be concurrently executed and independently controlled. This way, traversals of a subquery can time-share the CPU and be independently canceled without blocking or affecting each other. Furthermore, a scope allows users to customize the scheduling policy between and inside scope instances, supporting diverse scheduling policies for different parts of a query.

We build the *Banyan* engine for a multi-tenant GQS, based on an efficient distributed implementation of scoped dataflow. Banyan parallelizes a scoped dataflow into a physical plan of operators, and cooperatively schedules these operators on executors pinned on physical cores. On each executor, Banyan dynamically creates and terminates operators to instantiate and cancel scope instances. Banyan manages operators hierarchically as an operator tree: operators are scheduled recursively by their parent scope operators. This hierarchy allows customized scheduling within each scope, and provides performance isolation both across queries and within a single query. Banyan partitions the graph into fine-grained *tablets* and distributes tablets across executors. To handle workload skewness, Banyan dynamically migrates tablets along with the operators accessing them between executors for load balancing. In summary, the contributions of this paper include:

(1) We propose the scoped dataflow model, which introduces a novel scope construct to a dataflow. The scope explicitly exposes the concurrent execution and control of subgraphs in a dataflow to the finest granularity (Section 3).
(2) We build Banyan, an engine for GQS based on a distributed implementation of the scoped dataflow model. Banyan can leverage the many-core parallelism in a modern server, and can easily scale out to a distributed cluster (Section 4).
(3) We conduct extensive evaluations of Banyan on popular benchmarks. The results show that Banyan has 1-3 orders of magnitude performance improvement over the existing graph query engines and provides performance isolation and load balancing.

2 LIMITATIONS OF EXISTING DATAFLOW MODELS

In the dataflow model, a graph traversal query is represented as a directed graph of operators as vertices, where each operator sends and receives messages along directed edges. Figure 1(a) presents a typical logical dataflow for Example 1, where the *repeat* subquery is translated into vertices V_2, V_3 and V_4, the *where* subquery is translated into vertices V_5, V_6 and V_7[1]. Other dataflow models such as Timely [23] and GAIA [32] share the same main structure demonstrated in Figure 1(a). In these dataflow models, their topologies are fixed at compile time, and thus they are referred to as *topo-static* dataflows in the following discussions. Next, we use Example 1 on the data graph in Figure 1(b) to illustrate the limitations of using topo-static dataflows in a GQS.

Control on Concurrent Traversals (O1-1). A graph query usually has subqueries that can launch many independent traversals starting from different vertices. E.g., each sub-tree rooted at P_1, P_2, P_4 in Figure 1(b) is an independent traversal of the *where* subquery. Figure 1(c) demonstrates an example execution pipeline of the *where* subquery of Figure 1(a) in the topo-static dataflow model. Messages of different traversals are marked in different colors and are processed sequentially. Ideally, the red traversal P_2 can terminate right after V_7 processes $\langle P_2, G_2 \rangle$. However, P_2's remaining messages (in red box) cannot be trivially canceled since messages of different traversals are mixed in the pipeline. To cancel a specific traversal, the engine has to annotate each message with extra traversal metadata, and filter messages by their metadata at each operator.

Diverse Scheduling Policies (O1-2). Inside a graph query, different subqueries often have very diverse scheduling preferences. Consider the *repeat* subquery. Instead of blindly exploring all the 5-hop neighbors, a better strategy is to gradually expand the exploration radius, as closer neighbors (e.g., P_1 and P_2 in the first hop) are more likely to work at the same company with the start person. This corresponds to completing the traversals of earlier iterations first, i.e., in a BFS manner. Meanwhile, inside an iteration we prefer to finish checking if a neighbor is a match before the next, i.e., scheduling operators V_2-V_4 in a DFS manner. However, as depicted in Figure 1(d), in the topo-static dataflow model, messages of different iterations are mixed and may be misordered due to

[1]For simplicity, we omit projection operators in the dataflow, and annotate the schema of the message along each edge.

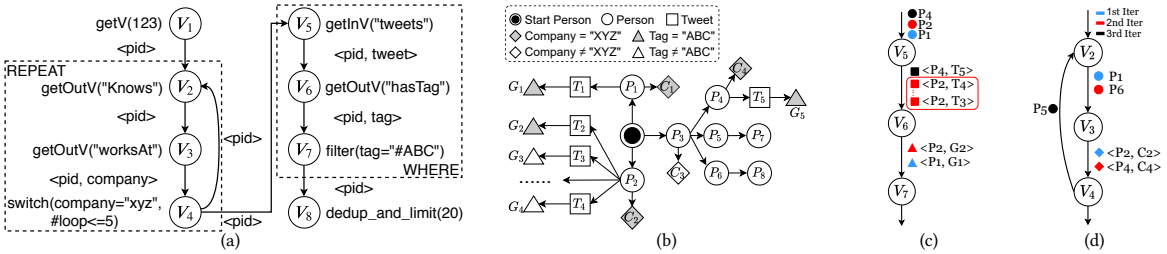

Figure 1: (a) The logical dataflow for Example 1. (b) An example data graph. (c)(d) Execution pipelines illustrating how the *where* and *repeat* subquery in Figure 1(a) process the example data graph in topo-static dataflow model. For simplicity, the execution pipelines only depict a snippet of messages generated during query execution.

parallel execution, e.g., P_5 belonging to the second iteration are processed before P_1 belonging to the first iteration in V_3 (P_5 was emitted by the second iteration). To enforce inter-iteration BFS, one has to annotate each message about which iteration it belongs to, and sort every incoming message according to this metadata in V_2-V_4. As far as we know, no existing graph query engine allows configuring subquery-level scheduling policies.

Performance Isolation (O2). In a GQS, the traversal scales of different queries could vary drastically. Even with the same query, different inputs could lead to traversals of very different scales. E.g., on the LDBC benchmark [19], we observed up to three orders of magnitude difference in query latency for the same query with different starting persons. A subquery traversal with heavy computation may indefinitely block other subquery traversals in the same query. E.g., in Figure 1(c), the traversal of P_2 (who tweeted a lot) blocks the traversal of P_4 (messages in black), even if P_4 can pass the predicate. The above observations reveal the necessity to provide performance isolation in various granularities, from the level of inter-user in a multi-tenant service, to the level of inter-traversal inside a subquery.

3 SCOPED DATAFLOW

Scoped dataflow is a new computational model that extends the existing dataflow model, with the ability to explicitly expose concurrent execution and control of subgraphs in a dataflow to the finest granularity. In this section, we define the structure of scoped dataflow, introduce the programming model, and demonstrate that scoped dataflow can tackle the problems discussed in Section 2.

3.1 Computation Model

Similar to the traditional dataflow model, a scoped dataflow is also based on a directed graph $G(V, E)$. Vertices in V send and receive messages along directed edges in E. The scoped dataflow introduces a new construct named *scope*. Formally, a scope is a sub-structure of the scoped dataflow $G(V, E)$, and has two system-provided vertices: an *ingress vertex* and an *egress vertex*. All the input messages entering a scope pass through its ingress vertex, and all the messages leaving a scope pass through its egress vertex. Inside a scope S, vertices which are neither the ingress nor egress of S are referred to as *internal vertices* of S. An internal vertex of S can belong to an inner scope of S. If $G(V, E)$ is cyclic, every cycle in $G(V, E)$ must be contained entirely within a scope S, and the backward edge must be from a vertex v in S, to the ingress vertex of S. Since the edges leaving a scope must pass through its egress vertex, v cannot be in any inner scope of S. We categorize scopes into two types: a scope without backward edges is called a *branch scope*, and a scope with backward edges is called a *loop scope*. Figure 2(a) shows an example of scoped dataflow. Scopes can be well-nested. The nesting level of a scope S is called its *depth*, denoted as d_S. The depth of a top-level scope is 1.

A scope marks a region inside a dataflow: the dataflow subgraph inside a scope can be dynamically replicated at runtime to create new subgraph instances, isolating the processing of different input data entering the scope. The newly instantiated dataflow subgraph of a scope is called a *scope instance*. The states of vertices in different scope instances are independent. In a scope S, scope instances are instantiated as follows:

- Every inner vertex of S and every edge connecting these inner vertices are copied in the new instance. Note that, the ingress/egress vertices of any inner scopes contained in S are also counted as S's inner vertices. The state of each copied stateful vertex is initialized as the default value.
- Every edge connecting S's ingress/egress vertex and an inner vertex, including backward edges, is copied by replacing the inner vertex with its corresponding copy in the new instance.

Figure 2(b) shows an example of instantiated scoped dataflow with three scope instances for Figure 2(a).

For a scoped dataflow $G(V, E)$, we denote its instantiated scoped dataflow as $\tilde{G}(\tilde{V}, \tilde{E})$. In $\tilde{G}$, each vertex (resp. edge) in a scope instance can be uniquely identified by the corresponding v (resp. e) in G, and a *scope tag* t of the scope instance. The format of *scope tag* is: $ScopeTag : \langle s_1, \cdots, s_d \rangle \in \mathbb{N}^d$, where s_k denotes the s_k-th scope instance in a scope of depth k. The vertex (resp. edge) in $\tilde{G}$ identified by v (resp. e) and the scope tag t is denoted by $\tilde{v}_t$ (resp. $\tilde{e}_t$). Vertices and edges not in any scope have an empty scope tag.

Programming Model. In a scoped dataflow, each message bears the scope tag of the edge it passes through. Every vertex implements the following APIs:

$$v.ReceiveMessage(msg : Message, e : Edge, t : ScopeTag)$$
$$v.OnCompletion(t : ScopeTag)$$

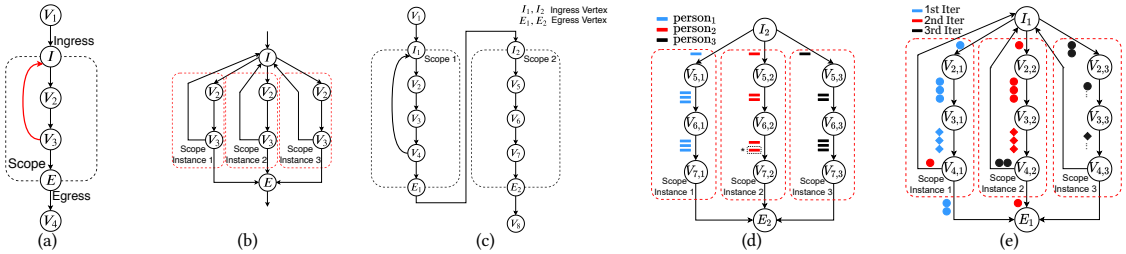

Figure 2: (a) An example loop scope, and (b) an example of its instantiation. (c) The scoped dataflow for Example 1, and an instantiation of its (d) branch scope and (e) loop scope.

A vertex may invoke two system-provided methods in the context of the above callbacks:

$$this.SendMessage(msg : Message, e : Edge, t : ScopeTag)$$
$$this.NotifyCompletion(t : ScopeTag)$$

Users write their logics according to the scoped dataflow G, while at runtime the logics are executed by $\tilde{G}$. E.g, $v.ReceiveMessage(msg, e, t)$ defines the logics how $\tilde{v}_t$ processes a message received along $\tilde{e}_t$. And $v.OnCompletion(t)$ defines the logics executed when vertex $\tilde{v}_t$ has no more input, e.g., an *aggregate* operator emits the final aggregation results when it sees all the inputs. $SendMessage(msg, e, t)$ sends a message along with edge $\tilde{e}_t$. $NotifyCompletion(t)$ can be called if an operator would like to terminate processing proactively, e.g., a *limit* operator terminates once it generates enough outputs.

Scope Instantiation. In a scope, the ingress vertex instantiates scope instances and routes the messages entering this scope to different scope instances. The egress vertex manages the termination of scope instances inside a scope. Each scope instances can be terminated independently. The ingress and egress vertices only act on the scope tags of messages passing through. Specifically, for each message *msg* passing through:

- The ingress vertex routes *msg* to a destination scope instance *SI* and sets the scope tag of *msg* to that of *SI*. If *SI* does not exist, the ingress vertex instantiates it.
- The egress vertex removes the last element in the scope tag of *msg*. When *OnCompletion(t)* is called in the egress, it terminates the scope instance with scope tag t.

Branch and loop scopes have different behaviors on how messages are mapped to scope instances. In a branch scope, every input message triggers the instantiation of a new scope instance. Whereas in a loop scope, messages from edges entering the scope with scope tag $\langle s_1, \cdots, s_d \rangle$ are routed to the scope instance with scope tag $\langle s_1, \cdots, s_d, 1 \rangle$; messages from backward edges with scope tag $\langle s_1, \cdots, s_d, s_{d+1} \rangle$ are routed to scope instance with scope tag $\langle s_1, \cdots, s_d, s_{d+1} + 1 \rangle$. A concurrency threshold, *Max_SI*, can be set to constrain the number of concurrent scope instances in a scope. We conducted experiments to study the overhead of scope instantiation (see Section 5.3).

Scope Scheduling. To fulfill the requirement of **O1-2**, a GQS should be capable of scheduling queries at various granularities, i.e., between the traversals/iterations of a subquery and inside the steps of a single traversal/iteration. Scoped dataflow supports these requirements by allowing customized scheduling policies for scopes. The scheduling policy of a scope can be decoupled into two parts: *inter-scope-instances* (inter-SI) policy and *intra-scope-instance* (intra-SI) policy. The inter-SI policy specifies the scheduling priorities of scope instances inside a scope. The intra-SI policy specifies the scheduling priorities of inner vertices (an inner scope as a whole is treated as a virtual inner vertex) inside a scope instance. Users can customize the scheduling policy of a scope using two comparators:

$$bool\ InterSI\_Comparator(t_1 : ScopeTag, t_2 : ScopeTag)$$
$$bool\ IntraSI\_Comparator(v_1 : VertexID, v_2 : VertexID)$$

The scheduling of a scoped dataflow follows the hierarchy of its internal scopes. Specifically, the priorities of vertices and scopes at the same depth are decided by the intra-SI comparator of the scope they belong to. E.g., the priorities of vertices (V_1, V_8) and scopes (S_1, S_2) in Figure 2(c) are decided by the intra-SI comparator of the "query" scope. Intuitively, when compared with the same-depth vertices, each scope is treated as an indivisible "virtual vertex". Inside a scope, if scope instance SI_i has a higher priority than SI_j, each vertex in SI_i has a higher priority than any vertex in SI_j. For any two vertices $\tilde{u_{t_1}}$ and $\tilde{v_{t_2}}$ in $\tilde{G}$, their scheduling orders are decided iteratively according to the following rules. Without loss of generality, we denote the depths of t_1 and t_2 as d_1 and d_2, and assume $d_1 \leq d_2$. We use $anc_d(\tilde{v_t})$ to denote the ancestor scope instance of $\tilde{v_t}$ at depth d.

- We start comparing the ancestor scope instances of $\tilde{u_{t_1}}$ and $\tilde{v_{t_2}}$ from depth 1 to depth d_1.
- At depth d, if $anc_d(\tilde{u_{t_1}})$ and $anc_d(\tilde{v_{t_2}})$ are the same, we proceed to depth $d+1$.
- At depth d, if $anc_d(\tilde{u_{t_1}})$ and $anc_d(\tilde{v_{t_2}})$ are different scope instances of the same scope S, the priority is determined by calling the inter-SI comparator of S on the tags of $anc_d(\tilde{u_{t_1}})$ and $anc_d(\tilde{v_{t_2}})$.
- At depth d, if $anc_d(\tilde{u_{t_1}})$ and $anc_d(\tilde{v_{t_2}})$ belong to different child scopes in scope S, the priority is determined by calling the intra-SI comparator of S on the tags of $anc_d(\tilde{u_{t_1}})$ and $anc_d(\tilde{v_{t_2}})$.

3.2 Progress Tracking

To correctly invoke $OnCompletion(t)$ for vertices, a scoped dataflow needs to track the processing progress of its vertices, i.e., when a vertex is guaranteed to have received all its inputs. The scoped dataflow model adopts an EOS-based progress tracking mechanism

inspired by the Chandy-Lamport algorithm [7]: after ingesting all the external inputs, the runtime automatically inserts an EOS message to the dataflow. EOS messages are propagated through the dataflow graph. Once a vertex receives EOS messages from all its incoming edges, it calls *OnCompletion*(t), and then emits an EOS message in all its outgoing edges. However, as a scoped dataflow can dynamically instantiate scope instances and may contain cycles, we extend the aforementioned EOS-based progress tracking mechanism in order to support the scoped dataflow.

Hierarchical Progress Tracking. We track progress hierarchically in a scoped dataflow, i.e., progress tracking inside and outside a scope are conducted separately. Progress tracking outside a scope S simply treats S as a virtual vertex, denoted as v_S. The runtime conducts progress tracking on the dataflow subgraph inside a scope S to decide the completion of v_S, which completes when the ingress vertex of S receives EOS from all its input edges, and all the scope instances inside S have completed. Once v_S reaches completion, the egress vertex of S emits EOS along all its outgoing edges.

If a scope has nested sub-scopes, the runtime treats its top-level sub-scopes as virtual vertices during progress tracking. Take the scoped dataflow in Figure 2(c) as an example. Treating scopes S_1 and S_2 as virtual vertices, i.e., V_{S_1} and V_{S_2}, removes cycles from the dataflow. For both V_{S_1} and V_{S_2}, it is their ingress vertices that receive EOS from the upstream neighbor and their egress vertices that emit EOS to the downstream neighbor. After I_1 in V_{S_1} receives EOS from V_1, it is aware that V_{S_1} has received all the inputs. Once V_{S1} finishes processing the inputs, E_1 in in V_{S_1} calls *OnCompletion*(t) and emits EOS for V_{S_2}, which further propagates the EOS as V_{S_1} does. Next, we explain how the progress tracking is done inside the branch scope and loop scope, respectively.

Tracking inside a Scope. In a branch scope, the runtime propagates EOS in each scope instance to track their progress independently. An ingress vertex reaches completion when it has received EOS from all its incoming edges. Once completed, the ingress vertex sends the number of scope instances it has spawned to the egress vertex of the same scope. When the egress vertex has tracked that all the scope instances in this scope have completed, it reaches completion and emits EOS to the outgoing edges.

In a loop scope, all the messages sent from iteration i to $(i+1)$ will pass the ingress along the backward edges. Note that the last loop iteration sends no data message but only EOS to the ingress. If an ingress only receives EOS but no data message from an iteration (scope instance), it can infer that this is the last loop iteration. This way, the ingress can infer the number of spawned scope instances. With this information, the egress vertex tracks the progress of scope instances in the same way as the branch scope. This process is guaranteed to be able to stop due to the simple fact that if you remove the ingress vertex inside a loop scope (note that the ingress vertex only forwards messages) and directly connect the edges according to the forwarding behavior of the ingress vertex, the instantiated scope dataflow is a DAG without cycles.

3.3 Scoped Dataflow in Action

In this section, we discuss the applicability of scopes, followed by an example explaining how the scoped dataflow model can be used to solve the challenges discussed in Section 2.

Applicability of Scopes. The scoped dataflow model is designed to facilitate fine-grained control on subquery traversals and enforce scope-level customization of scheduling policies. Concretely, scoping can benefit graph queries with:

(1) *where* subqueries through early cancellation. For those *where* subqueries that cannot be early canceled, scopes should be turned off since the instantiation of scope instance brings overhead (see more details from **E2** in Section 5.3).
(2) *loop* subqueries that can find matches more quickly following certain exploration strategy (e.g., BFS or DFS).

Besides, scope can serve as a resource container that guarantees performance isolation. From **E2** in Section 5.3 we can observe that the benefits of scope depend on the queries and the data. Further optimizations such as creating an optimal query plan can be done by the query compiler, which is beyond the scope of this paper.

Example 2 (Implementation of vertex V_7 in Figure 2(d))**.**

```
class V7Filter : Vertex {
  void ReceiveMessage(msg:Message, e:Edge, t:ScopeTag) {
    if (msg.GetTag() == "#ABC") {
      SendMessage(msg.getPersonId(), out_e, t);
      NotifyCompletion(t);}
  }
  void OnCompletion(t : Tag) { }
}
```

Example 3 (Implementations of inter-SI BFS and intra-SI DFS)**.**

```
bool InterSI_BFS:InterSI_Comparator(t_1:ScopeTag, t_2:
    ScopeTag) {
  return LexicalOrderCompare(t_1, t_2);
}
bool IntraSI_DFS:IntraSI_Comparator(v_1:VertexID, v_2:
    VertexID) {
  /* Assuming VertexID increments in topological order */
  return v_1 < v_2;
}
```

Examples of Scopes. Figure 2(c) shows the scoped dataflow for Example 1. Figure 2(d) zooms in the *where* subquery of Figure 2(c), and demonstrates instantiations of scope 2. Traversals triggered by different users entering the *where* subquery are mapped to different scope instances, which can be executed concurrently and controlled independently. This way, a user (e.g., the blue one) who posts many tweets without the specified tag will not block the progress of other users (e.g., the red and black ones) entering the *where* subquery.

Example 2 shows the implementation of the *filter* vertex v_7 in Figure 2(d), which enables early cancellation: On receiving a message *msg*, if the tag in *msg* is a match, the vertex notifies the completion of itself by calling *NotifyCompletion*(t) to trigger its cancellation. When a match (the red message marked in black box entering $v_{7,2}$) is found, the corresponding scope instance ($v_{5,2}, v_{6,2}, v_{7,2}$) can be canceled without impacting the other scope instances.

The scheduling policies of scopes can be flexibly configured. Figure 2(e) shows that iterations of the *repeat* subquery in Figure 2(c) are mapped into different scope instances of the loop scope. We configure a **BFS** inter-SI scheduling policy such that the blue scope instance (the first iteration) is executed first, then the red one (the second iteration), and the black one (the third iteration) as the last. Meanwhile, by enforcing a **DFS** intra-SI policy, vertices inside the blue scope instance are scheduled in the order of $\tilde{v}_{4,1}$, $\tilde{v}_{3,1}$ and $\tilde{v}_{2,1}$. Implementations of the inter-SI BFS and intra-SI DFS policy are presented in Example 3.

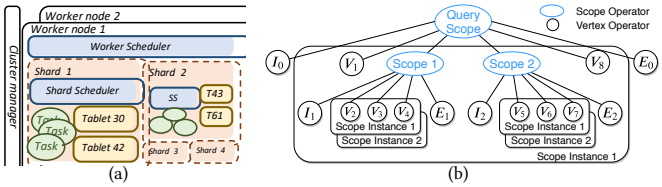

Figure 3: (a) The architecture of Banyan. (b) An operator tree mapped from the dataflow in Figure 2(c) in an executor.

4 BUILDING BANYAN ON SCOPED DATAFLOW

We build Banyan, an engine for GQS based on a distributed implementation of the scoped dataflow model. Banyan is designed to efficiently leverage the many-core parallelism of modern servers, and agilely balance the workloads across cores. Banyan can also scale out to a distributed cluster. The overall system architecture of Banyan is presented in Figure 3(a). A Banyan cluster consists of a group of worker nodes, each of which manages multiple executors. An executor exclusively runs in a system thread pinned in a physical core, and is in charge of a graph partition. Vertices in a scoped dataflow are parallelized into operators, and are mapped to executors. Executors communicate with each other through message queues inside the worker node and through TCP connections across worker nodes. In a worker node, executors schedule their own operators and are scheduled by the worker scheduler. The worker scheduler is responsible for balancing the workloads across executors in a worker node.

4.1 Parallelizing A Scoped Dataflow

Banyan parallelizes a dataflow G into a physical plan of operators in a data-parallel manner. Every vertex in G is parallelized into a set of operators, each encapsulating the processing logic of this vertex. Each edge in G has a partitioning function controlling the data exchange between the operators. In the physical plan, the vertices in G are replaced with the corresponding set of operators, and the edges in G are replaced with a set of edges connecting these operators. At runtime, when we instantiate a scope in G into scope instances, every scope instance inherits the physical plan of the scope. That is, every scope instance has a complete copy of the corresponding operators and edges of its scope.

Banyan partitions the graph into tablets distributed across executors. A tablet contains an exclusive set of graph vertices and all their in/out edges, along with their properties. We refer to vertices in G that need to access graph data as *graph-accessing* vertices. To exploit data locality, a graph-accessing vertex is parallelized into as many operators as the number of tablets, and are mapped to the executors hosting the tablets.

4.2 Executor Internals

Executors are single-threaded and each is pinned in a physical core. Executors schedule their operators cooperatively, allowing Banyan to concurrently process a large number of operators without facing the bottleneck caused by context switch. Cooperative scheduling is based on an asynchronous task-based programming interface. E.g., an operator blocked by an asynchronous I/O operation automatically yields, and its executor schedules another operator ready for execution. This way, Banyan can overlap CPU computation with networking and I/O to improve the resource utilization.

Scope Operator. To facilitate the scope-based scheduling in Banyan (see Section 3.1), we introduce *scope operators* to manage the creation, termination and scheduling for all the operators of a scope. On every executor containing operators of a scope S, we create a scope operator managing all the local operators of S in this executor. The scope operator of S is also managed by the scope operator of S's parent scope. This way, all the operators in an executor are managed as a forest of operator trees. In each tree, the leaf nodes are operators of vertices and the non-leaf nodes are the scope operators. Figure 3(b) shows the operator tree mapped from the scoped dataflow in Figure 2(a). The operators in an executor are scheduled hierarchically: (1) The executor schedules the root scope operators. (2) When a scope operator of scope S is scheduled, it further schedules its child (scope) operators following S's inter-SI and intra-SI scheduling policies.

Banyan enforces performance isolation in each single executor. Scope operator is the basic unit of resource allocation: resources allocated to scope operators at the same depth are isolated, and an operator can only consume the resources allocated to its parent scope operator. Once scheduled, an operator is assigned a *quota CPU time* by its parent, which constrains the maximum amount of CPU time this operator can use at this round of scheduling. The vertex operator updates its quota after processing a message, and yields once the quota is used up. By modeling different queries or tenants as the top-level scopes, Banyan can naturally support performance isolation across queries or users.

4.3 Hierarchical Operator Management

In this subsection, we introduce how operators are addressed, created and terminated in Banyan.

Operator Addressing. In Banyan, each operator has a unique address, encoding the path from the executor to this operator in the operator tree. The address consists of three parts:

$$\langle exec\_id,\ (sop\_id_1, s_1), \cdots, (sop\_id_d, s_d),\ op\_id \rangle$$

where $exec\_id$ identifies the hosting executor of the operator; op_id represents the ID of the operator; $(sop\_id_1, s_1), \cdots, (sop\_id_d, s_d)$ denotes the chain of *ancestor scope operators* (sop_id_k) and the corresponding scope instance IDs (s_k). Actually, $\langle s_1, \cdots, s_d \rangle$ is the scope tag of the operator.

To facilitate hierarchical scheduling, each scope operator maintains a directory of its child operators as a prefix tree, using the addresses of child operators as the keys and the pointers to these operators as the values. In the prefix tree, the operators of a scope instance are naturally grouped together as they share the same prefix in their addresses, and thus can be quickly located.

Operator Creation and Termination. Operator creation is event-driven in Banyan. Sending a message to a non-existing operator triggers the creation of this operator and all its non-existing ancestor scope operators. A scope operator provides a system-level API *TerminateScope(scope_instance_id)* to terminate a scope instance. Terminating a scope operator will terminate all its managed scope

Table 2: Statistics of LDBC datasets

Dataset	#Vertices	#Edges	CSV Size
LDBC-1	3, 181, 364	17, 299, 165	882M
LDBC-100	282, 637, 871	1, 777, 459, 239	88G

Table 3: Statistics of SE datasets

Dataset	#Vertices	#Edges	d_{max}	d_{avg}	CSV Size
LJ	4, 847, 571	43, 369, 619	20, 333	17.9	$464M$
OR	3, 072, 441	117, 185, 083	33, 313	38.1	$1.7G$
FS	65, 608, 366	1, 806, 067, 135	5, 214	27.5	$31G$

instances in cascade. Messages sent to terminated operators are ignored. Banyan recycles objects used for operators through memory management to avoid excessive memory allocations.

4.4 Parallelizing Progress Tracking

As explained in Section 3.2, progress tracking inside a scope requires the ingress vertex to notify the egress vertex the number of scope instances. When the ingress and egress vertices of a scope are parallelized, a single ingress operator may not be aware of all the scope instances in this scope. To tackle this problem, in a branch scope, each ingress operator broadcasts to all the egress operators the largest ID of scope instances it has instantiated. An egress operator takes the maximum among these IDs as the total number of scope instances to track. In a loop scope, every time when an ingress operator sees only the EOS messages but no data message from a specific loop iteration, it broadcasts the ID of this scope instance to all the egress operators. If an egress operator receives a specific scope instance ID from all the ingress instances, it can conclude that this ID equals the number of scope instances.

To reduce the cost of operator instantiation, Banyan skips creating operators that only receive EOS messages. EOS messages sent to non-existing operators are buffered in their parent scope operator. If the operator is created later, these buffered EOS messages are inserted into their mailboxes. Otherwise, the parent scope operator emits EOS messages on behalf of the non-existing child operator after receiving all the corresponding EOS messages.

4.5 Load Balancing

Realistic graphs are often scale-free, which may lead to a skewed workload distribution among different tablets. And this skewness changes dynamically, as the graph accessing patterns of the incoming queries continuously change. In graph queries, graph-accessing operations are often the most costly part during the execution. To facilitate load balancing between executors, we deliberately partition the graph into more tablets, and migrate tablets together with their graph-accessing operators across executors. Upon migrating a tablet, as graph traversal queries are usually short-lived, we do not migrate the executing operators of existing queries, but only redirect the incoming queries.

5 EVALUATIONS

We evaluate the performance of Banyan in the following aspects:

- (**E1**) We study the overall performance of Banyan by comparing single-query latency with state-of-the-art graph query engines (Section 5.2).
- (**E2**) We study the effects and overheads of scopes on query performance, by comparing the scoped dataflow with the Timely dataflow model (Section 5.3).
- (**E3**) We study how well Banyan can scale up in a many-core server and scale out in a distributed cluster (Section 5.4).
- (**E4**) We study how well Banyan can enforce performance isolation and load balancing (Section 5.5).
- (**E5**) We study the performance of Banyan on subgraph enumeration workloads (Section 5.6).

5.1 Experiment Setup

Benchmarks. We use three benchmarks in the experiments: the LDBC Social Network Benchmark [2, 19], the Complex Query(CQ) benchmark and the subgraph enumeration (SE) benchmark.

LDBC is a popular benchmark of graph traversal queries. We selected 12 queries (IC_1 - IC_{12}) from the 14 *Interactive Complex Read* queries in the LDBC benchmark. IC_{13} and IC_{14} are excluded as they both have shortest-path subqueries, which are typical graph analytic queries. We use two LDBC datasets with scale factor 1 and 100, denoted as LDBC-1 and LDBC-100. Table 2 shows the statistics of these two datasets. For each query on both datasets, we use the LDBC generator to generate 50 parameters.

Real-world service scenarios (e.g. search engines) often select the top-k results from a limited size of recalled candidates to guarantee interactive response. This is different from the query patterns in LDBC (all LDBC queries require sorting the entire results). To better study the effects of scopes, we compose the CQ benchmark with 6 queries (CQ_1 - CQ_6 in Appendix A) by adjusting the LDBC queries, e.g., removing the *sort* operator. Each CQ query has 10 parameters generated by the LDBC benchmark for both datasets.

The SE benchmark consists of two subgraph patterns (Figure 8(a) and Figure 8(b)) and three real-world datasets: *LJ*, *OR* and *FS* obtained from [38]. Table 3 lists the statistics of these datasets.

System Configurations. All the experiments are conducted on a cluster (up to 8 machines) where each machine has 755G memory and 2 Intel Xeon Platinum 8269CY CPUs (each with 26 physical cores and 52 hyper-threads).

For interactive graph traversal queries (LDBC and CQ benchmarks), we choose four baseline systems from the most popular or latest graph databases/engines: two single-machine ones—Neo4j 4.1.1 [26] and JanusGraph 0.5.0 [15], as well as two distributed ones—TigerGraph 3.1.0 [39] and GAIA [32]. For subgraph enumeration (SE benchmark), we choose Huge [44], a state-of-the-art subgraph enumeration system, as the baseline. We also compare scoped dataflow with Timely dataflow [23] to study the effects of scopes on query performance. Queries in the Timely dataflow model are implemented using Banyan with scopes turned off.

Unless explicitly explained, all the experiments are conducted in a container which has 32 cores and 700G memory. We configure a cache size (if available) large enough to store the entire datasets. We build the same set of indexes for all systems, i.e., a primary index on vertex ID for each type of vertices. In graph databases, we execute queries without transactions or as read-only transactions to

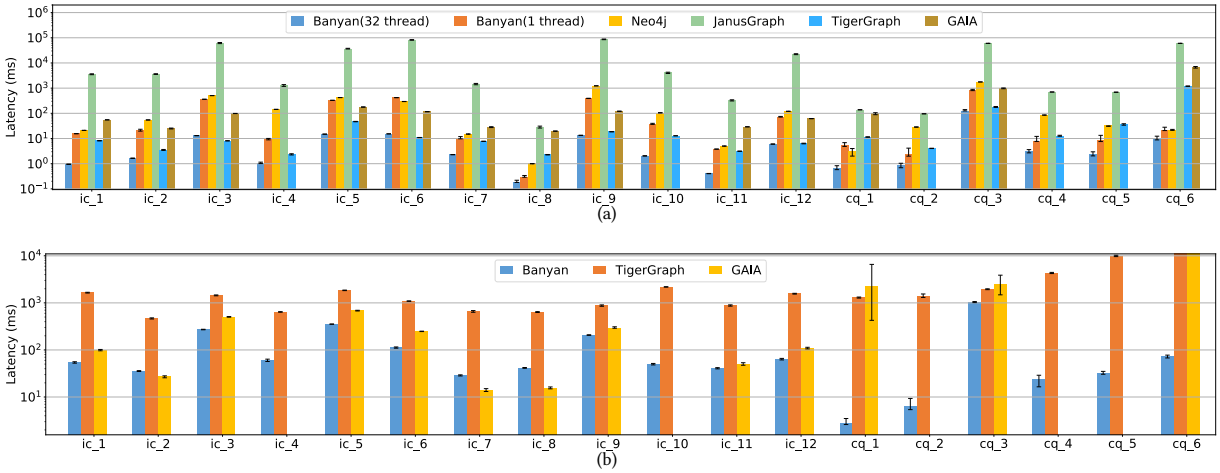

Figure 4: (a) Query latency on a single machine with 32 cores. (b) Query latency in a 4-node cluster (each node has 8 cores).

minimize the transaction overhead. Configurations of the baselines are as follows:

- *Neo4j 4.1.1.* We use 32 worker threads and turn on query cache.
- *JanusGraph 0.5.0.* We use BerkeleyJE 7.5.11 [29] as the storage.
- *TigerGraph 3.1.0.* The distributed query mode is used in the distributed experiments. TigerGraph requires installing a query before execution, and the installation takes much more time (more than 1 min on average) than query execution. We exclude the installation time in the reported results.
- *GAIA* is only experimented on a subset of the LDBC and CQ queries, as it does not support IC_4, IC_{10}, CQ_2, CQ_4 and CQ_5 [2]. The number of workers is set to 32 in GAIA.
- *Huge* is only experimented on the SE benchmark. The graphs are stored in the format of compressed sparse row (CSR) in memory.
- *Banyan.* We use a C++ version of the backend storage used by JanusGraph (BerkeleyDB 18.1.32 [30]), and directly import the databases exported from JanusGraph. For each dataset, Banyan randomly partitions the graph into 64 tablets. We apply loop scope on all the *repeat* subqueries and branch scope on *where* subqueries whose branches can be early canceled. Unless otherwise specified, we use 32 executors for query execution. In the SE benchmark, Banyan uses the same in-memory CSR to store graphs as Huge does.

Experiment Methodology. To control query submission, we extend the LDBC client to allow specifying the number of concurrent queries (W) a client can submit. Once started, the client tries to submit as many queries as possible, but under the constraint of W. If the threshold of W has been reached, a new query will only be submitted after the completion of a previous one. Unless explicitly specified, we use $W = 1$ throughout the experiments. As LDBC/CQ queries are templates, unless otherwise specified, we follow the LDBC benchmark convention and for each query report the average latency of all the parameters. Throughout this section, for each data point we run the corresponding experiment 10 times to warm up the system, and collect the results from the following 10 runs. We report the minimum, maximum, and average values of the 10 results in the figures. Traversal queries that run longer than 60 seconds are marked as timeout.

[2]The authors of GAIA confirmed that their compiler is still under development and cannot support some Gremlin operators like *sideEffect* and *store*.

5.2 Overall System Performance

In this section, we study the performance of Banyan by comparing its single-query latencies on both the LDBC and CQ benchmarks with baseline systems.

Single-machine. In this experiment, we use the LDBC-1 dataset so that baselines like JanusGraph can finish most queries before timeout. The results are depicted in Figure 4(a).

On the LDBC queries, Banyan has 5X to three orders of magnitude latency improvement over all the baseline systems except for TigerGraph. Banyan outperforms TigerGraph by up to 14X for 10 out of the 12 LDBC queries (including all the large queries), and is slightly slower on IC_3 and IC_6 (both are small queries). As TigerGraph is not open-sourced, we cannot analyze how the query installation helps in query execution. This results demonstrates that Banyan can well utilize the many-core parallelism.

On the CQ queries which can benefit more from the scoped dataflow, the advantage of Banyan further widens, e.g., up to 130X and 726X faster than TigerGraph and GAIA, respectively. This improvement is led by the fine-grained control and scheduling enabled by scoped dataflow: (1) subquery traversals (*where* subqueries in CQ_3, CQ_4, CQ_5 and CQ_6) can be early canceled, and (2) customized scope-level scheduling policies (e.g., DFS in the loop of CQ_1 and BFS in the loop of CQ_2) can trigger the (sub-)query cancellation earlier. GAIA mixes messages of different "context" in the same execution pipeline, and thus cannot efficiently perform traversal-level early cancellation. This inefficiency is reflected by the performance of GAIA on CQ_1 and CQ_6. In addition, GAIA cannot configure scope-level scheduling policies, which influences its performance on CQ_3 and CQ_6. As JanusGraph and Neo4j cannot parallelize queries starting from a single vertex, we also present

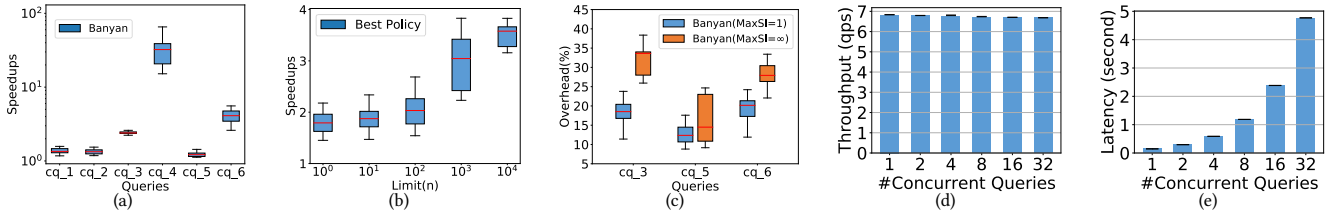

Figure 5: (a) The per-parameter latency speedups of scoped dataflow compared to Timely. (b) The per-parameter latency speedups of the *best-case* scheduling policy compared to the FIFO policy. (c) The overheads of scope instantiation when MAX_SI is set to 1 and unlimited. The (d) throughput and (e) latency of Banyan under different numbers of concurrent queries.

the single-thread performance of Banyan, which outperforms the both systems on all queries.

Distributed. For the distributed experiment, we use the LDBC-100 dataset and a cluster of 4 worker nodes each hosting a container of 8 cores. As Figure 4(b) shows, Banyan outperforms TigerGraph by 5X to 40X on the LDBC benchmark, and 2X to three orders of magnitude on the CQ benchmark. These results are consistent with the single-machine experiment, as the benefits of efficiently utilizing hardware parallelism and the scoped dataflow are still tenable in the distributed environment. Banyan is faster than GAIA on most of the LDBC queries and all the CQ queries, except for three small queries (IC_2, IC_7 and IC_8). This is because the graph partitioning in Banyan can incur some overhead due to message passing between executors.

5.3 Benchmark on Scoped Dataflow

In this section, we use the CQ benchmark on the LDBC-100 dataset to study: (1) the effects of scoped dataflow, (2) the effects of scheduling policy and (3) the overhead of scope instantiation. We run each query with ten different parameters, compute the speedup(overhead) between different competitors on each parameter, and report the boxplot of all the per-parameter speedups(overheads).

Effects of Scoped Dataflow. We evaluate the speedups of scopes by comparing it against Timely. Figure 5(a) shows that the effects of scoped dataflow are:

- Query-dependent. On average, the scoped dataflow brings 1.3X to 36X latency improvement compared to Timely. The speedup on CQ_1 is relatively small, as CQ_1 has no subquery which can be early canceled, and its speedup mainly comes from scope-level scheduling policy, i.e., using DFS in the loop subquery. On the other hand, as CQ_4 contains a *where* subquery nested with a loop subquery, canceling a *where* traversal saves a huge amount of computation.
- Data-dependent. The speedup of scoped dataflow varies on different parameters, e.g., 14X to 86X on CQ_4. This is because different traversals of the *where* subquery in CQ_4 have very different costs.

The optimal scoped dataflow plan should be determined by a query compiler according to the query structure and data statistics.

Effects of Scheduling Policy. In this experiment, we select CQ_6 and compare the latency of CQ_6 in Banyan between two cases: (1) the *best-policy* case where the intra-SI policy of the query and the *where* subquery are both set to DFS, and (2) the *FIFO* case where all the scheduling policies are set to FIFO in the query. We vary n in CQ_6's *limit(n)* clause from 1 to 10^4. Figure 5(b) shows the boxplots of speedups brought by *best-policy* over *FIFO*. By increasing the value of n from 1 to 10^4, the speedup of *best-policy* widens from 1.8X to 3.5X. This is because *FIFO* schedules more wasted traversals(no final output), and this wastage becomes worse when the number of total traversals increases. Similar effects can be observed in other CQ queries and thus omitted. This experiment shows that customized scheduling policy is necessary for graph queries.

Overhead of Scope Instantiation. To quantify the overhead of scope instantiation, we turn off early cancellation in Banyan, use purely FIFO for scheduling, and compare its single query latencies with Timely. We experiment on CQ_3, CQ_5 and CQ_6, as these queries contains *where* subqueries that can instantiate a large number of scope instances. We remove the *limit* clause in these queries such that Banyan and Timely perform the same number of traversals.

Figure 5(c) shows that, without limiting MAX_SI, Banyan is on average 25% slower than Timely, as Banyan suffers from extra scheduling overheads among SIs and a high memory pressure. Setting MAX_SI in Banyan to 1 shrinks the gap to 13%. Note that MAX_SI is an executor-local configuration. With 32 executors running in parallel, setting MAX_SI to 1 allows Banyan to instantiate at most 32 concurrent SIs of a scope, which is enough to saturate the multi-core parallelism. This experiment shows that the overhead of scope instantiation is limited compared with the benefits of scopes as shown in the first experiment of **E2**.

5.4 Scalability of Banyan

In this set of experiments, we use both the CQ and LDBC benchmarks with the LDBC-100 dataset.

Scale-Up. In this experiment, we use a single container and vary the number of cores from 1 to 64, and report the query latency of Banyan in Figure 6(a). We can see that the query latency scales almost linearly up to 32 cores. This is because Banyan can efficiently parallelize a scoped dataflow into fine-grained operators and evenly distribute them across executors. The performance improvement of Banyan stagnates when scaling up from 32 cores to 64 cores, as the per-executor workload becomes too small to fully utilize the computation resources, and the hyper-threading has a negative impact on the cache locality.

Scale-Out. In Figure 6(b), we study the scale-out performance of Banyan. We use a container of 8 cores, and increase the number of containers from 1 to 8. By increasing the number of containers, the latencies of large queries (e.g., IC_3, IC_5, IC_6 and IC_9) decrease in

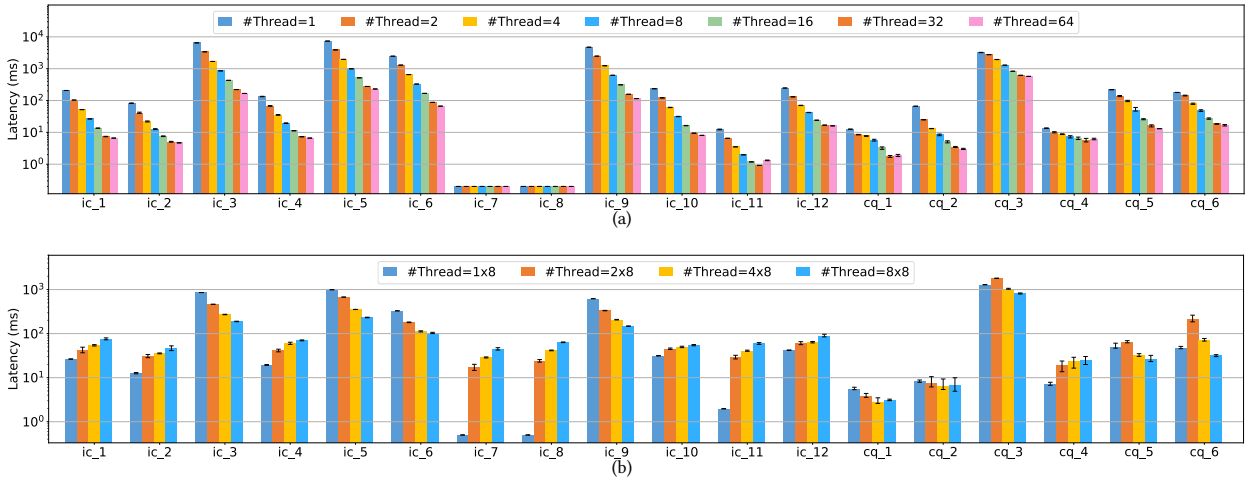

Figure 6: (a) The scale-up performance of Banyan with increasing number of executors on a single machine. (b) The scale-out performance of Banyan with increasing number of worker nodes in the cluster.

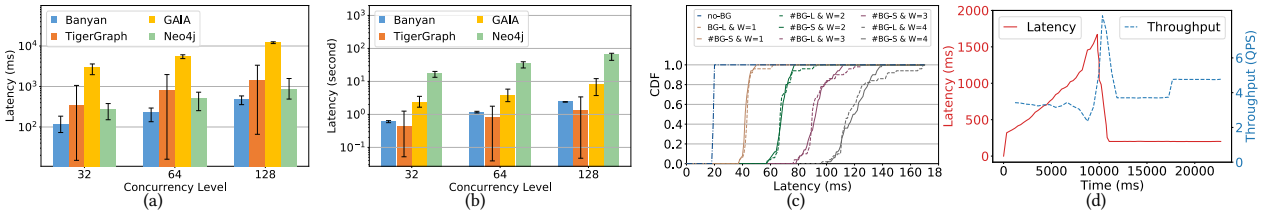

Figure 7: The latency of Banyan, TigerGraph, GAIA and Neo4j under different concurrent workloads: (a) the latency of small queries and (b) the latency of large queries. (c) The latency CDF of the foreground query under different background workloads. (d) The throughput and latency of Banyan before and after load balancing.

a nearly linear manner. This shows that the scoped dataflow can be well parallelized in a distributed environment. For small queries (e.g., IC_1, IC_2, IC_{10}, IC_{11} and CQ_4) with limited computation to distribute, scaling out results in slightly worse query latency due to the increased network communication cost.

Scalability with Concurrent Queries. In this experiment, we use IC_6 with a fixed parameter to isolate the latency difference caused by different queries and parameters. We vary W, the submission concurrency of the client, from 1 to 32. Figure 5(d) and 5(e) show Banyan's throughput and latency. As shown in Figure 5(d), by increasing the number of concurrent queries, Banyan can provide a stable throughput (less than 2% throughput decrease when $W = 32$). The query latency is increasing linearly with more concurrent queries executing in the system (Figure 5(e)). These results clearly show that Banyan can fairly allocate the CPU time among concurrent queries, and incur little overhead.

5.5 Performance Isolation & Load Balancing

In this subsection, we study the performance isolation and load balancing in Banyan. In the following experiments, we use a fixed parameter for each selected query.

Performance Isolation. First, we compare the performance isolation in Banyan, Neo4j, TigerGraph and GAIA under the LDBC benchmark using LDBC-1. We simulate a mixed workload of large queries (IC_9) and small queries (IC_1), and vary the submission concurrency W from 32 to 128. For each W, $\frac{W}{2}$ large queries and $\frac{W}{2}$ small queries are concurrently executing in the system. The latencies of small and large queries are depicted in Figure 7(a) and Figure 7(b) , respectively. The results show that Banyan provides on average 2X-23X better latencies for small queries compared with the baselines (Figure 7(a)), and 3X-30X performance boost on large queries compared with GAIA and Neo4j (Figure 7(b)). TigerGraph runs 50% faster than *Banyan* on the large queries, but sacrifices its performance on small queries (on average 3.3X worse than Banyan). Note that for each W, small queries and large queries are executing concurrently in the system. As TigerGraph cannot well isolate the resource consumption among concurrent queries, the progress of small queries are heavily blocked by the large queries. The stability of query latency in TigerGraph is also much worse than Banyan. In Figure 7(a), when W is set to 128, the latency of TigerGraph fluctuates at a range of nearly 3000ms on small queries. In comparison, Banyan only fluctuates within 50ms.

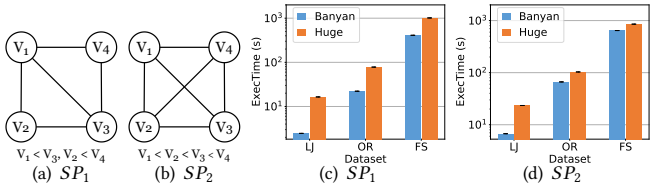

Figure 8: Subgraph patterns (a) SP_1 and (b) SP_2; The performance of Banyan and Huge on (c) SP_1 and (d) SP_2.

In the second experiment, we conduct a controlled experiment on how background workload impacts a foreground query. We generate a background workload of different queries using the LDBC-100 dataset, i.e., of purely large queries (IC_9) or purely small queries(IC_{10}), and vary the concurrency W from 1 to 4. While the background queries are executing, we submit a small foreground query (IC_1). For each background configuration, we submit the foreground query for 100 times and plot the CDF of the query latency in Figure 7(c). We also plot the foreground latency without any background workload (No-BG). When W is fixed, the latency of the foreground query is quite stable. Changing the background workload from small queries (BG-S) to large queries (BG-L) has negligible impact on the latency of the foreground query. E.g., the 95% percentile latency of the foreground query is only increased by 3.5% (resp. 9.5%) for $W = 1$ (resp. $W = 4$).

The above two experiments show that Banyan can enforce performance isolation so that large queries will not block small queries.

Load Balancing. We simulate a skewed workload to evaluate load balancing in Banyan. The 64 tablets of LDBC-100 dataset are distributed among 8 executors in a skewed manner: evenly distributing 48 tablets on 4 executors and the rest on the others. We repeatedly submit IC_6 every 270ms. At t_1 (around 9, 000ms), we re-balance the distribution of tablets such that each executor has 8 tablets. At t_2 (around 17, 000ms), we set the submission interval to 210ms. The latency and throughput are reported in Figure 7(d). We can see the query latency continuously increases when the workload is skewed. After the re-balance (t_1), the query latency immediately drops, and restores to a stable level (around 200ms) after 3000ms. The throughput first bursts as Banyan is clearing the buffered work. After increasing the input rate at t_2, the throughput increases to nearly 1.3X of the initial level, while the latency remains stable.

5.6 Subgraph Enumeration

In this experiment, we compare Banyan and Huge on the SE benchmark. The both systems are deployed in a cluster of two 8-core containers. As SP_1 and SP_2 have no subquery structures, scopes are turned off in Banyan. As Figure 8 shows, Banyan outperforms Huge by 2.2X on query execution time. This is because Banyan can well leverage the many-core parallelism. Parallelizing subgraph enumeration queries in the distributed environment requires transferring huge amount of data. Huge optimizes this cost using its hybrid push/pull mechanism. Differently, Banyan adopts the push-based model to optimize latencies for interactive graph traversal queries.

6 RELATED WORK

Graph Databases and Graph Engines Neo4j [26], Neptune [27], TinkerPop [40] and JanusGraph [15] are single-machine graph databases. TinkerPop, JanusGraph and Neptune only utilize multiple threads for inter-query parallelization [16, 28, 41]. Neo4j can only parallelize top-level traversals starting from different vertices inside a query. Distributed graph databases [10, 31, 39] are mainly optimized for OLTP queries. Differently, Banyan focuses on read-only graph traversal queries. Grasper [8] proposes the Expert model to support adaptive operator parallelization. As Grasper uses the same set of experts for concurrent queries, it cannot support fine-grained control and performance isolation as Banyan does. GAIA [32] introduces virtual *scope* to facilitate data dependency tracking in graph queries, such that nested subqueries can be correctly parallelized. Note that their scope is a logical concept to annotate subquery traversals. GAIA compiles queries into topo-static dataflows, and thus suffers the limitations discussed in Section 2.

G-SPARQL [34], Trinity.RDF [45] and Wukong [36] target for SPARQL queries. G-SPARQL [34] uses a hybrid query execution engine that can push parts of the query plan into the relational database. Trinity.RDF [45] utilizes graph exploration to answer SPARQL queries. Wukong [36] supports concurrent execution of sub-query. The subquery in Wukong [36] is generated *statically* and is of *coarser* granularity than scope instances, and thus cannot support goal **O1** (see Section 2) required by a GQS.

Graph analysis systems like Pregel [21], PowerGraph [11, 20] and GraphX [12] are all inspired by the BSP model. These systems often access the entire graph for multiple iterations to answer a query. Differently, interactive graph traversal queries usually start from one or a handful of vertices, traverse a few hops and only access a very limited part of the graph. The per-iteration global synchronization in the BSP model makes it difficult to fulfill the stringent latency requirement of interactive graph traversal queries.

Dataflow Engines. Dataflow systems such as [4, 5, 14] adopt BSP paradigm, and do not support cycles. Flink [3] supports cycles but requires barrier synchronization between loop iterations. Naiad [23] proposes the Timely dataflow model for iterative and incremental computations, but cannot support branch scopes for *where* subqueries as Banyan does. All the above dataflows are topo-static dataflow models. Cilk [6] and CIEL [24] support dynamic dataflows, but only at the level of coarse-grained tasks. This design may incur huge overhead to control nimble tasks like scope instances in scoped dataflow. Compared to existing dataflow models, the scoped dataflow model allows concurrent execution and independent control on the replicated execution pipelines, and provides a new way to support loops in dataflow, allowing users to control the scheduling policy in cyclic computation.

Subgraph Matching Systems. Subgraph queries usually have no subquery structure and cannot be canceled early. RapidMatch [37] compares the exploration-based and join-based approaches on subgraph queries, and proposes a join-based engine, utilizing graph structural information for filtering and plan generation. [18] implemented a set of representative algorithms and optimizations to study their effectiveness. Huge [44] is a join-based subgraph enumeration system. It considers the variance of join algorithms and communication modes (pull or push) to optimize the query plan.

Huge uses an adaptive BFS/DFS scheduler to improve the resource utilization under memory constraints. Differently, Banyan proposes scoped dataflow to support fine-grained control and scheduling for latency-sensitive graph traversal queries. All these systems do not consider performance isolation for concurrent query execution.

Pull vs. Push-Based Query Engines. In pull-based engines [33, 42], operators pro-actively request data from their predecessors. This eases the control logics like backpressure. In push-based engines [1, 17], operators push their results to the destination operators, which leads to a simpler control flow with less loops and more opportunities for sharing and pipelining the intermediate results. Banyan is an event-driven engine. The instantiation and scheduling of operators are driven by the outputs of their predecessors. Therefore, Banyan can only adopt the push-based model.

7 CONCLUSIONS AND FUTURE WORK

We present a novel scoped dataflow model, and a new engine named Banyan built on top of it for GQS. Scoped dataflow targets at solving the need for sophisticated fine-grained control and scheduling in order to fulfill stringent query latency and performance isolation in a GQS. We demonstrate through Banyan that the new dataflow model can be efficiently parallelized, showing its scale-up ability on modern many-core architectures and scale-out ability in a cluster. The comparison with the state-of-the-art graph query engines shows Banyan can provide at most 3 orders of magnitude query latency improvement, very stable system throughput and performance isolation. Besides graph queries, the scoped dataflow model can also be applied to general service scenarios that involve complex processing pipelines and require delicate control of the pipeline execution.

A COMPLEX QUERIES

Complex Query 1.

```
g.V(person_id)
 .repeat(__.out('knows')).times(5)
 .dedup().limit(n)
```

CQ_1: Given a start person, find persons that the start person is connected to by exactly 5 steps. Return n distinct person IDs.

Complex Query 2.

```
g.V(person_id)
 .sideEffect(out('workAt')
 .store('companies'))
 .repeat(__.out('knows'))
    .times(5)
    .emit(__.out('workAt')
             .where(within('companies'))
             .count().is(gt(0)))
 .dedup().limit(n)
```

CQ_2: Given a start person, find persons that the start Person is connected to by at most 5 steps. Only persons that *workAt* the same company with the start person are considered. Return n distinct person IDs.

Complex Query 3.

```
g.V(person_id)
 .out('knows').union(identity(), out('knows'))
 .dedup()
 .where(__.in('hasCreator').out('hasTag')
          .out('hasType').values('name')
          .filter{it.get().contains('Country')})
 .order().by().limit(n)
```

CQ_3: Given a start person, find persons that are his/her friends and friends of friends. Only consider persons that have created messages with an attached 'Country' tag. Sort the persons by their IDs and return the top-n person IDs.

Complex Query 4.

```
g.V(person_id)
 .sideEffect(out('workAt')
 .store('companies'))
 .out('knows')
 .where(__.repeat(__.out('knows'))
              .times(4)
              .emit(__.out('workAt')
                       .where(within('companies'))
                       .count().is(gt(0)))
          .dedup().count().is(gt(0)))
 .limit(n)
```

CQ_4: Given a start person, find persons that are his/her friends. Only persons that meet the following constraints are considered: for each S_person in persons, find S_persons that S_person is connected to by at most 4 steps; If any person in S_persons *workAt* the same company as the start person, S_person is selected as a candidate result. Return n distinct person IDs.

Complex Query 5.

```
g.V(person_id)
 .sideEffect(out('workAt')
 .store('companies'))
 .repeat(__.out('knows'))
    .times(5)
    .emit(__.out('workAt')
             .where(within('companies'))
             .count().is(gt(0)))
 .dedup()
 .where(__.in('hasCreator').out('hasTag')
          .out('hasType').values('name')
          .filter{it.get().contains('Country')})
 .limit(n)
```

CQ_5: Given a start person, find persons that the start person is connected to by at most 5 steps and *workAt* the same company with the start person. Only consider persons that have created messages with an attached tag of class 'Country'. Return n distinct person IDs.

Complex Query 6.

```
g.V(person_id)
 .repeat(__.out('knows')
           .where(__.in('hasCreator')
                    .out('hasTag').out('hasType')
                    .values('name').filter{it.get()
                        .contains('Country')}))
 .times(5).dedup().limit(n)
```

CQ_6: Given a start person, find persons that the start person is connected to by exactly 5 steps. Only persons that meet the following constraints are considered: for each S_person in persons, if every I_person in the path from the start person to S_person has created Messages with an attached tag of class 'Country', S_person is selected as a candidate result. Return n distinct person IDs.

REFERENCES

[1] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed evaluation of subgraph queries using worstcase optimal lowmemory dataflows. *arXiv preprint arXiv:1802.03760* (2018).

[2] Renzo Angles, János Benjamin Antal, et al. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020). arXiv:2001.02299 http://arxiv.org/abs/2001.02299

[3] Apache Flink 2021. Apache Flink. https://flink.apache.org/.

[4] Apache Hadoop 2021. Apache Hadoop. https://github.com/apache/hadoop.

[5] Apache Spark 2021. Apache Spark. https://spark.apache.org/.

[6] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices* 30, 8 (1995), 207–216.

[7] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* (1985).

[8] Hongzhi Chen, Changji Li, Juncheng Fang, Chenghuan Huang, James Cheng, Jian Zhang, Yifan Hou, and Xiao Yan. 2019. Grasper: A High Performance Distributed System for OLAP on Property Graphs. In *Proceedings of the ACM Symposium on Cloud Computing*. 87–100.

[9] Cypher 2021. *Cypher Query Language*. Retrieved Dec 20, 2021 from https://neo4j.com/developer/cypher/

[10] DGraph 2021. DGraph: Not everything can fit in rows and columns. https://dgraph.io.

[11] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. 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)*. 17–30.

[12] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 599–613.

[13] GraphDB Annual Growth 2021. *Graph Database Market Size, Share and Global Market Forecast to 2024*.

[14] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 59–72.

[15] Janusgraph 2021. JanusGraph: Distributed, open source, massively scalable graph database. https://janusgraph.org.

[16] Janusgraph 2021. JanusGraph Transactions. https://docs.janusgraph.org/basics/transactions/.

[17] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment* 10, 3 (2016), 217–228.

[18] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2019. Distributed Subgraph Matching on Timely Dataflow. *Proc. VLDB Endow.* (2019).

[19] LDBC Benchmark 2021. LDBC. http://ldbcouncil.org.

[20] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8 (2012), 716–727.

[21] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[22] Ragnar Mellbin and Felix Åkerlund. 2017. Multi-threaded execution of Cypher queries. https://lup.lub.lu.se/student-papers/record/8926892/file/8926896.pdf. (2017). Technical Report.

[23] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[24] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. Ciel: a universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*. 113–126.

[25] Neo4j 2021. Neo4j Execution Plans. https://neo4j.com/docs/developer-manual/3.0/cypher/execution-plans/.

[26] Neo4j 2021. Neo4j: The Fastest Path To Graph Success. https://neo4j.com.

[27] Neptune 2021. AWS Neptune. https://aws.amazon.com/neptune/.

[28] Neptune 2021. Query queuing in Amazon Neptune. https://docs.aws.amazon.com/neptune/latest/userguide/access-graph-queuing.html.

[29] Oracle Berkeley DB 2017. BerkeleyJE 7.5.11. Retrieved Dec 20, 2021 from https://docs.oracle.com/cd/E17277_02/html/index.html

[30] Oracle Berkeley DB 2021. BerkeleyDB 18.1.32. https://www.oracle.com/database/technologies/related/berkeleydb-downloads.html.

[31] OrientDB 2021. OrientDB. https://www.orientdb.org.

[32] Zhengping Qian, Chenqiang Min, et al. 2021. 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)*. USENIX Association.

[33] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. 2019. Fast and robust distributed subgraph enumeration. *arXiv preprint* (2019).

[34] Sherif Sakr, Sameh Elnikety, and Yuxiong He. 2012. G-SPARQL: a hybrid engine for querying large attributed graphs. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 335–344.

[35] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 505–516.

[36] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and concurrent {RDF} queries with RDMA-based distributed graph exploration. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 317–332.

[37] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. RapidMatch: A Holistic Approach to Subgraph Query Processing. *Proc. VLDB Endow.* (2020).

[38] The SNAP datasets 2022. Stanford SNAP. http://snap.stanford.edu/data/index.

[39] TigerGraph 2021. TigerGraph 3.1.0. https://www.tigergraph.com.

[40] Tinkerpop 2021. Apache Tinkerpop. http://tinkerpop.apache.org/.

[41] Tinkerpopo 2021. Tinkerpopo Threaded Transactions. Retrieved Dec 20, 2021 from http://tinkerpop.apache.org/docs/current/reference/#_threaded_transactions

[42] Zhaokang Wang, Rong Gu, Weiwei Hu, Chunfeng Yuan, and Yihua Huang. 2019. BENU: Distributed subgraph enumeration with backtracking-based framework. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 136–147.

[43] Da Yan, James Cheng, M Tamer Özsu, Fan Yang, Yi Lu, John Lui, Qizhen Zhang, and Wilfred Ng. 2016. A general-purpose query-centric framework for querying big graphs. *Proceedings of the VLDB Endowment* 9, 7 (2016), 564–575.

[44] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. Huge: An efficient and scalable subgraph enumeration system. In *Proceedings of the 2021 International Conference on Management of Data*. 2049–2062.

[45] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A distributed graph engine for web scale RDF data. *Proceedings of the VLDB Endowment* 6, 4 (2013), 265–276.

[46] Peixiang Zhao and Jiawei Han. 2010. On graph query optimization in large networks. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 340–351.