GAIA: A System for Interactive Analysis on Distributed Graphs Using a High-Level Language

Zhengping Qian, Chenqiang Min, Longbin Lai, Yong Fang, Gaofeng Li, Youyang Yao, Bingqing Lyu, Xiaoli Zhou, Zhimin Chen, and Jingren Zhou, *Alibaba Group*

https://www.usenix.org/conference/nsdi21/presentation/qian-zhengping

This paper is included in the Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation.

April 12–14, 2021

978-1-939133-21-2

Open access to the Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

GAIA: A System for Interactive Analysis on Distributed Graphs Using a High-Level Language

Zhengping Qian *Alibaba Group*
Chenqiang Min *Alibaba Group*
Longbin Lai *Alibaba Group*
Yong Fang *Alibaba Group*
Gaofeng Li *Alibaba Group*
Youyang Yao *Alibaba Group*
Bingqing Lyu *Alibaba Group*
Xiaoli Zhou *Alibaba Group*
Zhimin Chen *Alibaba Group*
Jingren Zhou *Alibaba Group*

Abstract

GAIA (GrAph Interactive Analysis) is a distributed system designed specifically to make it easy for a variety of users to *interactively* analyze big graph data on large clusters at low latency. It adopts a high-level language called Gremlin for graph traversal, and provides automatic parallel execution. In particular, we advocate a powerful new abstraction called Scope that caters to the specific needs in this new computation model to scale graph queries with complex dependencies and runtime dynamics, while at the same time maintaining the simple and concise programming model. GAIA has been deployed in production clusters at Alibaba to support a variety of business-critical scenarios. Extensive evaluations using both benchmarks and real-world applications have validated the effectiveness of the proposed techniques, which enables GAIA to execute complex Gremlin traversal with orders-of-magnitude better performance than existing high-performance engines, and at much larger scales than recent state-of-the-art Gremlin-enabled systems such as JanusGraph.

1 Introduction

Nowadays an increasing number of Internet applications generate large volume of data that are inherently connected in various forms. Examples include data in social networks, e-commerce transactions, and online payments. Such data are naturally modeled as graphs to encode complex relationships among entities with rich set of attributes. Unlike traditional graph processing that requires programming for each individual task, it is now very common for domain experts, typically non-technical users, to directly explore, examine, and present graph data in an interactive environment in order to locate specific or in-depth information in time.

As an example, consider the graph depicted in Figure 1, which is a simplified version of a real query employed at Alibaba for credit card fraud detection. By using a fake identifier, the "criminal" may obtain a short-term credit from a bank (vertex 1). He/she tries to illegally cash out money by forging a purchase (edge $2 \rightarrow 3$) at time t_1 with the help of a merchant (vertex 3). Once receiving payment (edge $1 \rightarrow 3$) from the bank (vertex 1) at time t_2, the merchant tries to send the money back (edges $3 \rightarrow 4$ and $4 \rightarrow 2$) to the "criminal" via multiple accounts of a middle man (vertex 4) at time t_3 and t_4, respectively. This pattern eventually forms a cycle ($2 \rightarrow 3 \rightarrow 4 \cdots \rightarrow 2$). Such fraudulent activities have become one of the major issues for online payments, where the graph could contain billions of vertices (e.g., users) and hundreds of billions to trillions of edges (e.g., payments). In reality, the entire fraudulent process can involve a complex chain of transactions, through many entities, with various constraints, which thus requires complex interactive analysis to identify.

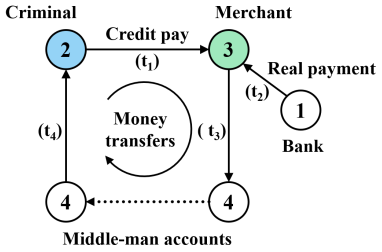


Figure 1: An example graph model for fraud detection.

Our goal is to make it easy for a variety of users to *interactively* analyze big graph data on large clusters at low latency. Achieving this goal requires a different distributed infrastructure than the popular batch-oriented big graph processing systems [4, 15, 16, 26, 39, 49] in two aspects:

Programming Model. Existing systems, including the most recent high-performance data engines such as Naiad [27], demonstrate that it is possible to scale well-known graph algorithms such as PageRank [5] and connected components [23] to large clusters. Even so, their programming interfaces all leave room for improvement for our target users, who typically lack the background on distributed computing or programming in general [13].

Memory Management. Existing systems[1] typically base their execution on the bulk synchronous parallel (BSP) model [44], where the computation proceeds iteratively, and

[1] Here, we focus on the distributed graph analytical systems. Other systems such as Neo4j, ZipG, and JanusGraph, etc. will be surveyed in Section 7.

```
Q1: g.V('account').has('id','2').as('s')
      .repeat(out('transfer').simplePath())
      .times(k-1)
      .where(out('transfer').as('s'))
      .path().limit(1)
```

Figure 2: An example Gremlin query for cycle detection.

in each iteration, all vertices in a graph will conduct the same computation , and send any updates along their edges to drive the computation of the next iteration. The BSP-based engines, however, are not suitable for interactive graph queries because of two reasons. Firstly, the interactive queries typically require maintaining application state along with the traversal paths to enable complex analysis [14, 37], which can grow exponentially with the number of iterations, and cause memory crisis in the underlying execution platforms. Secondly, in interactive environments, there are typically multiple queries sharing the limited amount of memory on the same set of machines, on which (a large part of) the input graph is cached in memory to provide required performance, making the above memory crisis a more critical issue.

In this work, we exploit Gremlin [37] to provide a high-level language for interactive graph queries. Gremlin is widely adopted by leading graph system vendors [1, 6, 21, 29, 30], which offers a flexible and expressive programming model to enable non-technical users to succinctly express complex traversal patterns in real-world applications. For example, one can write the above fraud-detection query in just a couple of lines using Gremlin, as shown in Figure 2 (which we explain in Section 3). In contrast, even common operations like cycle detection, which is a core part of the fraud-detection use case, is tricky to implement in existing graph systems [16, 36].

The flexibility of Gremlin mainly stems from nested traversal with dynamic control flow such as conditionals and loops. While attempting to scale Gremlin queries, we are immediately confronted with the challenges of resolving fine-grained data dependencies [10] with dynamic control flow [45]. Therefore, existing Gremlin-enabled, large-scale systems either adopt a *sequential* implementation in centralized query processing with data being pulled from a remote storage (such as JanusGraph [21] and Neptune [1]), or offer a limited subset of the language constructs (such as the lack of nested loops in [20]). In addition, GAIA must handle dynamics related to variations in memory consumption in an interactive context.

In this paper, we present a system, GAIA, that takes on the challenges of making Gremlin traversal work efficiently at scale with low latency. In particular, GAIA makes the following technical contributions.

- *Scope Abstraction.* We propose the Scope abstraction to allow GAIA to dynamically track fine-grained data dependencies in a Gremlin query. This enables Gremlin traversal to be modeled as a dataflow graph for efficient parallel execution with correctness guarantee.
- *Bounded-Memory Execution.* Leveraging the Scope abstraction, we are able to devise advanced optimizations in parallel graph traversal, such as bounded-memory execution and early-stop optimization, which lead to further runtime improvement and memory saving.
- *GAIA System.* We have developed a full-fledged distributed system, GAIA, and made it available at: https://github.com/alibaba/GraphScope/tree/main/research/gaia. An extended version of GAIA with enterprise features has been deployed in real production clusters at Alibaba to support a variety of business-critical scenarios. Extensive evaluations using both benchmarks and real-world applications have validated the effectiveness of the proposed techniques, which enables GAIA to execute complex Gremlin traversal with orders-of-magnitude better performance than existing engines, and at much larger scales than the state-of-the-art Gremlin-enabled systems such as JanusGraph.

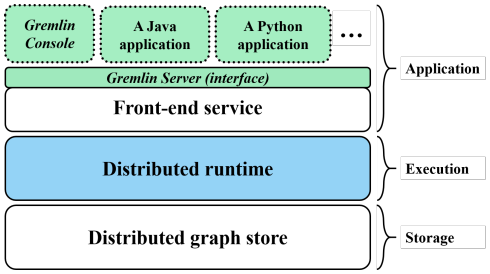


Figure 3: GAIA system architecture.

2 System Architecture

GAIA is a full-fledged, in-production system for interactive analysis on big graph data. Achieving this goal requires a wide variety of components to interact, including software for cluster management and distributed execution, language constructs, and development tools. Due to space limit, we highlight the three major layers that are sufficient to understand this paper, namely application, execution, and storage, in Figure 3, and give an overview to each of them below.

Apache TinkerPop [3] is an open framework for developing interactive graph applications using the Gremlin query language [37]. GAIA leverages the project to supply the application layer. GAIA implements the Gremlin Server [18] interface so that the system can seamlessly interact with the TinkerPop ecosystem, including development tools such as Gremlin Console [17] and language wrappers such as Java and Python.

The GAIA execution runtime provides automatic support for efficient execution of Gremlin queries at scale, which constitutes the main contribution of this paper. Each query is compiled by the front-end service into a distributed execution plan that is partitioned across multiple compute nodes for parallel execution. Each partition runs on a separate compute node, managed by a local executor, that schedules and executes computation on a multi-core server.

The storage layer maintains an input graph that is hash-partitioned across a cluster, with each vertex being placed

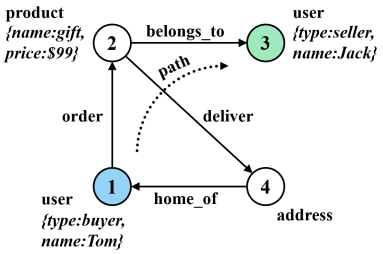


Figure 4: An example "e-commerce" property graph.

together with its adjacent (both incoming and outgoing) edges and their attributes. In this paper, we assume that the storage is coupled with the execution runtime for simplicity, that is each local executor holds a separate graph partition. In production, we implement a distributed graph storage with index and cache features, decoupled from the execution, that supports real-time updates with snapshot isolation (similar to Kineograph [11]), which allows users to query fast-changing graphs with consistency guarantee. Furthermore, GAIA provides multiple options for fault tolerance using checkpoints, replication, and/or relying on a Cloud storage. Production details are outside the scope of this paper.

3 Programming with GAIA

GAIA is designed to faithfully preserve the programming model of TinkerPop [3], and as a result it can be used to scale any existing TinkerPop applications to large compute clusters without any modification. In this section, we provide a high-level view of the programming model, highlighting the key concepts including the data model and query language.

Gremlin [37] enables users to define ad-hoc traversals on property graphs [2]. A property graph is a directed graph in which vertices and edges can have a set of properties. Every entity (vertex or edge) is identified by a unique identifier (`ID`), and has a (`label`) indicating its type or role. Each property is a key-value pair with combination of entity `ID` and property name as the key. Figure 4 shows an example property graph. It contains `user`, `product`, and `address` vertices connected by `order`, `deliver`, `belongs_to`, and `home_of` edges. A path following vertices $1 \rightarrow 2 \rightarrow 3$, shown as the dotted line, indicates that a buyer "Tom" ordered a product "gift" offered by a seller "Jack", with a price of "$99".

In a Gremlin traversal, a set of *traversers* walk a graph according to particular user-provided instructions, and the result of the traversal is the collection of all halted traversers. A traverser $T = (l, P)$[2] is the basic unit of data processed by a Gremlin engine. Each traverser maintains a location l that is a reference to the current vertex, edge or property being visited, and (optionally) the path history P. For example, consider a traversal which starts from vertex 1 (with only one traverser at the location of vertex 1), follows outgoing edges, and reaches its 2-hop neighbors in Figure 4. A possible intermediate result can be a collection of a single traverser located at vertex 2 with the corresponding path history. The final result is a collection of two traversers, located at vertex 3 and 4, respectively, with different paths, $1 \rightarrow 2 \rightarrow 3$ and $1 \rightarrow 2 \rightarrow 4$.

Nested traversal is another key concept in Gremlin. It allows a traversal to be embedded within another operator, and used as a function to be invoked by the enclosing operator for processing input. The role and signature of the function are determined by the type of the enclosing operator. For example, a nested traversal within the `where` operator acts as a predicate function for conditional filters, while that within the `select` or `order` operator maps each traverser to the output or ordering key for sorting the output, respectively.

Nested traversal is also critical to the support for loops, which are expressed using a pair of the `repeat` and `until`/`times` operators. A nested traversal within the `repeat` operator will be looped over until the given break predicate is satisfied. The predicate (or termination condition) is defined within the `until` operator, applied to each output traverser separately from each iteration. The `times` operator can also terminate a loop after a fixed number of k iterations.

Example 3.1. *Figure 2 shows a Gremlin query Q1 for the motivating example in Section 1 that tries to find cyclic paths of length k, starting from a given account. First, the source operator* `V` *(with the* `has` *filter) returns all the* `account` *vertices with an identifier of "2". The* `as` *operator is a* modulator *that does not change the input collection of traversers but introduces a name (s in this case) for later references. Second, it traverses the outgoing* `transfer` *edges for exact $k-1$ times, skipping any repeated vertices (by the* `simplePath` *operator). Third, the* `where` *operator checks if the starting vertex s can be reached by one more step, that is, whether a cycle of length k is formed. Finally, for qualifying traversers, the* `path` *operator returns the full path information. The* `limit` *operator at the end indicates only one such result is needed.*

4 Compilation of Gremlin

GAIA compiles a Gremlin query into a dataflow graph, where each vertex (operator) performs a local computation on input streams from its incoming edges and produces output streams to its outgoing edges, and can optionally maintain a state. The input graph is modeled as a read-only state shared by all the dataflow operators. We map each Gremlin operator onto a dataflow operator, and the collections of traversers as data streams. In the following, we will use the term traverser interchangeably with data. Figure 5(b) shows an example dataflow graph corresponding to the following Gremlin query (Q2) that conducts a 2-hop traversal followed by an aggregation that counts the total number of traversed paths.

```
Q2: g.V(2).out().out().count()
```

We introduce source operators as special drivers that generate output only from the input graph to drive the rest of the

[2] In [37], a traverser is modelled as a 6-tuple set, while we include necessary elements to understand this paper.

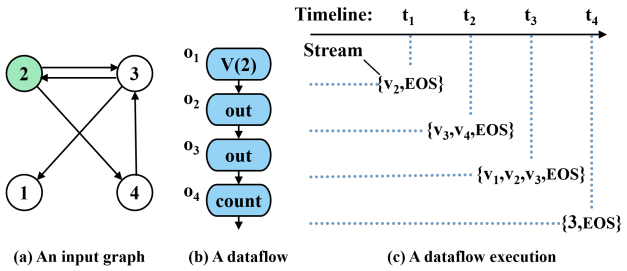


Figure 5: Dataflow graph and execution for query Q2.

dataflow computation (e.g., `V(2)`). We use sink operators to denote those that generate output streams for the computation to be consumed elsewhere (e.g., `count`). Since Gremlin imposes no restrictions on the execution order of traversers, we can pack a segment of traversers to a same operator into a batched input and schedule the computation at a coarse granularity for efficient execution.

To preserve the operator semantics for barriers, we insert an End-of-Stream (or EOS) marker at the end of the output streams of each source operators, as a special punctuation event that asserts the completeness of output. The EOS markers will be propagated through the dataflow, layer by layer, so that any downstream operators can be notified on the completeness of their inputs by waiting to collect those markers.

Example 4.1. *Figure 5(c) illustrates the progression of the dataflow execution of Q2 against the input graph in Figure 5(a). o_1 generates a data stream of $\{(v_2,\emptyset), EOS\}$ as output, where v_2 denotes the vertex with* `ID` *2. Note that the path history has been pruned (and omitted later) as the downstream operators do not need it. o_2 consumes v_2, generates output $\{v_3, v_4\}$, and finally propagates EOS to its output. Subsequently, o_3 outputs $\{v_1, v_2\}$ after consuming v_3, and $\{v_3, EOS\}$ for the rest of its input. Finally, o_4 outputs the counting of $\{3\}$ - it can do so as the EOS marker has been received. The dataflow thus terminates.*

4.1 Challenges in Compiling Nested Traversal

Many of the salient features of Gremlin such as dynamic control flow rely on nested traversal, which introduces additional complexity to the above design. Let's look into another query Q3 slightly amended from Q2, in which a segment of operators (`out().count()`) is nested within a `select`-projection.

```
Q3: g.V(2).out()
      .select('neighbor_count')
      .by(out().count())
```

Given a set of vertices $N(v_2)$ as the outgoing neighbors of a vertex v_2, the query asks to count the number of k-hop paths starting from each vertex $u \in N(v_2)$ (let $k = 1$ for simplicity), and output pairs of $(u, \#$ paths starting from $u)$. In this example, each input traverser that represents a vertex of $N(v_2)$ does its own computation (of the counting of paths), namely

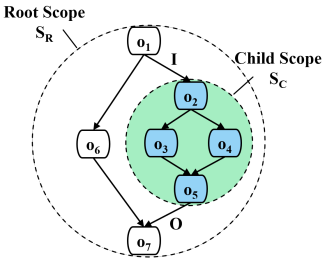


Figure 6: Dataflow and scope example: the filled circle highlights a scope with input stream I and output stream O.

at a fine granularity. In other words, the `count` operation has to be executed separately for each vertex $u \in N(v_2)$.

We define a *context* as an execution environment for a dataflow that includes a unique (possibly empty) state for its computation. Without nested traversal (and/or dynamic control flow), all computation of each Gremlin operator, and the whole dataflow, can run in a single context. For example, in query Q2, only `count` maintains a state (for partial counting) - there is only one such state needed to count all traversed paths. With nested traversal, this property no longer holds as a stateful operator in a nested traversal can dynamically demand the separation of contexts. For example, in query Q3, due to the semantics of `select`, there must be an individual state (context) maintained for each vertex $u \in N(v_2)$ in order to produce correct results.

One may argue that the above example is not so hard to tackle. However, this is just a simplest example involving sub-traversals in Gremlin. Such context separation is also important in dynamic control flow such as loop, in which each iteration must run separately from another. One can even encounter sub-traversals involved with arbitrary combination of complex structuring constructs, making the system design uncontrollably complex. In addition, the number of separate contexts required for the correct execution of a *single* Gremlin traversal can be proportional to that of the intermediate traversers (e.g., `select` in query Q3), which can be of millions to billions in our case. While it is possible to dynamically create physical contexts as in [45], doing so at such a fine granularity for Gremlin is clearly infeasible in practice.

4.2 The Scope Abstraction

To address the issues posted by Gremlin traversal, we propose the Scope abstraction to help emancipate the system from the need of maintaining context information.We first define the concept of a Scope.

Definition 4.1. A Scope is a subgraph in a dataflow (sub-dataflow) that satisfies the following condition: for any operators o_1 and o_2 in the sub-dataflow and any operator o in the dataflow, o must also be in the sub-dataflow if o is on a directed path from o_1 to o_2.

A Scope has the same logical structure (and function) as a dataflow operator, which can thus be reduced to one vir-

tual "operator" in the dataflow graph. Naturally, we refer to a Scope context, as the context of its enclosed operator. It is allowed that a Scope S_p contains another sub-dataflow as a nested Scope S_c as long as it satisfies the definition of Definition 4.1. S_p is called the parent Scope of S_c, and S_c is accordingly the child Scope of S_p. The whole dataflow is a special Scope that we call as *root* Scope. The dataflow regarding the nested relationships of Scopes naturally form a hierarchical structure.

Example 4.2. *In the dataflow graph as shown in Figure 6, the sub-dataflow comprised of o_2, o_3, o_4 and o_5 (as well as all their edges) is a* Scope *S_c (as highlighted) and can be reduced to one operator with I as its input stream, and O as its output stream. The whole dataflow is the root* Scope*, which is the parent* Scope *of S_c.*

As we mentioned earlier, it is costly to create physical dataflow operators as in [45] for a Gremlin query that potentially requires a separate context for each data item. We therefore propose the Scope abstraction to handle the separation of execution contexts in a Scope in a more light-weighted manner. A Scope abstraction consists of three primitives, namely `Enter`, `Exit`, and `GoTo`, and the interface of Scope policy. Specifically, `Enter` forwards a data item from a parent Scope[3] to a child Scope, while `Exit` sends data item back to a parent Scope. As `GoTo` is primarily used for loop control flow, we will introduce it in Section 4.3.

The Scope policy is installed by the compiler on each `Enter` and `GoTo` primitives to fulfil different context-switch mechanisms. Logically, we use a sequence number as context identifier to identify an execution context in a Scope, the Scope policy contains the following interfaces (their implementations are in Section 4.4):

- `CreateOrOpen(Data:`e`,CtxID:`s`)`: To create a new isolated context for the input data e, or open an existing context uniquely identified by s.
- `GetContext(Data:`e`)`: To obtain the context identifier of the data e.
- `Complete(Data:`e`,CtxID:`s`)`: To mark that there will be no more data for the context of s, after receiving e.

As an example, we present a built-in scope policy called `CONTEXT_PER_ENTRY` (more policies will be introduced as follows). `CONTEXT_PER_ENTRY` creates a new context for each input data. Let *seq* be a sequence number, initialized to 0. For each input e, the `CONTEXT_PER_ENTRY` policy first applies `CreateOrOpen`(e, seq) to create a new context for e. It then immediately calls `Complete`(e, seq) to indicate that there will be no more data for the context of *seq*. Finally, the policy increments *seq* by 1 such that any future data will enter a different context. In the following, we will detail how the Scope abstraction facilitates the compilation of a Gremlin query with nested traversals.

[3] It is more precisely a context of the Scope, while we refer to it as Scope for short.

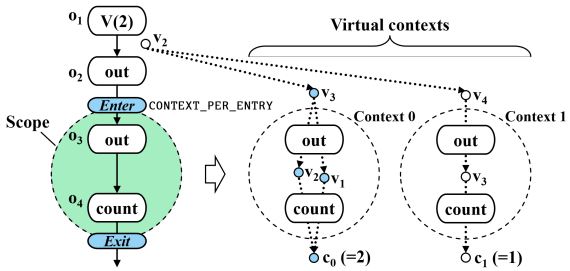


Figure 7: An example Scope execution with separate contexts.

4.3 Compilation of Gremlin using Scope

Compilation of a Gremlin query without dynamic control flow or nested traversal is as similar to that in existing systems [41, 46, 47], we do not elaborate on it further. Both dynamic control flow and nested traversal introduce sub-traversals in a Gremlin query. GAIA compiles each such sub-traversal into a Scope enclosed by a pair of `Enter` and `Exit` primitives (can be multiple of them nested within each other). The Scope abstraction handles the context separation in a unified way. Due to space limit, this section presents the compilation process of three representative Gremlin operators (`select`, `where`, and `repeat`) to highlight the common pattern of using the Scope abstraction.

Example 4.3. *Figure 7 illustrates an example that* GAIA *compiles the query Q3 (Section 4.1) into a dataflow using* Scope*, in which the `select`-projection introduces a* Scope *that encloses the sub-traversal of `out().count()`. As there requires a separate execution context for each data entering the* Scope*,* GAIA *installs a `CONTEXT_PER_ENTRY` policy on the `Enter`. This way, each data can drive their own computation of `out().count()` in isolation, without concerning about the context separation as posted in Section 4.1.*

Dynamic control flow such as `where`-conditionals and `repeat`-loops introduce addition complexity, as presented in the following query:

```
Q4: g.V(2).as('s')
      .repeat(out().simplePath())
      .times(k-1)
      .where(out().eq('s'))
      .path().limit(1)
```

We next focus on the compilation of these constructs, inspired by TensorFlow [45]. However, unlike [45], they can be applied to a much finer granularity of each individual traversal path in Gremlin. This is enabled by the Scope abstraction. We further introduce the following primitive operators:

- `Copy` takes in a data e and outputs two identical data.
- `Switch` takes a data from its input and a boolean value p, and forwards the data to either the `True` branch of d_t or `False` branch of d_f, based on the predicate p.
- `Merge` accepts two input streams and merges them into one single output stream.

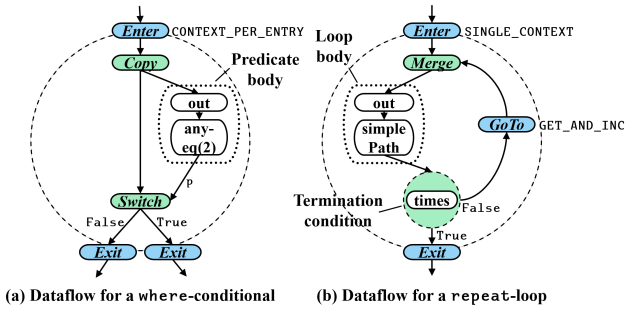


Figure 8: Compilation of control-flow constructs.

Conditional. Figure 8(a) shows an example of compiling a `where`-conditional. Conceptually, the `where` statement determines whether a data, while arriving at `where`, will continue to traverse, if the sub-traversal is evaluated to be true, or be abandoned otherwise. As the conditional check happens for each individual data, a `CONTEXT_PER_ENTRY` policy will be installed by the compiler in the `Enter` while entering the `where` Scope. Each data enters the `Copy`, where one data goes into the predicate body to drive the sub-traversal, and the other data goes to the `Switch`. Based on returned boolean value of the predicate body, the data with a `True` predicate will get out of the Scope via the `True` branch, and the data with a `False` predicate will go via the `False` branch (and get discarded if not further used).

Loop. We first introduce other two built-in Scope policies.

- `SINGLE_CONTEXT` policy calls `CreateOrOpen`$(e,0)$ for each data e indicating that they all enter one context of 0. It calls `Complete`$(e,0)$, if and only if $e = EOS$.
- `GET_AND_INC` policy first calls `GetContext`(e) to obtain the context of e as seq. Then it increases seq by 1 as seq', and calls `CreateOrOpen`(e,seq') to enter the new context. It finally calls `Complete`(e,seq'), if and only if $e = EOS$.

Figure 8(b) illustrates the compilation of `repeat`-loop. The compiler installs the `SINGLE_CONTEXT` policy on the `Enter` that forwards a data into the loop Scope, with a new context of 0, or, in the 0-th iteration. Additionally, it installs the `GET_AND_INC` policy on the `GoTo`. The `GoTo` primitive, as mentioned earlier in Section 4.2, is used to explicitly switch the context of data. Specifically in a loop, it leverages the `GET_AND_INC` policy to allow any data produced from current loop context to get switched to the next iteration. Naturally, the context identifier can now serve as the loop count. The loop body compiles any sub-traversal that will be run iteratively. Eventually, the data in the loop context will go though a conditional Scope as we have discussed above. This conditional Scope checks whether a termination condition is satisfied (such as arriving at the maximum iteration by `times`, or traversing to a certain vertex by `until`). The data with a `False` predicate is able to exit the loop, while the data with a `True` predicate will proceed to the next iteration as a feedback data stream via the `GoTo`, updating its context via the `GET_AND_INC` policy to indicate entering the next iteration. Note that a context must have been created or opened for each data e in a Scope, and thus `GetContext`(e) can be safely called. The feedback data will be eventually merged back to the input (of the sub-traversal) to drive the computation of next iteration.

4.4 Implementing Scope

It is challenging to implement Scope both correctly and efficiently. While it is always possible to create physical dataflow operators for each separate context, due to potentially unbounded number of such contexts in graph traversals (as described in Section 4.1), this is clearly infeasible in practice. GAIA instead dynamically tracks dependencies among input, output, and internal states for each operator in a dataflow.

GAIA labels each traverser with a tag, which is a k-ary vector of context identifiers, denoted as $T = [s_1, s_2, \ldots, s_k]$[4], where the dimension indicates the level of potentially nested Scope. The root Scope is by default identified by a tag of $[\,]$. We define the following operations on a tag T:

$T[\wedge]$	To get the last context identifier of T.
$T[\wedge \rightarrow s]$	To replace the last context identifier of T with s.
$T[+1]$	To increase the dimension of T by 1, with the new slot filled with a $\emptyset$.
$T[-1]$	To reduce the dimension of T by 1.

From now, each data e will be tagged as $(T;e)$, which allows the system to be aware of the Scope and its different contexts. The primitives of `Enter` and `Exit`, and the interface functions in the Scope abstraction will explicitly modify the tag, as follows.

- `Enter` increases the dimension of the tag by 1 to indicate entering a Scope, as $(T[+1];e)$.
- `Exit` reduces the dimension of the tag by 1 to indicate leaving a Scope, as $(T[-1];e)$.
- `CreateOrOpen`$((T;e),s)$ return a newly tagged data with the last context identifier of T replaced as s, as $(T[\wedge \rightarrow s];e)$.
- `GetContext`$((T;e))$ returns the last context identifier of T, as $T[\wedge]$.
- `Complete`$((T;e),s)$ produces a tagged EOS marker to indicate the end of current context s, as $(T[\wedge \rightarrow s];EOS)$.

Such data tagging is automatically handled by GAIA system, and is transparent to any user interface. For the primitive operators introduced in Section 4.3, they do not need to worry about tags, and hence can still treat the tagged data as a "normal" data. For a computing operator o (with the logic f_o) in Gremlin, such as `out` and `count`, GAIA handles the computation as follows. It first extracts the actual data e, and apply the computation logic $f_o(e)$. The computation will generate a set

[4]Such tagging appears to be similar to the timestamps in Naiad [27], but it is used for dependency tracking in GAIA, without any physical meaning of event time as in Naiad [27].

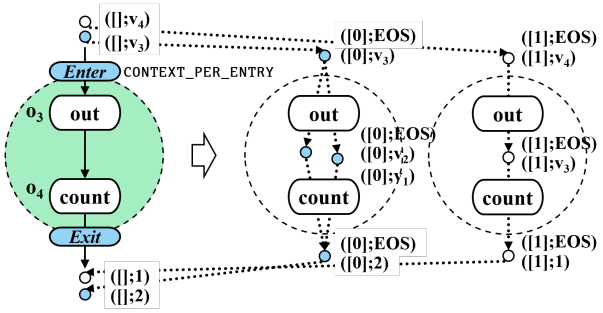


Figure 9: An execution with dynamic dependency tracking.

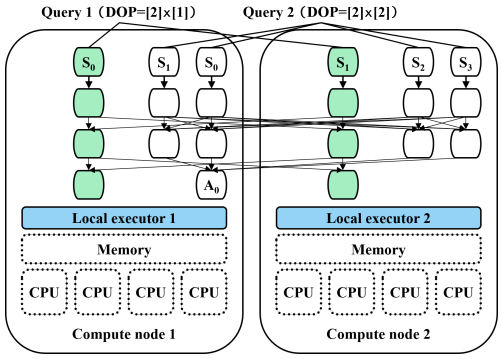


Figure 10: Distributed execution on two compute nodes.

of traversers Ω, and potentially modify a state τ of the operator. Then for all $e' \in \Omega$, GAIA re-tags e' with T and sends it to the output stream. To handle any stateful computation, GAIA maintains an associated map with tag T as the key and state τ as the value, so that it can operate on the right state from different execution contexts transparently, as if the operator runs in isolation.

Example 4.4. *Figure 9 shows the above process for the dataflow in Figure 7. Initially, it accepts and computes inputs* $\{([\,];v_3), ([\,];v_4)\}$ *(path history is omitted) from the parent* Scope *context. The* `Enter` *of the* `select` Scope *turns the inputs to* o_3 *as* $\{([0];v_3), ([0];EOS), ([1];v_4), ([1];EOS)\}$ *according to the* `CONTEXT_PER_ENTRY` *policy. Next,* o_3 *outputs* $\{([0];v_1), ([0];v_2), ([1];v_3)\}$*. Note that EOS is omitted for now.* o_4 *can then maintain a hash table with the tag as key and the partial count as value. Finally, while* o_4 *receives the EOS for the corresponding context, it can output the results as* $\{([0];2), [[1];1]\}$*. The* `Exit` *restores the tags from* o_4*'s output and generates* $\{([\,];2), ([\,];1)\}$.

Handling EOS Markers. An EOS marker can be introduced by both the source operator and the `Complete` function inside a Scope (`Enter`). An EOS marker can go through any computing operator without doing any de-facto computation, while it must be carefully handled in the primitive operators, especially `Enter` and `Exit` with the presence of Scopes.

Given a Scope, we call an EOS marker produced from outside the Scope as external EOS, and an EOS produced inside the Scope as internal EOS. An external EOS marks the termination of a context in the parent Scope, and must exit back to the parent Scope. Conversely, an internal EOS fulfills the same purpose only in the current Scope, and should only be propagated within. It is thus critical to differentiate the semantics of the EOS markers in a Scope. To do so, we implement the policy installed on the `Enter` to not call `CreateOrOpen` on the external EOS marker, which can then be recognized as a $\emptyset$ context. In the `Exit`, GAIA only allows the external EOS to leave the Scope.

Recall that `Switch` is another primitive operator used in conditional and loop Scope that delivers a data to either branch based on the predicate. The EOS marker, however, will always be propagated to both branches. In the loop Scope, the external EOS, once propagating through the nested conditional, will be held in the `Exit` of loop, and only released after the system verifies that all loop contexts terminate (using known techniques [45]). For the internal EOS, it will be tagged as the other data in the `GoTo`. As long as any data with a tag T is propagated to the next iteration, the EOS with T must also be propagated over to `GoTo` (meaning that the associated loop context has not terminated); otherwise, it will leave the loop Scope and get discarded.

5 Distributed Execution

GAIA runs queries via a set of worker processors in a shared-nothing cluster, where each worker executes a fragment of the computation. For each query, GAIA first compiles it into a dataflow graph using the techniques in Section 4, then it partitions the source operator in the dataflow according to the input graph partition, with the segment of operators that follow the source replicated across the set of workers. A local executor manages the computation on each worker by scheduling the operators to run. It starts from the source operator and repeatedly executes the following *ready* operators. Here, an operator is ready if all its inputs are available to consume. For now, GAIA requires the users to manually specify a degree of parallelism (DOP) for a query upon submission. We leave it as an interesting future work to automatically derive the DOP. According to the DOP, the local executor parallelizes the operators to execute on the multiple CPU cores, as illustrated in Figure 10. While GAIA can support multiple concurrent queries, we focus on single query processing in this paper.

5.1 Bounded-Memory Execution

Graph traversal can produce paths of arbitrary length, leading to memory usage growing exponentially with the number of hops. Although it is very common for Gremlin queries to terminate with a top-k constraint and/or aggregate operation, such an explosion of *intermediate* results can often lead to memory crisis, especially in an interactive environment with limited memory configuration. While several techniques exist for alleviating memory scarcity in dataflow execution, such as backpressure and memory swapping, they cannot be directly

applied in GAIA due to potential deadlocks [25, 31] and/or high (disk I/O) latency. To ensure *bounded-memory execution* without sacrificing performance (parallelism), the local executor in GAIA employs a new mechanism for dataflow execution, called *dynamic scheduling*.

Dynamic Scheduling. For each operator, GAIA packs a segment of consecutive traversers in a stream into a single batch, and such a batch constitutes the finest data granularity for communication and computation. A *task* can be logically viewed as the combination of an operator and a batch of data to be computed. GAIA dynamically creates tasks corresponding to each operator when there is one or more batches available from all its inputs[5]. The local executor maintains all the tasks in a same scheduling queue to share resources.

We implement our own memory allocator that will report the total amount of memory used (for each query) so that the executor can watch the memory consumption. When it reaches a predefined threshold (high-watermark), the executor stops scheduling more tasks from the queue, except for those corresponding to the sink operators that will be sent to the clients. The executor resumes scheduling tasks when the memory consumption drops below another predefined threshold (low-watermark). It is possible that a single task (with a high-degree vertex) execution may produce too much output to exhaust the memory. To avoid this issue, we suspend a task when its output data exceeds a *capacity* bound, and resume it after the data has been consumed.

Data shuffling between two machines may introduce dependencies between their task scheduling. For example, a task can cause another executor to run into low memory, if it sends too much data to that executor. In this case, the sender task will be suspended until the receiver executor recovers from low memory. We implement a mechanism to send backpressure signals across network to allow cooperation of schedulers.

An execution of a dataflow graph with cyclic edges can potentially deadlock using bounded memory. In the specific context of graph traversal, this can be caused either by *infinite loops* such as traversing along a cyclic path without termination, or *inappropriate scheduling* such as buffer exhausted by a BFS-prioritized traversing (will be discussed later) that prevents downstream or sink operators from being scheduled to drain the buffered intermediate data. To address infinite loops, we apply a configurable limit N of the maximum number of iterations allowed in a loop (with a small buffer reserved for each iteration), and let the `GoTo` declare a deadlock when the limit N is reached. Once a deadlock is detected, the corresponding query is terminated with a clear error message. To handle inappropriate scheduling, we adopt a hybrid traversal strategy as described below.

Hybrid Traversal Strategy. As mentioned above, the memory crisis mainly stems from the intermediate paths, and therefore the traversal strategies can greatly impact the memory usage. There are two typical traversal strategies, namely (breadth-first-search) BFS-like traversal and (depth-like-search) DFS-like traversal. BFS-like traversal can better utilize parallelism, while it may produce data all at once that drives high the memory usage. On the contrary, DFS-like traversal tends to consume much less memory, while it may suffer from low parallelism. With this observation, we propose to allow the local executor to schedule tasks with priorities according to its topological order (i.e. the traversal depth) in the dataflow. Specifically, the executor can schedule the tasks located at the same order with higher priority for a BFS-like traversal, and prioritize those at downstream to follow a DFS-like traversal. Note that such strategy works naively for all the tasks but those in a loop context, where the traversers from different iterations may be executed in the same task. To resolve this, we let the operator's buffer reorder (and group) traversers by their iteration markers (obtained from the context identifier) before packing them into batches. This makes sure that we can prioritize tasks unambiguously even within loops. To balance the memory usage with the performance (parallelism), GAIA by default adopts a hybrid traversal strategy, that is, it uses BFS-prioritized scheduling as it has better opportunities for parallelization, and automatically switches to DFS-prioritized in case that the current operator arrives at the memory bound.

[5]The only exception is `Merge`, which is ready to run when data become available at any of its inputs.

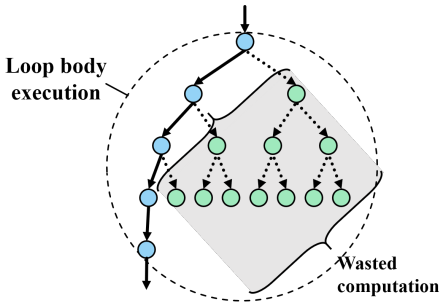


Figure 11: A loop execution with wasted computation.

5.2 Early-Stop Optimization

Traversing all candidate paths fully is often unnecessary, especially for interactive queries with dynamic conditions running on diverse input graphs. For example, in the following query Q5, only the first k results are needed.

```
Q5: g.V(2).repeat(out().simplePath())
      .times(4).path()
      .limit(k)
```

This leads to an interesting tradeoff between parallel traversal and wasted computation, as further illustrated in Figure 11. It shows an example run of query Q5 with $k = 1$. The circle denotes the traversal specified by the `repeat`-loop. Assume we have enough computation resource (CPU cores), the paths can be explored in a fully parallel fashion. However, once a 4-hop path is found, all the remaining parallel traversal will be no longer required.

For real-world queries on large graph data, such wasted computation can be hidden deeply in nested traversals (e.g., a predicate that can be evaluated early from partial inputs) and significantly impact query performance. While avoiding such wastage is straightforward in a sequential implementation, it is challenging to do so for a fully-parallel execution.

Normally, the execution of a particular context terminates when the EOS markers arrive at all the exits (from this context), including any `Exit` or `GoTo`. In the above example, an operator (e.g., `limit`) can actually terminates early after producing k outputs, before receiving any input EOS markers. GAIA further allows `Complete`$((T;e),s)$ to be called by any operators in a Scope to explicitly produce a tagged EOS marker (for current context s) to indicate the completeness of its output (after sending e downstream). However, this alone does not prevent upstream computation from continuing producing output that is no longer required and thus the corresponding computation is wasted.

To minimize such wastage, when a `Complete` is issued by an operator, it creates a *cancellation token* associated with the same context tag that is sent backward along input edges to its upstream operators within the Scope. The token serves as a signal for receiving operators to clear any unsent output data and immediately insert an EOS marker for the particular output stream. If such a token has been received from all output streams, the operator further propagates it to its own upstream operators, recursively, until it encounters the `Enter` for the same Scope. Such cancellation notification is implemented at a system level by GAIA. Due to space limit, We omit further details on propagation of cancellation tokens in any child Scope and/or through the `GoTo` to its dependent, previous contexts. We validate that such early-stop optimization can significantly improve query performance in Section 6.

6 Evaluation

6.1 Experimental Setup

Datasets. We generate 5 graph datasets as shown in Table 1 for experiments using Linked Data Benchmark Council (or LDBC) data generator [12], where G_x denotes that the graph is generated with $scale = x$. We use G_{300} as the default dataset if not otherwise specified. Note that G_{1000} is the largest data graph that LDBC can generate.

Table 1: The LDBC datasets.

Name	# vertices	# edges	Agg. Mem.
G_1	3M	17M	4GB
G_{30}	89M	541M	40GB
G_{100}	283M	1,754M	156GB
G_{300}	817M	5,269M	597GB
G_{1000}	2,687M	17,789M	1,960GB

Queries. For comparison, we consider graph queries from the Social Network Benchmark defined by LDBC [12] to model industrial use cases on a social network akin to Facebook. We choose 10 out of 14 *complex read* queries (denoted as CR-1...14) from LDBC's Interactive Workload[6].

In addition, the cycle-detection query Q6 is considered: given m (by default 10) starting nodes in V, it traverses from V via at most k (by default 4) hops, and returns those vertices among V that can form at least n (by default 10) cycles along the traversal. We modify the query based on the production query as shown in Figure 1 to align with the LDBC data. This query also shows the functionality of *prepared statement* ("Discussion", Section 4.3) enabled by the Scope abstraction, which wraps multiple starting vertices into one query.

The driver client provided by LDBC is modified to run each of the queries 20 times from a set of randomly selected parameters. Average query latency is reported.

Configurations. In the following experiments, we by default warm up all the systems to keep the computation-relevant data in memory. We do this to focus on benchmarking the computing engine instead of storage access.

All the queries have been implemented using Gremlin for all systems except Neo4j (using Cypher officially), with correctness cross-verified. The compiling time of these queries in our system is typically within 1ms, which is negligibly small compared to the query runtime, and will be ignored thereafter. We allow each query to run for at most 1 hour, and mark an OT if a query can not terminate in time. We manually configure the degree of parallelism (DOP) while running each query in GAIA. In the following, we denote DOP = $[x] \times [y]$ for running y threads in x machines.

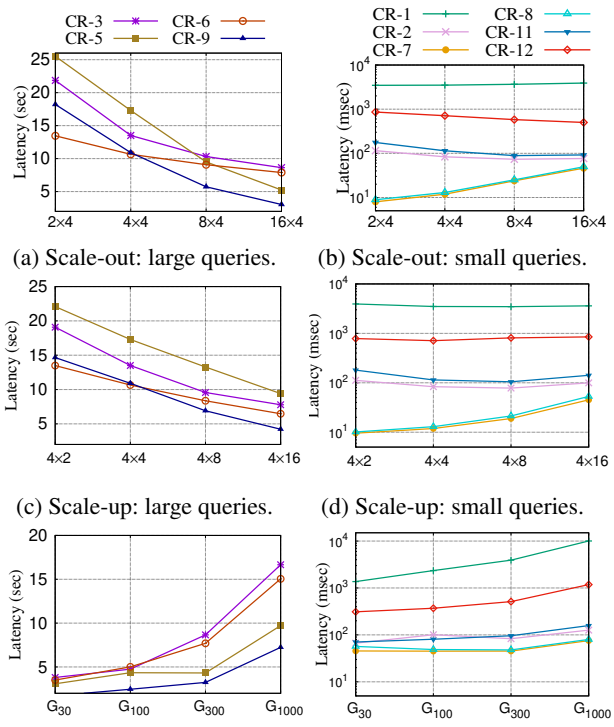
We compare GAIA with the systems in Table 2. While Neptune [1] is another popular Gremlin-enabled graph database, we do not benchmark it as it is only available in AWS, and its performance is similar to JanusGraph as shown in [42]. Timely [43] is the publicly available implementation of Naiad [27]. Plato [32] is an open-sourced implementation of Gemini [49] (Gemini does not support (de)serializing vector-like data for sending paths across network). We implement GAIA using Rust [38], and are working on open-sourcing the engine and storage.

Table 2: The evaluated systems.

System	Version
TinkerGraph [3]	3.4.1
Neo4j-Community [29]	3.5.8
OrientDB [30]	tp3-3.0.15
JanusGraph [21]	0.4.0-hadoop2
Timely [43]	latest release in Github
Plato [32]	latest release in Github

We deploy a cluster of up to 16 machines, and each machine configures one 24-core Itel(R) Xeon(R) Platinum 8163 CPUs

[6] The remaining queries are either too simple (such as simple point-lookup queries) or rely on user-defined logic (such as CR-4, 10, 13, 14), which is not supported by other popular TinkerPop-based systems.

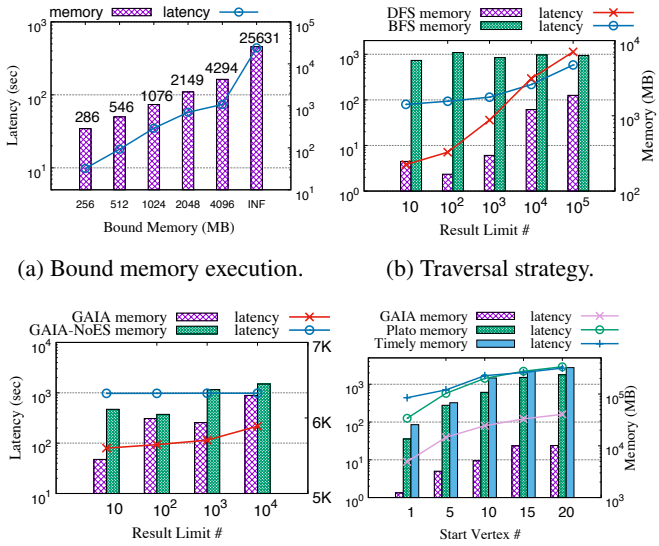


(a) Scale-out: large queries. (b) Scale-out: small queries.

(c) Scale-up: large queries. (d) Scale-up: small queries.

(e) Data size: large queries. (f) Data size: small queries.

Figure 12: The scalability experiment.



(a) Bound memory execution. (b) Traversal strategy.

(c) Early stop. (d) Compare with big engines.

Figure 13: The experiment of our design choices.

at 2.5GHz and 512GB memory. The servers are connected through 25Gbps network.

6.2 Scalability

To the best of our knowledge, GAIA is the only system that can handle Gremlin queries at scale. In this experiment, we study the scalability of GAIA while running all LDBC queries. We divide these queries into two groups based on their runtime to better present the result: (1) large queries CR-3, 5, 6 and 9; (2) small queries CR-1, 2, 7, 8, 11, 12.

Scale-out. To study the scale-out performance, we fix y to 4 while varying x as 2, 4, 8, 16[7], and report the latency of each case in Figure 12a and Figure 12b. We analyze the result regarding the two query groups:

Large queries. These queries traverse large amount of data and run relatively longer, while they scale well with up to 6× performance gain from 2 machine to 16 machines. While CR-3 performs the worst to obtain only 3× performance gain, we recognize that it contains very complex nested sub-traversals that may introduce extra cost in synchronization (e.g. waiting for the EOS marker).

Small queries. Due to either effective filtering or small range of traversal, the small queries only touch a small amount of data and thus are not computation-intensive. We expect that their performance may not be improved with more parallelism, while CR-2 and CR-12 still run consistently faster as shown in Figure 12b. CR-1, as a relatively slow query in this group, demonstrates seemingly counter-intuitive result. The query actually asks to print out a lot of information after locating the target vertices, which constitutes a majority of the computation that cannot benefit from more parallelism.

Scale-up. We then fix x to 4, and vary y as 2, 4, 8, 16, and report the result in Figure 12c and Figure 12d. Similar to the scale-out cases, the large queries scale consistently, while small queries do not gain speedup, as more parallelism is used. It is interesting to compare the scale-out and scale-up cases with the same DOP, $[4] \times [16]$ vs. $[16] \times [4]$ as an example, we can observe $[16] \times [4]$ cases in-general perform better, even it requires more communication. The result shows that (1) communication cost is not a critical impact factor for GAIA, for which the dynamic scheduling techniques can seamlessly hide the communication cost by allowing ready tasks to get scheduled; (2) data contentions may be a more serious issue for interactive graph queries, as they are more often confronted in fewer machines.

Data Size. Finally, we fix the DOP as $[16] \times [4]$, and run the queries over the datasets of G_{30}, G_{100}, G_{300} and G_{1000}. Note that the sizes of these graphs are roughly linear to their *scale* factors. The result is in Figure 12e and Figure 12f. For the large queries, GAIA scales quite well with the growing of the data. For the small queries (except CR-1, as explained earlier), the performance stays roughly stable, as these queries only touch a small amount of data.

Discussions. The experiment demonstrates reasonable trends of scalability of GAIA: in general, the larger the query, the better the scalability. Due to the irregularity of graph data (and queries), it is challenging to derive the optimal DOP for each query, while we leave it as an interesting future work.

[7] G_{300} is too large to be held on one machine.

6.3 Our Design Choices

We study our design choices in this experiment by drilling down to the performance factors including bounded-memory execution (Section 5.1), hybrid traversal strategies (Section 5.1) and early-stop optimization (Section 5.2). We run Q6 on G_{300} using the DOP of $[16] \times [4]$, and report the query latency and peak memory usage among all machines. We use Q6 here as it includes complex nested Scopes with fine-grained dependency, and it is a real query in production. We conduct this experiment while adjusting the query parameters m (number of starting vertices), k (the hop limit) and n (the result limit) in Q6, and the system parameter of memory upper-bound of each query (default 10GB) and traversal strategies (default hybrid), and whether early stop is enabled (default enabled). We configure the following variants of GAIA, namely GAIA (default settings), GAIA-DFS (manual DFS-prioritized strategy)[8], GAIA-NoMB (without/infinity memory bound) and GAIA-NoES (without early stop).

Dynamic Scheduling. In this experiment, we study the effectiveness of dynamic scheduling. We vary the memory upper-bound as 256, 512, 1024, 2048, 4096 (MB) and infinity with $m = 10$ starting vertices, and report the result in Figure 13a. The actual memory usage (as labelled) of all cases is very close to the bounded value, and is noticeably smaller than the unbounded case, which has surged to more than 25GB. An interesting observation is that the latency increases with the memory bound. Note that graph traversal exhibits massive parallelism and all the CPU cores available can be fully utilized with just "enough" memory. Additional memory incurs overheads (in allocation, buffering, etc.) rather than benefits.

Traversal Strategy. To verify the effectiveness of the hybrid traversal strategy in GAIA, we compare GAIA with GAIA-DFS/BFS. We vary n from 10 to 10^5, and report the time cost and memory usage in Figure 13b. GAIA-DFS outperforms GAIA when $n \leq 1000$. This is because that DFS strategy will prioritize scheduling operators in the deeper order (in the dataflow), which can potentially escape earlier (thanks to early stop) as soon as n cycles have been found. As n increases, the hybrid strategy gradually catches up with, and eventually outperforms DFS, as it can compute the required number of cycles in a lower order. This experiment shows that the best traversal strategy can be query- (and data-) dependent, and the hybrid strategy is a more generic option.

Early Stop. We compare the performance of GAIA and GAIA-NoES (without early stop). We vary n from 10 to 10^4, and report the query latency and memory usage in Figure 13c. When early stop is turned off, both the query latency and memory usage remain fairly stable, as GAIA always computes all result, regardless of the limit number. When early stop is turned on, it can be observed that both the query latency and memory consumption drop noticeably, compared to the cases without early stop. In particular, the early-stop optimization enables 12× improved performance and $1GB$ memory saving when the limit number is 10.

[8] Note that the BFS-prioritized strategy often causes out-of-memory, and is thus excluded from our test.

Table 3: Comparison GAIA variants with big-data engines.

	GAIA	-DFS	-NoMB	-NoES	Plato	Timely
Lat./Sec.	79	4	440	972	1431	1690
Mem./GB	5.2	0.3	25.6	6.1	108	205

Comparing with Big-Data Engines. Finally, we compare our GAIA with existing high-performance engines, Timely and Plato, in this experiment. We implement Q6 in Timely and Plato[9], which contains 105 and 95 logical lines of codes, respectively. In comparison, the Gremlin query is written in 5 lines as presented in Figure 2. The query latency and memory consumption of these engines, while varying m as 1, 5, 10, 15, 20, is shown in Figure 13d. GAIA achieves 16× and 14× better performance, and consumes 21× and 10× less memory, than Timely and Plato, respectively. To demonstrate how GAIA benefits from the proposed techniques to outperform existing engines, we further bring different variants of GAIA into the comparison, and the results of $m = 10$ are in Table 3. The performance of GAIA drops by 5.5× without memory bound, and by over 12× without early stop, while the latter is already in the same order as those of Plato and Timely. Note that GAIA-DFS even outperforms the default GAIA (hybrid) due to the small result limit ($n = 10$). This experiment shows that the novel design choices of GAIA, notably the Scope abstraction, and the techniques proposed on top of it, enable more convenient programming and efficient execution of the Gremlin queries over big-data engines.

6.4 Comparison with Graph Databases

Small-Scale DB. Although GAIA is designed to scale, we show that GAIA demonstrates efficiency while compared to graph databases on one single machine. Specifically, we use the small graph G_1 so that all the systems can load and process queries in reasonable time; and for each LDBC query, we choose the best query performance among the 4 systems (TinkerGraph, Neo4j, OrientDB and JanusGraph) as the *BSTI* for the query; then we vary the DOP of GAIA, and report the relative performance of GAIA to BSTI in Figure 14.

GAIA performs comparably to the BSTI in most cases except for queries CR-3 (up to 7× worse) and CR-12. Neo4j performs better than any other systems on these queries. Further investigation shows that, instead of faithfully traversing the graph, Neo4j applies a *join* on some partial result to generate the output, which turns out to be more efficient in these cases. We leave better query optimization of Gremlin on GAIA as future work. As a whole, GAIA has an average relative performance of just around 1.8 using single thread, and of 0.73 using 16 threads, among all LDBC queries.

Large-scale DB. We use G_{100} in this experiment to run all LDBC queries. Note that we only compare JanusGraph, as

[9] For fair comparison, we implement cycle detection in Timely and Plato using the same algorithm as in GAIA. In addition, we exploit all possible optimizing options from both systems for the test.

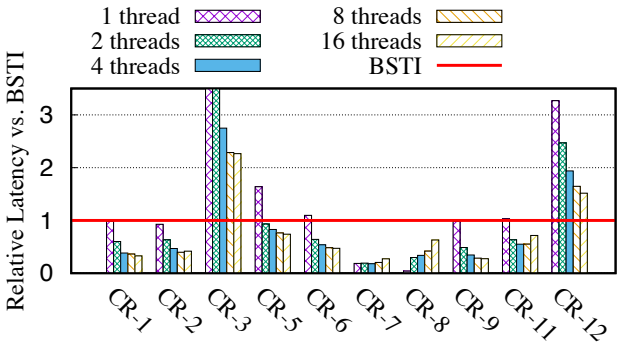


Figure 14: GAIA performance relative to the best single-threaded implementation (BSTI).

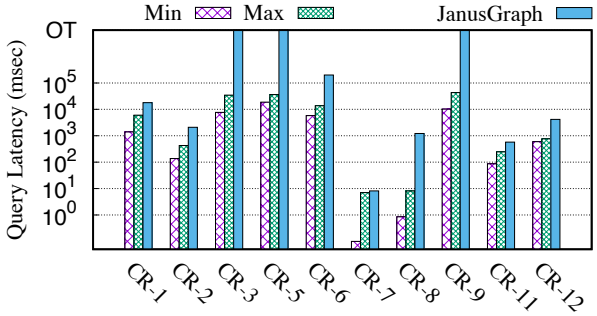


Figure 15: Compare GAIA with JanusGraph.

it is the only system that can store graph at this scale. JanusGraph cannot process query in parallel, and we run GAIA in one machine for fair comparison. The graphs are stored in 8 machines for JanusGraph, and one single machine[10] for GAIA. We run each query on GAIA with DOP varying from 1 to 16, and report its max and min latency for each query while compared to JanusGraph. The result is reported in Figure 15. JanusGraph fails to answer many queries (CR-3,5,9) due to OT. As shown, even the maximum latency (single-thread) of GAIA is much shorter than that of JanusGraph in all cases. Although GAIA is designed to scale in a cluster, it can further benefit from multi-core parallelism in a single machine to improve query performance, especially for large queries, as can be seen in Figure 15.

7 Related Work

Graph Databases. Gremlin is widely supported by many graph databases, such as Neo4j [29], OrientDB [30], JanusGraph [21], and cloud-based services including Cosmos DB [6] and Neptune [1]. However, their query processing is limited to one single process. Driven by rapidly growing needs to query large graph data, several distributed in-memory graph systems emerge, such as Trinity [40], ZipG [24], Wukong+S [48], Grasper [20], and A1 [9]. Trinity and ZipG offer their own programming models that are less flexible than Gremlin. Grasper adopts Gremlin but provides a limited subset of the language constructs (e.g., the lack of nested-loop support). Wukong+S and A1 leverage RDMA for serving *micro-second* queries with much higher concurrency, which is not the main target scenario of GAIA.

Graph Processing Systems. In contrast to many other systems that deal with batch-oriented iterative graph processing, such as Pregel [26], PowerGraph [15], GraphX [16], and Gemini [49], GAIA focuses on low-latency graph traversal at scale. It is hard to support graph traversal in existing graph processing systems. Firstly, their programming abstractions [22] are usually low-level, makes these systems a privilege for experienced users only [13]. Moreover, they typically adopt the bulk synchronous parallel (BSP) execution model, which is more suitable for an iterative *routine* processing over the *whole* graph, but can be inefficient for running graph traversal that visits an *arbitrary portion* of the graph.

Dataflow Engines and Dependency Tracking. A number of existing systems such as CIEL [28], Naiad [27], and TensorFlow [45] offer generic data-parallel computing infrastructures with support for dynamic control flow. While it is possible to program the logic of a Gremlin query on top of these frameworks, it is extremely challenging to do so in the pursuit of both correctness and efficiency, largely due to the fine-grained dependency in Gremlin traversal. Tracking dependency has been exploited to compute what is absolutely necessary when there are limited changes to the input (e.g., incremental computing as in Incoop [7], DryadInc [33], Nectar [19]), or frugal re-computation to repair lost state as in MadLINQ [34] and TimeStream [35].

Declarative Programming Languages. Graph queries are typically expressed using graph traversal and pattern matching. Correspondingly, Gremlin [37] and Cypher [14] are the most popular query languages. Cypher allows users to specify a graph pattern with variables. However, based on our production experience, it is often challenging to compose ad-hoc query pattern for a particular task. Therefore, we support Gremlin instead of Cypher in this work. Other notable research projects in parallel declarative languages, such as Cilk [8], can be leveraged by GAIA in theory, but they are not particularly tailored for distributed graph traversal.

8 Conclusion

GAIA has been in use by a small community of domain experts for over a year in production at Alibaba. Our overall experience is that GAIA, by combining the benefits of Gremlin with the power of distributed dataflow execution, proves to be a simple, useful and efficient programming environment for interactive analysis on big graph data.

Acknowledgments

We thank Benli Li, Pin Gao, and Donghai Yu for answering Plato related questions. We are grateful to Alibaba GraphScope team members for their support. Thanks also to the NSDI review committee, as well as our shepherd Anurag Khandelwal, for their valuable comments and suggestions.

[10] Note that JanusGraph is properly warmed up to reduce the cost of pulling data from remote storage.

References

[1] Amazon Neptune. https://aws.amazon.com/neptune/. [Online; accessed 2-March-2021], 2019.

[2] Renzo Angles and Claudio Gutierrez. Survey of Graph Database Models. *ACM Comput. Surv.*, 40(1), February 2008.

[3] Apache TinkerPop. http://tinkerpop.apache.org/. [Online; accessed 2-March-2021], 2019.

[4] Ching Avery. Giraph: Large-Scale Graph Processing Infrastructure on Hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 11(3):5–9, 2011.

[5] Konstantin Avrachenkov and Nelly Litvak. The Effect of New Links on Google PageRank. *Stochastic Models*, 22(2):319–331, 2006.

[6] Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/. [Online; accessed 2-March-2021], 2019.

[7] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: MapReduce for Incremental Computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[9] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, Matthew Renzelmann, Alex Shamis, Timothy Tan, and Shuheng Zheng. A1: A Distributed In-memory Graph Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 329–344, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The Impact of Synchronization and Granularity on Parallel Systems. *SIGARCH Comput. Archit. News*, 18(2SI):239–248, May 1990.

[11] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the Pulse of a Fast-Changing and Connected World. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 85–98, New York, NY, USA, 2012. ACM.

[12] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 619–630, New York, NY, USA, 2015. ACM.

[13] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, and Jiaxin Jiang. Grape: Parallelizing Sequential Graph Computations. *Proceedings of the VLDB Endowment*, 10(12):1889–1892, 2017.

[14] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445, New York, NY, USA, 2018. ACM.

[15] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[16] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 599–613, 2014.

[17] Gremlin Console. http://tinkerpop.apache.org/docs/3.4.3/reference/#gremlin-console. [Online; accessed 2-March-2021], 2019.

[18] Gremlin Server. http://tinkerpop.apache.org/docs/3.4.3/reference/#connecting-gremlin-server. [Online; accessed 2-March-2021], 2019.

[19] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI*, volume 10, pages 1–8, 2010.

[20] Chen Hongzhi, Li Changji, Fang Juncheng, Huang Chenghuan, Cheng James, Zhang Jian, Hou Yifan, and Yan Xiao. Grasper: A High Performance Distributed System for OLAP on Property Graphs. In *ACM Symposium on Cloud Computing 2019*, Socc'19, 2019.

[21] JanusGraph. http://janusgraph.org/. [Online; accessed 2-March-2021], 2019.

[22] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. High-Level Programming Abstractions for Distributed Graph Processing. *CoRR*, abs/1607.02646, 2016.

[23] U Kang, Mary McGlohon, Leman Akoglu, and Christos Faloutsos. Patterns on the Connected Components of Terabyte-Scale Graphs. In *2010 IEEE International Conference on Data Mining*, pages 875–880. IEEE, 2010.

[24] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. ZipG: A Memory-Efficient Graph Store for Interactive Queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1149–1164, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Andrea Lattuada, Frank McSherry, and Zaheer Chothia. Faucet: A User-level, Modular Technique for Flow Control in Dataflow Engines. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, BeyondMR '16, pages 2:1–2:4, New York, NY, USA, 2016. ACM.

[26] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[27] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[28] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A Universal Execution Engine for Distributed Data-Flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 113–126, USA, 2011. USENIX Association.

[29] Neo4j. https://neo4j.com/. [Online; accessed 2-March-2021], 2019.

[30] OrientDB. https://orientdb.com/. [Online; accessed 2-March-2021], 2019.

[31] Thomas M Parks. Bounded Scheduling of Process Networks. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, 1995.

[32] Plato: A Framework for Distributed Graph Computation. https://github.com/Tencent/plato. [Online; accessed 2-March-2021], 2020.

[33] Lucian Popa, Mihai Budiu, Yuan Yu, and Michael Isard. DryadInc: Reusing Work in Large-Scale Computations. *HotCloud*, 9:2–6, 2009.

[34] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. MadLINQ: Large-Scale Distributed Matrix Computation for the Cloud. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 197–210, 2012.

[35] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 1–14, 2013.

[36] Rodrigo Caetano Rocha and Bhalchandra D Thatte. Distributed Cycle Detection in Large-Scale Sparse Graphs. *Proceedings of Simpósio Brasileiro de Pesquisa Operacional (SBPO'15)*, pages 1–11, 2015.

[37] Marko A. Rodriguez. The Gremlin Graph Traversal Machine and Language (Invited Talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, pages 1–10, New York, NY, USA, 2015. ACM.

[38] Rust Programming Language. https://www.rust-lang.org/. [Online; accessed 2-March-2021], 2020.

[39] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2013.

[40] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 505–516, New York, NY, USA, 2013. Association for Computing Machinery.

[41] The HIVE project. http://hadoop.apache.org/hive/. [Online; accessed 2-March-2021], 2020.

[42] TigerGraph. https://www.tigergraph.com/benchmark/. [Online; accessed 2-March-2021], 2018.

[43] Timely Dataflow. https://github.com/TimelyDataflow/timely-dataflow. [Online; accessed 2-March-2021], 2019.

[44] Leslie G Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[45] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. Dynamic Control Flow in Large-Scale Machine Learning. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 18:1–18:15, New York, NY, USA, 2018. ACM.

[46] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-parallel Computing Using a High-Level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[47] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, April 2012. USENIX Association.

[48] Yunhao Zhang, Rong Chen, and Haibo Chen. Sub-millisecond Stateful Stream Querying over Fast-Evolving Linked Data. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 614–630, New York, NY, USA, 2017. ACM.

[49] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 301–316, 2016.