PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine

Nicholas P. Roth
Vasileios Trigonakis
Sungpack Hong
Hassan Chafi
Oracle Labs
{first.last}@oracle.com

Anthony Potter
Boris Motik
Ian Horrocks
University of Oxford
{first.last}@cs.ox.ac.uk

Abstract

Graph querying and pattern matching is becoming an important feature of graph processing as it allows data analysts to easily collect and understand information about their graphs in a way similar to SQL for databases. One of the key challenges in graph pattern matching is to process increasingly large graphs that often do not fit in the memory of a single machine. In this paper, we present PGX.D/Async, a scalable distributed pattern matching engine for property graphs that is able to handle very large datasets. PGX.D/Async implements pattern matching operations with asynchronous depth-first traversal, allowing for a high degree of parallelism and precise control over memory consumption. In PGX.D/Async, developers can query graphs with PGQL, an SQL-like query language for property graphs. Essentially, PGX.D/Async provides an intuitive, distributed, in-memory pattern matching engine for very large graphs.

1 Introduction

Graph processing is becoming an integral part of big-data analytics. This is partially because graphs can naturally represent data that captures fine-grained relationships among entities. Graph analysis can provide valuable insights about such data by examining these relationships.

Typically, graph analysis is performed with two distinct but correlated methods, namely *computational analysis* and *pattern matching queries*. With computational analysis, the user runs various algorithms that traverse the graph, often repeatedly, and calculate certain (numeric) values to get the desired information, e.g., PageRank or shortest paths [21]. Pattern matching queries are declaratively given as graph patterns. The system finds every subgraph of the target graph that is topologically isomorphic/homomorphic to the query graph and satisfies any accompanying filters. For example, the following PGQL [27] query:

```
SELECT a, b WHERE (a WITH age > 18)-[:friend]->(b)
```

returns all pairs of vertices `a, b`, with `a.age > 18`, where `a` is connected to `b` with a `friend` edge.

In this paper, we introduce PGX.D/Async, a scalable, distributed pattern-matching engine for property graphs. As the name suggests, PGX.D/Async extends PGX.D, a computational graph analysis framework [18], adding graph pattern matching features. PGX.D/Async accepts PGQL [27] query as its input.

Existing systems typically take one of two approaches for implementing graph pattern matching. One such approach processes queries in a similar way to relational databases, with joins to traverse edges (e.g., [14]). Another technique interprets the queries on the graph and runs each operator separately in a breadth-first manner (e.g., [24]). These two approaches are not ideal for distributed systems, for they result in (i) high memory utilization due to potential intermediate state explosion [28] and (ii) low performance due to extensive, eager communication [17].

Instead, PGX.D/Async adapts a query evaluation strategy based on *asynchronous depth-first traversal* [23] to the property graph setting and adds dynamic memory management for controlling message buffers. PGX.D/Async achieves controllable high degrees of parallelism (for good performance and scalability) as well as precise flow control with a deterministic guarantee of query completion under a finite amount of memory. In detail:

- **Depth-First Traversal (DFT)** allows for a highly parallel, scalable, and asynchronous model of execution. DFT results in fewer intermediate results than breadth-first traversal, while allowing for controllable parallelism and memory consumption. Asynchrony helps improve the performance of queries on distributed graphs by using work from other stages to hide the effects of workload imbalance and communication latency within a stage.
- **Strict, Precise Flow Control** ensures that any query can execute with a limited amount of memory. If at any point in the execution, the next operation of a graph traversal would violate flow control, the corresponding worker thread can just suspend this operation and work on a different one (i.e., a later vertex match in the path or an incoming message from another machine).

In brief, every PGQL query in PGX.D/Async is broken into *stages*. Each stage is responsible for visiting / matching one vertex. The execution of a query proceeds sequentially—for depth-first traversal—and transitions to the next stage via *hop engines*. A hop engine (e.g., an in- or out-neighbor match) is responsible for selecting the next vertex to hop to based on any filters in the query. For example, Figure 1 shows how a simple PGQL query is translated into PGX.D/Async stages. In stage 0, vertex `p` is matched (if `p` has property `age` with `age < 18`). Then, an out-neighbor hop engine is used to traverse to the targets of stage 1. Both stages 0 and 1, as well as the hop engine, add to the output context, such that if there is finally a match, the last stage can forward the query output.

To support asynchronous execution, each stage contains the current intermediate output and the context for continuing the execution

GRADES'17, Chicago, IL, USA

978-1-4503-5038-9/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3078447.3078454

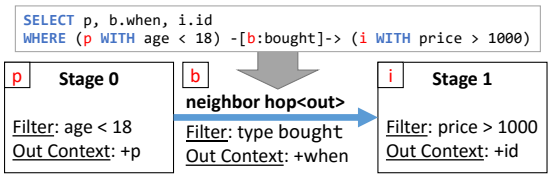

Figure 1. Example: From a PGQL query to PGX.D/Async stages.

of the query for each worker thread in the computation. Consequently, if a worker on stage 0 of Figure 1 needs to hop to a neighbor on a different machine for stage 1, it simply transmits the whole context of its current stage. This worker, instead of blocking to wait for a response, can continue processing operations on other stages of the same query instance submitted internally to facilitate work-sharing or externally by remote machines.

The rest of the paper is organized as follows. In Section 2, we discuss some background and existing work related to graph analytics and graph pattern matching. We describe the design and implementation of PGX.D/Async and show evaluation results in Sections 3 and 4 respectively. Finally, in Section 5 we conclude the paper and discuss future directions for PGX.D/Async.

2 Background and Related Work

RDF vs. Property Graph. There are two different kinds of graph databases: First, there are graph databases [1, 9, 12, 17, 23, 28] that adopt the classic *RDF* data model. The RDF model, regularizes the graph representation as a set of triples. In the RDF model, even constant literals are encoded as graph vertices, and such artificial vertices can induce noise to the outcome of the graph analysis and overhead for graph querying. Second, there are graph databases [5, 7, 24] that adopt the *Property Graph* (PG) data model where nodes and edges in a graph can be associated with arbitrary *properties* as key-value pairs. PGX.D/Async is an efficient distributed in-memory query pattern matching for property graphs.

Graph Query Languages. PGX.D/Async adopts PGQL [27] as a front-end graph query language, while there are a few other alternatives. SPARQL [11] is a query language for RDF model, standardized by the World Wide Web Consortium. For property graph, to the contrary, there is no standard query language. Cypher [4] is a popular graph query language for property graph; however it is missing some fundamental graph querying functionalities, such as regular path queries and graph construction [27].

Graph Pattern Matching. As we mention in the previous section, there are two approaches for implementing graph pattern matching, namely joins and breadth-first traversals (BFTs) on the graph. For example, GraphFrames [14] implements distributed graph pattern matching on top of Apache SPARK's dataframes: One dataframe for vertices and another for edges. Consequently, a stage for matching an edge is naturally mapped into a join operation.

However, these approaches can cause performance issues in distributed execution. First, they both result in a potentially high maximum memory utilization due to the volume of intermediate results and states. Extending a pattern with BFTs/joins can result in exponentially many active intermediate results. In contrast, with depth-first traversals, each worker in PGX.D/Async tries to complete a query instance before starting a new one, thus reducing the number of active intermediate results. Second, with BFTs/joins, communication across the machines might be in the critical path. To traverse one edge all machines in the cluster could be involved. In contrast, communication in PGX.D/Async happens asynchronously—when a traversal crosses the borders of a machine, the worker that initiated the traversal simply sends a message to the target machine and continues with performing some other local computation.

We are not the first to recognize that asynchrony and depth-first traversal are suitable for distributed graph pattern matching. The authors of Trinity.RDF [28] point out that traditional RDF join approaches result in excessive intermediate results and propose a solution where multiple triple joins are eagerly evaluated (in a depth-first manner). Similarly, Triad [17] uses asynchronous communication to execute independent "leaves" of the query in parallel.

Potter et al. [23] extend these ideas to design a query engine for distributed RDF graphs. Their work extends the data exchange operator, originally introduced in Volcano [16], with dynamic data exchange, an approach that detects whether a join of RDF triplets requires communication across machines. Execution proceeds asynchronously in each machine (communication happens for completing intermediate queries across machines) and flow control ensures query completion within memory bounds. In this paper we bring these ideas to property graphs, extending them with dynamic memory management for message buffers, presenting an implementation of the resulting algorithm and evaluating its effectiveness.

Graph Libraries and Graph Processing Frameworks. There are many systems that focus on the fast execution of graph analysis algorithms. First, there exists many different graph libraries for different programming languages. Examples include NetworkX [8] (Python), Jung [6] (Java), Stinger [15] and Boost Graph Library [3] (C++). Essentially, these libraries provide their own graph data structures; for custom algorithms the users should come up with their own parallel implementation on top of the library's data structures. Second, there are many (distributed) graph processing frameworks which consider parallel or distributed execution of graph algorithms [2, 19–22, 26].

We implement PGX.D/Async in the PGX.D distributed graph analytics system [18]. PGX.D implements a relaxed version of the bulk-synchronous model, where graph algorithms proceed with global steps. This execution model is suitable for algorithms, such as PageRank, that iteratively traverse the (whole) graph. Essentially, PGX.D/Async augments PGX.D with a complementary, asynchronous execution model. PGX.D/Async directly builds on top of the task and data management components of PGX.D.

3 PGX.D/Async: A Distributed Graph Pattern Matching and Querying Engine

PGX.D/Async accepts PGQL queries as its input. On a high level, the execution of a query proceeds in four steps (the boxes of Figure 2):

i. PGX.D/Async compiles the PGQL into an query plan (using the standard PGQL compiler [10]);
ii. PGX.D/Async transforms the query plan into a distributed query plan, which handles some intricacies of querying a distributed graph;
iii. PGX.D/Async translates the distributed query plan into an execution plan in which every stage—each responsible for matching or visiting one vertex—has highly structured and well-defined inputs, outputs, and constraints; and

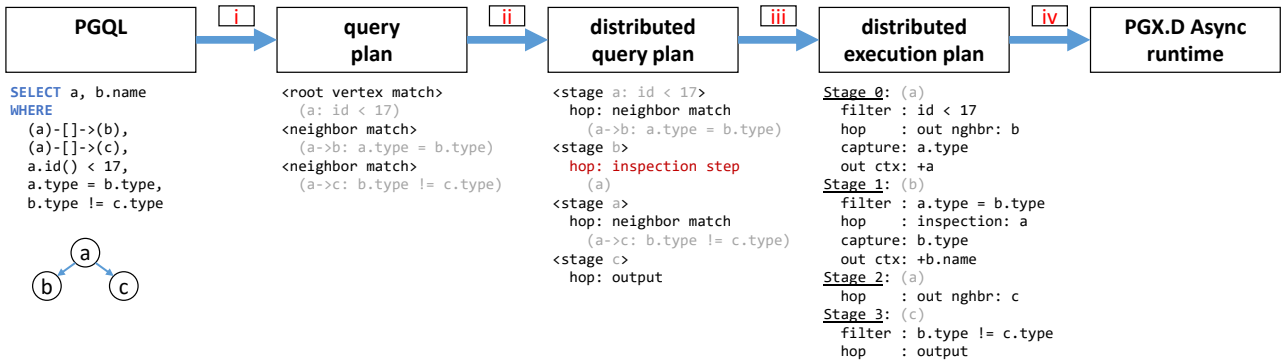

Figure 2. Example: From a PGQL query to PGX.D/Async stages.

iv. PGX.D/Async compiles the execution plan into memory compact data structures and launches the computation.

In what follows, we first describe how PGX.D/Async transforms the query plan to an execution plan and then we detail the anatomy of a stage and the design of PGX.D/Async's runtime.

3.1 From Query Plan to a Distributed Execution Plan

In step i, PGX.D/Async translates the PGQL query to a generic query plan, which comprises graph matching operators, such as *(constant) vertex*, *neighbor*, and *edge match*. This plan is designed for shared-memory execution—i.e., it assumes that operators can access the properties of every vertex—and can be used as-is in PGX. This assumption is not true in a distributed environment: A later operator cannot directly access the properties and the neighbors of "earlier" vertices as they might reside on a remote machine. For instance, the query plan of Figure 2 includes a final neighbor match operator for `a -> c`, which comes after matching `a -> b`. In shared memory, implementing such operation is inexpensive, because the runtime can simply access the edges of `a`. In contrast, in a distributed setting, the computation would potentially need to move across machines in order to visit vertex `a` again. Similarly, both neighbor match operators compare properties of the target vertex (e.g., `b` in `a -> b`) with properties of the source vertex. As earlier, these properties are not easily accessible in a distributed setting.

PGX.D/Async removes these limitations in steps ii and iii. In step ii, PGX.D/Async modifies the query plan to account for such situations and transform the list of match operators into a list of *stages* with *hop engines*. In our example, the distributed query plan includes an inspection step to ensure that the query execution will return the context to vertex `a` in order to match the edge to `c`.

Then, in step iii, PGX.D/Async translates the distributed query plan to a distributed execution plan. This process also chooses the memory layout of the context object for each stage and binds variables used by hop engines and filters to offsets in the context or directly-accessible graph properties. Furthermore, PGX.D/Async performs dependency analysis so that earlier stages keep enough context for later stages to be able to complete without remote communication. In the example of Figure 2, stage 0 adds `a.type` to the context so that stage 1 can use it in its filter. Similarly, stage 1 adds `b.name` to the context so that it can be outputted upon finding a complete match. Essentially, the distributed query plan makes certain that the required vertices are accessible, while the execution plan ensures that data dependencies are efficiently fulfilled.

Finally, in step iv, the PGX.D/Async runtime executes the execution plan—described in Section 3.3.

3.2 Anatomy of a Stage

A stage represents the processing of a single vertex, except for inspection stages which are used for re-visiting already processed vertices. Stages advance sequentially, in accordance with PGX.D/Async's depth-first traversal. Each stage also contains the necessary components to allow for:

1. Matching vertices according to filters (e.g., `a.age > 18`);
2. Transitioning from one vertex (a stage) to the next one (next stage) in well-defined ways (hop engine);
3. Performing flow control;
4. Handling pause and resume logic for worker threads, in case they have to block the processing of a stage; and
5. Delivering messages to the next stage.

In what follows, we describe each component individually.

Vertex Function. The *vertex function* is responsible for (i) applying the PGQL vertex filters, and (ii) building the parts of the output context of the stage that correspond to vertex properties (the vertex filter does not handle edge data).

Hop Engine. A *hop engine* defines how to traverse to the next vertex. Various hop engines implement traversal for specific patterns:

- *Vertices*: Hop to one or more vertices. Inspection stages are implemented by configuring this hop engine to hop to a single vertex, as discussed above.
- *Out neighbors*: Hop to the outgoing neighbors of the current vertex. For example, `(a) -[]-> (b)` defines such a hop, assuming that the stage of `a` is earlier than the one of `b`.
- *In neighbors*: Hop to incoming neighbors of the current vertex. For example, `(a) <-[]- (b)` defines such a hop, assuming that the stage of `a` is earlier than the one of `b`.
- *Common neighbors*: Hop to the common neighbors of two vertices. For example: `(a) -[]-> (b) <-[]- (c)` requires to visit the common neighbors between `a` and `c`.
- *Output*: Pass the output context of the current stage to the collected global output. This is always the hop engine of the last stage in a query.

Note that these hop engines offer some flexibility to the query plan generator. For instance, a `(a) -[]-> (b)` out-neighbors hop can be transformed to a `(b) <-[]- (a)` in-neighbors hop. Similarly, the common neighbors hop can be translated and implemented

as separate out/in neighbor hops. As we discuss in Section 5, we include the common neighbors hop engine in order to allow for implementing efficient distributed common neighbors algorithms.

In the exact same way as the vertex function of a stage, the hop function of a hop engine can include filters on information specific to the hop engine (e.g., edge filters) and add relevant information to the output context of the stage.

Message Manager. The *message manager* handles incoming and outgoing messages, essentially providing messaging queues to worker threads. These messages include both work and acknowledgment messages. Each stage has a message manager, with more than one outgoing slots per worker. Outgoing messages from a stage are delivered to the next stage on a different machine. Messages received by the message manager are picked up by idle workers and the computations that they denote are performed to completion or until flow control prevents them from advancing.

The message manager tracks whether the queues of a worker are full. In the latter case, when the thread cannot send a message due to full queues, it blocks the computation for the current stage and performs computation either queued up from incoming messages or work sharing pertaining to stages other than the stage at which the current computation stack began. The message manager protects workers from adding additional computations to already-full bulk messages and thus implicitly performs the first form of flow control in PGX.D/Async, disallowing worker threads from having arbitrarily many undelivered messages.

Flow Control Manager. The *flow control manager* controls the memory requirements of computations by tracking the number of unprocessed messages from one machine to another for each stage, thus allowing any computation to have finite memory requirements. Essentially, the flow control manager might delay the transmission of messages to specific machines in order to avoid overloading those machines. Thus, the user of PGX.D/Async can configure the amount of memory that each machine can use. We describe the implementation of "global" flow control later in Section 3.3.

3.3 Runtime

The runtime consists of a number of worker threads per machine (typically slightly less than the number of hardware contexts of each machine). These workers are initialized by the task manager [18] on system boot time. Computations in PGX.D are defined as tasks and are placed in task queues. For example, launching the asynchronous computation is a single task, while another task represents waiting for the computation to finish and handling incoming asynchronous work messages. These are the only tasks used by PGX.D/Async. For the most part, PGX.D/Async ignores the synchronous task model of PGX.D, layering its own model on top of these two synchronous supersteps ("bootstrap" and "await completion"). In this text, a *computation* shall refer to the in-place depth-first traversal of the graph within a single machine by one or more PGX.D/Async stages.

It is also important to note that we design PGX.D/Async to operate in the same situations as PGX.D, augmenting it with query answering capabilities. In a multi-tenant enterprise system such as PGX.D, precisely tracking resource usage while ensuring complete results is extremely important. Furthermore, perhaps most importantly, as the distributed facet of PGX, PGX.D/Async must be able to operate with limited resources on arbitrarily large graphs. In light

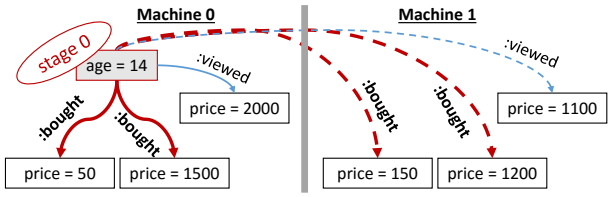

Figure 3. Query of Figure 1: Moving from stage 0 to stage 1. Only the thick, red edges will be followed by the hop engine of stage 0.

of these considerations, we dedicate a substantial part of this section to a disucssion on flow control and memory management.

The execution starts by adding one of two types of bootstrap tasks to the task manager, depending on whether the query origin is a single (e.g., `(v WITH id() = 17)-[]->()`) or multiple vertices (e.g., `(v)-[]->()`). The former task is explicitly created by calling the `apply()` function of stage 0 once on the machine where the origin vertex resides, while the latter calls the `apply()` function of stage 0 on each vertex in the graph, at which point the vertex function eliminates vertices that fail to satisfy any filters. For all other stages, work comes from a recursive call to `apply()` or a message. Both scenarios occur when workers call `apply()` to move to later stages of the computation. For example, Figure 3 depicts a potential configuration of the query in Figure 1:

```
... (p)-[:bought]->(i), p.age < 18, i.price > 1000
```

In the example, there is already a vertex match in machine 0 for stage 0. Subsequently, the hop engine of stage 0 will only select the `:bought` edges to follow, resulting in both local computation and messaging to machine 1 with the task of continuing this computation.

To bootstrap a query in PGX.D/Async, workers call `apply` on the instantiated runtime class corresponding to stage 0 for each starting vertex of the computation. Initially, the workers attempt to execute stage 0 in-place in addition to recursively executing subsequent stages on matched vertices. When the computation blocks because of flow control or full messaging queues, this line of computation is paused. Instead of idling, the worker repeatedly calls `DoWork` (Figure 4) on stage 0 while periodically checking if it can continue the bootstrapping process. After a worker finishes its bootstrapping task, it runs a second PGX.D task that repeatedly calls `DoWork` to handle messages until all machines finish the query.

`DoWork` handles incoming messages (intermediate results) for the current and the subsequent stages in *descending* order to maximize performance. Intermediate results produced by later stages are expected to produce less net work than those produced by earlier stages and therefore require fewer future resources. This improves performance and reliability of the messaging layer by alleviating memory pressure and avoiding message floods. Note that the functions `TryResume` and `FinishWork` in Figure 4 attempt to retry the currently-blocked operation and finish running the current stage on each intermediate result in the current message, respectively. Both functions return `true` upon success.

Termination Protocol

Detecting the termination of a query (or even of a single stage) is not straightforward in the asynchronous setting of PGX.D/Async. Earlier stages might send messages to the "current" stage, thus triggering more computation (or delivering more results). We solve this problem using a lightweight termination protocol which incrementally detects the termination of each stage. The protocol in PGX.D/Async

is an adaptation of the termination protocol first described by Potter et al. [23] in the context of query pattern matching on RDF graphs.

In summary, termination is detected by tracking the per-machine completion of each stage in the query. To achieve this behavior, machines track the completion of a stage locally and exchange special `COMPLETED` messages once a stage is completed. In particular, machine k can finish processing stage n if it knows that all machines have finished processing all stages up to $n-1$; and the machine has processed all received messages for this stage. At this point, machine k sends a `COMPLETED` message to all other machines informing them that they will not receive further messages from k for stage n. Essentially, the termination of a query is performed incrementally: Stage 0 cannot receive any messages from previous stages, hence machines can send their first `COMPLETED` message as soon as they finish bootstrapping the query. Then, stage 1 can complete after all stage 0 `COMPLETED` messages are received and the the machine has finished processing all messages delivered to stage 1, etc.

Flow Control

To maintain a strict memory bound while satisfying PGX.D/Async's termination condition, each stage n is independently restricted such that on any machine m, no more than $a_{n,m}$ unprocessed messages can be in transit to or stored for that stage.[1] This limit is enforced by the flow control on stage $n-1$ of the sending machine (using a counter). In a simple implementation, this counter can have a constant maximum $b_{n,m} := \frac{a_{n,m}}{M-1}$ on a cluster with M machines. When the counter reaches $b_{n,m}$, flow control does not let any more messages pass to stage n on machine m. When machine m acknowledges that it finished processing one or more messages originating at the current machine (and therefore does not need to preserve their contents in memory), the counter $a_{n,m}$ on the current machine is decremented by the number of messages acknowledged.

Maximizing Memory Efficiency of Flow Control. This lightweight form of flow control works well in many situations (e.g., when the workload is balanced across machines). However, large clusters and computations with many stages could require an exorbitant amount of memory to achieve the expected performance. The following modifications selectively allow counters to steal capacity from each other to improve memory efficiency without sacrificing the flow-control precision and performance of the original algorithm:

1. After the termination algorithm detects that a stage n is finished, stage n's capacity $a_{n,m}$ is distributed among $\{a_{n+1,m}, a_{n+2,m}, ..., a_{N-1,m}, a_{N,m}\}$
2. During a computation, each stage on machine m can request additional bandwidth from the same stage on machine m_s.

Note that dynamic memory management improves the utilization of the memory used for message buffers over the previous flow control mechanism [23].

[1] On our homogeneous cluster, we set the same $a_{n,m} := a_n$ on each machine.

```
                                  ▷ Do outstanding work on a stage n and a worker w
function DoWork(n, w)
    if n < N_STAGES then          ▷ Operate on the latest stage possible
        DoWork(n + 1, w)
    end if
    if State[n,w].blocked then
        if TryResume(n, w) ∧ FinishWork(n, w) then
            State[n,w].blocked ← False
        end if
    else if GetMessage(n) ∧¬FinishWork(n, w) then
        State[n,w].blocked ← True
    end if
end function
```

Figure 4. Continuing the work of a suspended stage.

4 Evaluation

In this section, we describe some evaluation results with PGX.D/Async. The main goals of this evaluation are to illustrate that PGX.D/Async (i) indeed succeeds in analyzing large graphs, much larger than what we can fit in a single machine, and (ii) scales with the number of machines, thus enabling the analysis of increasingly larger graphs using more processors.

Experimental Settings. We perform our evaluation on a cluster of 32 identical machines, connected with an InfiniBand network (with a Mellanox SX6512, 54.54 Gb/s per port switch). The machines use Linux 2.6.32 (OEL 6.5) and gcc 4.9.0. Each machine is a 2-socket Intel Xeon E5-2660 (2.20 GHz) with 8 cores (16 hyperthreads) each, 256 GB DDR3 RAM, and a Mellanox Connect-IB network card. Note that our cluster uses InfiniBand, however, the implementation of PGX.D (and PGX.D/Async) is independent from the underlying interconnection fabric, as it does not exploit any of its special features (e.g., RDMA). Still, the underlying messaging library utilizes RDMA—when available—for performance.

For the experiments in this paper, we disable the ghost nodes functionality of PGX.D. Additionally, we execute each set of queries five times, after one extra warmup execution after loading the graph and report the median execution time (note that we observe low deviation in our results). The partitioning of vertices to machines is random, except that the system attempts to distribute a similar number of edges to each machine. Finally, we set the flow control limit of PGX.D/Async to use up to 17 GB of memory per machine.

4.1 Experimental Results

Comparing to a Single Machine. For our first experiment, we use the BSBM SPARQL e-commerce benchmark [13]. We transform the generated RDF graph to a property graph and we rewrite the queries to PGQL. We use an eight million products set, which corresponds to a graph with 250 million vertices and almost one billion edges—the graph tightly fits in the main memory of a single machine. We evaluate PGX.D/Async with the 10 parts of BSBM query 5.

Figure 5 includes the results, normalized to the per-query execution time of PGX on a single machine. Clearly, some queries, such as P8 and P9, do not scale in PGX.D/Async. The reasons are that (i) these queries have inherently limited parallelism and they are very short, and (ii) we have not yet implemented the intra-machine workload balancing capabilities described above. Essentially, this lack of scalability is mainly due to the overheads of a distributed

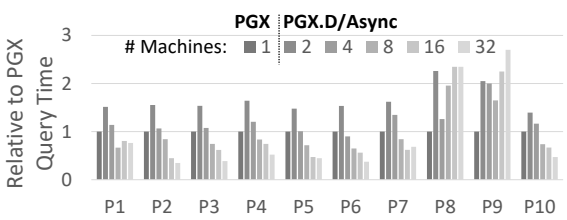

Figure 5. Relative (to PGX) time to complete BSBM queries.

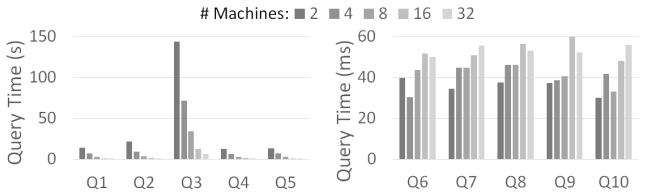

Figure 6. Time to complete queries on a random graph.

system: For example, PGX takes 3 ms to complete a tiny query on a tiny graph, compared to 37 ms of PGX.D/Async on two machines, and more than 50 ms on 32 machines. On the remaining queries, we see that PGX.D/Async scales with the number of machines and outperforms the single machine PGX results.

Querying a Very Large Graph. In our second experiment, we use an artificial uniformly random graph of 200 million vertices and two billion edges to show that, indeed, PGX.D/Async can handle graphs that do not fit in the memory of a single machine. We evaluate 10 randomly selected queries, with four edge patterns each. Figure 6 includes the time to complete these queries on 2–32 machines. We separate the queries into heavy queries (with runtime in the scale of seconds) and fast queries. Evidently, PGX.D/Async achieves very good scalability on the heavy queries, since there is enough work to leverage the additional machines. In contrast, for small queries, we observe similar behavior as in our BSBM experiments: Adding more machines does not bring any benefits and, as expected, using more machines introduces some overhead.

5 Conclusion and Future Work

We presented the PGX.D/Async distributed engine for efficient graph querying and pattern matching. PGX.D/Async employs asynchronous depth-first traversal, thus avoiding problems, such as large intermediate result sets that could run a system out of memory, while improving the potential for parallelism in certain situations. We implemented PGX.D/Async in PGX.D and maintained compatibility with the PGX shared-memory graph analytics framework. This way, data analysts can reuse their existing PGQL scripts to analyze ever-larger graphs that do not fit in the main memory of a single machine. As we showed in our evaluation, PGX.D/Async can load and query graphs that require hundreds of gigabytes of memory.

Future Work. We have currently implemented the basic PGQL filtering and pattern matching capabilities in PGX.D/Async in order to show that the asynchronous depth-first approach is indeed a good generic solution for distributed graphs. In the future, we intend to extend our PGX.D/Async implementation in several directions.

Complete PGQL Support. We will implement the PGQL functionality is currently missing in our PGX.D/Async implementation. This includes aggregations (e.g., `COUNT`, `SUM`), grouping (i.e., `GROUP BY`), sorting (i.e., `SORT`), and recursive paths.

Graph Isomorphism. PGX.D/Async currently only supports graph homomorphism. With homomorphism, two graphs can be matched, even though they do not have the same number of vertices. Isomorphism requires that there is a one-to-one mapping of the edges of one graph to another. For isomorphism, the runtime must not only check the edges that are matched, but also that there are no edges that connect the matched vertices and do not belong to the pattern.

Common Neighbors. Patterns like `(a) -[]-> (c) <-[]- (b)` enumerate the common neighbors of `a` and `b`, which is an expensive operation in a distributed setting. We intend to optimize the runtime with specialized common neighbor operators [25] which calculate common neighbors by simply exchanging the edges of one another.

Query Scheduling. Depending on the actual properties of the graph, the sequence with which a pattern is matched can play an important role in performance. For example, in the following query:

```
SELECT person, band WHERE
  (person)-[:likes]->(song)-[:from]->(band),
  person.gender = "female", song.style = "rock",
  band.name = "Uknown1"
```

we would prefer to start by matching the vertex `band` as it (probably) has the lowest selectivity. In practice, having this information about the graph structure ahead of time and knowing how to use it quickly and efficiently is very difficult. Still, with data collection during graph loading and runtime, we believe that it is possible to significantly improve query performance.

References

[1] AllegroGraph. http://franz.com/agraph/allegrograph/.
[2] Apache Giraph Project. http://giraph.apache.org.
[3] Boost Graph Library (BGL). http://www.boost.org/doc/libs/ 1_55_0/libs/graph/-doc/index.html.
[4] Cypher - the Neo4j query Language. http://www.neo4j.org/learn/cypher.
[5] InfiniteGraph. http://www.objectivity.com/infinitegraph.
[6] Java universal network/graph framework. http://jung.sourceforge.net.
[7] Neo4j graph database. http://www.neo4j.org/.
[8] NetworkX. https://networkx.github.io.
[9] Oracle Spatial and Graph, RDF Semantic Graph,. http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/rdfsemantic-graph-1902016.html.
[10] PGQL: Property Graph Query Language. http://pgql-lang.org/.
[11] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.
[12] Virtuoso Universal Server. http://virtuoso.openlinksw.com/.
[13] Bizer, C., and Schultz, A. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.* (2009).
[14] Dave, A., Jindal, A., Li, L. E., Xin, R., Gonzalez, J., and Zaharia, M. GraphFrames: An Integrated API for Mixing Graph and Relational Queries. In *GRADES* (2016).
[15] Ediger, D., McColl, R., Riedy, J., and Bader, D. A. STINGER: High Performance Data Structure for Streaming Graphs. In *HPEC* (2012).
[16] Graefe, G., and Davison, D. L. Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Execution. *IEEE Trans. Software Eng.* (1993).
[17] Gurajada, S., Seufert, S., Miliaraki, I., and Theobald, M. TriAD: A Distributed Shared-Nothing RDF Engine Based on Asynchronous Message Passing. In *SIGMOD* (2014).
[18] Hong, S., Depner, S., Manhardt, T., Lugt, J., Verstraaten, M., and Chafi, H. PGX.D: A Fast Distributed Graph Processing Engine. In *SC* (2015).
[19] Kang, U., Tsourakakis, C. E., and Faloutsos, C. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM* (2009).
[20] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB* (2012).
[21] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: A System for Large-scale Graph Processing. In *SIGMOD* (2010).
[22] Nguyen, D., Lenharth, A., and Pingali, K. A Lightweight Infrastructure for Graph Analytics. In *SOSP* (2013).
[23] Potter, A., Motik, B., Nenov, Y., and Horrocks, I. Distributed RDF Query Answering with Dynamic Data Exchange. In *ISWC* (2016).
[24] Sevenich, M., Hong, S., van Rest, O., Wu, Z., Banerjee, J., and Chafi, H. Using Domain-Specific Languages For Analytic Graph Databases. *PVLDB* (2016).
[25] Sevenich, M., Hong, S., Welc, A., and Chafi, H. Fast In-Memory Triangle Listing for Large Real-World Graphs. In *SNAKDD* (2014).
[26] Sundaram, N., Satish, N., Patwary, M. M. A., Dulloor, S., Anderson, M. J., Vadlamudi, S. G., Das, D., and Dubey, P. GraphMat: High Performance Graph Analytics Made Productive. *PVLDB* (2015).
[27] van Rest, O., Hong, S., Kim, J., Meng, X., and Chafi, H. PGQL: A Property Graph Query Language. In *GRADES* (2016).
[28] Zeng, K., Yang, J., Wang, H., Shao, B., and Wang, Z. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB* (2013).