# Distributed Asynchronous Regular Path Queries (RPQs) on Graphs (industry track)


Tomáš Faltín*
Oracle America, Inc.
United States
tomas.faltin@inria.fr

Vasileios Trigonakis
Oracle America, Inc.
United States
vasileios.trigonakis@oracle.com

Ayoub Berdai†
Oracle America, Inc.
United States
ayoub.berdai@oracle.com

Luigi Fusco
Oracle America, Inc.
United States
luigi.fusco@oracle.com

Călin Iorgulescu
Oracle America, Inc.
United States
calin.iorgulescu@oracle.com

Jinsoo Lee
Oracle America, Inc.
United States
jinsoo.lee@oracle.com

Jakub Yaghob
Charles University
Czechia
jakub.yaghob@matfyz.cuni.cz

Sungpack Hong
Oracle America, Inc.
United States
sungpack.hong@oracle.com

Hassan Chafi
Oracle America, Inc.
United States
hassan.chafi@oracle.com



## Abstract

Graph engines play a crucial role in modern data analytics pipelines, serving as a middleware for handling complex queries across various domains, such as financial fraud detection. Graph queries enable flexible exploration and analysis, akin to SQL in relational databases. Among the most expressive and powerful constructs of graph querying are regular path queries (RPQs). RPQs enable support for variable-length path patterns based on regular expressions, such as `(p1:Person)-/:Knows+/->(p2:Person)` that searches for non-empty paths of any length between two persons.

In this paper, we introduce a novel design for distributed RPQs that builds on top of distributed asynchronous pipelined traversals to enable (i) memory control of path explorations, with (ii) great performance and scalability. Through our evaluation, we show that with sixteen machines, it outperforms Neo4j by 91× on average and a relational implementation of the same queries in PostgreSQL by 230×, while maintaining low memory consumption.



## CCS Concepts

• **Information systems** → **Parallel and distributed DBMSs**; **Main memory engines**; **Database query processing**; *Graph-based database models*; *DBMS engine architectures*.

## Keywords

regular path queries, reachability, distributed graph processing, graph querying, graph databases

**ACM Reference Format:**
Tomáš Faltín, Vasileios Trigonakis, Ayoub Berdai, Luigi Fusco, Călin Iorgulescu, Jinsoo Lee, Jakub Yaghob, Sungpack Hong, and Hassan Chafi. 2023. Distributed Asynchronous Regular Path Queries (RPQs) on Graphs (industry track). In *24th International Middleware Conference Industrial Track (Middleware Industrial Track '23), December 11–15, 2023, Bologna, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626562.3626833




*Middleware Industrial Track '23, December 11–15, 2023, Bologna, Italy*



ACM ISBN 979-8-4007-0427-7/23/12.
https://doi.org/10.1145/3626562.3626833


## 1 Introduction

Graph engines play a crucial role in modern data analytics pipelines, serving as middleware to handle complex queries across various domains. Graph querying enables interactive analysis of graphs. This is typically done using one of the many query languages, such as SQL/PGQ [1], PGQL [2] or Cypher [3]. Graph queries are similar to what SQL offers to relational databases, with an additional focus on vertex/edge patterns and support for graph specific algorithms.[1]

Graph queries can express both *fixed patterns*, such as `(p1:Person)-[:Knows]->(p2:Person)`, and *variable-length patterns*, including regular path queries (RPQs). Variable-length patterns represent the true power of graphs as they are workloads that are far better suited in terms of expressiveness for graph query languages and in terms of performance on top of graph indices in graph engines. For instance, a user can very intuitively express the following RPQ query in PGQL:

```
PATH p AS (:Person)-[:Knows]-(:Person)
SELECT COUNT(*)
FROM MATCH (p1:Person) -/:p+/-> (p2:Person)
```

that counts the distinct pairs of persons connected by ≥ 1 `Knows` hops. However, expressiveness brings complexity: Variable-length patterns are a very challenging workload due to the potential explosion of intermediate explorations, especially in large graphs.

In this paper, we present `RPQd`, a novel algorithm for implementing RPQs on distributed graphs. `RPQd` deviates from the traditional breadth-first traversal (BFT) shortest-path algorithm and instead leverages asynchronous distributed traversals of PGX.D/Async [4].

*This project was completed while the author was a research assistant for Oracle Labs. The author is currently affiliated with Inria.

†This project was completed while the author was a research assistant for Oracle Labs. The author is also affiliated with Mohammed V University in Rabat, Ecole Mohammadia d'Ingénieurs, SIP Research Team.

[1]The recent publications of the SQL/PGQ ISO standard extension [1] makes this similarity even more profound

In brief, RPQd executes recursive depth-first traversal (DFT) explorations within a machine and buffers remote matches together to reduce the overheads of messaging. This enables RPQd to implement flow control for controlling the memory expansion of RPQ explorations. Unlike fixed patterns, RPQs involve explorations of variable depth, requiring RPQd to implement dynamic flow control, combined with an incremental termination detection protocol.

Additionally, RPQd implements min and max quantifiers and builds a reachability index on-the-fly to detect cycles (avoiding infinite loops) and to eliminate duplicate paths. Furthermore, the unique design could enable support for generic cross-filters between RPQ and non-RPQ patterns, such as:

```
PATH p AS (pa:Person)-[:Knows]-(pb:Person)
SELECT COUNT(*)
FROM MATCH (p1:Person) -/:p*/-> (p2:Person)
WHERE p1.age <= pa.age AND pb.age <= p2.age
```

that counts the number of persons who form a chain of people knowing each other having their age in ascending order. To the best of our knowledge, RPQd is the first distributed RPQ querying system to support such a powerful feature.

We implement and test RPQd on top of PGX.D/Async [4] with LDBC [5] graphs. Our evaluation demonstrates that with four machines, RPQd is on average more than 29× and 96× faster than Neo4j[1] [6] and PostgreSQL [7], respectively. Focusing on three original LDBC queries only, RPQd is on average 17× and 227× faster than Neo4j and PostgreSQL, respectively. Moreover, we analyze two queries in-depth and show that RPQd is able to execute with low memory footprint due to its DFT matching style. Our last experiment shows how RPQd behaves on queries of varying depths and identifies the primary bottlenecks in our algorithm.

## 2 Background and Related work

*Property Graph.* Graphs are popular for their ability to efficiently model relationships between entities. In a graph, the connection between two entities is materialized and easily traversable when executing pattern-matching queries compared to the relational model requiring joins. The property graph model represents the graph topology as vertices and edges, along with properties and labels. Properties can be associated with any vertex or edge, and take the form of typed key-value pairs. Labels are key-only and represent types or categories, e.g., person or animal.

*Graph Querying and Pattern Matching.* Several languages for graph querying exist, such as the standard SQL/PGQ [1], PGQL [2], Cypher [3], GQL [8], and SPARQL [9]. In this work, we use PGQL. Graph querying searches for generic *homomorphic* patterns with optional filters together with projecting or aggregating requested data, including arbitrary expressions from the vertices, edges, and properties. This approach differs from graph-mining systems that are typically not that generic and focus on *isomorphic* matching [10] allowing for more specialized optimizations.

Graph traversals for pattern matching traditionally follow either a breadth-first (BFT) [9], depth-first traversal (DFT) [4] execution, or a mix of the two [11].

*Reachability Queries.* Reachability queries are a fundamental graph workload and are used in a large number of applications, e.g., in financial fraud detections or bioinformatics [12]. Reachability tests if there exists a path between two vertices in a graph. PGQL supports variable-length paths for reachability queries in the form of *regular path queries* (RPQs). A path pattern declaration starts with the `PATH` keyword. Any pattern containing at least one vertex is a valid path pattern. Quantifiers allow for specifying an upper and/or lower bound on the number of repetitions of the path pattern that makes up the matched variable-length pattern.

Various approaches exist, finding different tradeoffs between the expensive building of a complete offline index in order to answer queries at runtime in $O(1)$ [13, 14], or performing a depth-first or breadth-first search, with a runtime of $O(n+m)$. Another method using two-hop indexing [15] labels vertices in $O(n^4)$ and executes queries in $O(m^{1/2})$.

*Regular Path Queries.* Regular path queries express a path between two vertices as a regular expression on the labels of the sequence of traversed edges. They were first proposed by Cruz et al. [16] and have since been extensively studied in different variants [17–21], with distributed graphs being a popular use case due to the possible semantic web applications. Label-constrained RPQs [22–25] are limited to constraints on edge labels. General path queries extend this concept to graphs with infinitely many labels [26] which is equivalent to having arbitrary filters in property graphs.

Neo4j's Cypher [3] allows variable-length paths by specifying an edge with a range of repetitions. PostgreSQL [7] supports the non-standard `RECURSIVE WITH` statement for recursive queries, returning a table constructed by `UNION` of the output rows of queries running recursively on preceding query's output.

PGQL allows reachability queries with a subset of regular path semantics, in particular, it allows repetition of path patterns. When executing a query, the runtime matches all destinations reachable from a given source by following paths that respect the pattern. The pattern consists of (repeatedly) matched vertices and edges, doing homomorphic matching, which corresponds to relational joining. This differs from the traditional concept of RPQs that focuses on unique matching of vertices and edges to the pattern (isomorphism) that results automatically in acyclicity and duplication elimination [27]. The regular language `(a)*bb(a)+` over the label alphabet `{a,b}` can be translated into PGQL using two variable-length patterns in the same query. One restriction is that the "OR" operation is limited to the single vertex or label match.

## 3 RPQd: Distributed RPQs

The main design goal of RPQd is to implement a fast, fully in-memory solution for distributed variable-length queries of any size on top of asynchronous DFT distributed queries [4] to inherit the controllable memory consumption characteristics of asynchronous traversals.

Figure 1 illustrates the design of RPQd. The query involves matching of a 2-hop pattern where the source vertex has a non-zero length path leading to the second vertex restricted with filters. Section 3.1 describes the translation of queries to execution plans, while Section 3.2 provides a description of the execution itself. We then describe important runtime aspects: (i) how to limit the execution memory in Section 3.3, (ii) how to terminate in Section 3.4, and (iii) how to remove duplicates and avoid cycles in Section 3.5.

[1] Using Neo4j Community Edition (benchmarks not audited by Neo4j).

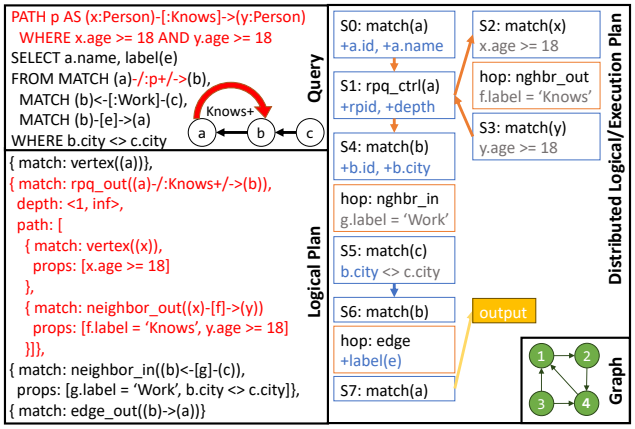


**Figure 1: Query planning, final execution plan and example graph (in green). RPQ-related parts depicted in red. Transition hops in orange, inspection hops in blue, and output in yellow.**

**Table 1: Query planning operators.**

| Stage/Hop | Operator | Example | Description |
|---|---|---|---|
| Stage | Vertex match | `(x)->(y)` | Matches vertices without following edges |
| RPQ control stage | RPQ match | `(x)-/:path*/->(y)` | Matches vertices and controls RPQ matching |
| Neighbor hop | Neighbor match | `(x)->(y)` | Matches neighbors of the current vertex |
| Edge hop | Edge match | `(x)->...` `(y)->(x)` | Matches again an already matched vertex |
| Inspection hop | - | `(x)->(y)->(z),` `(y)->(w)` | Transfers the execution back to an already matched vertex |
| Transition hop | - | - | Transfers the execution between stages |
| Output hop | (the last hop) | - | Stores the intermediate results |

## 3.1 Query Planning

Every PGQL query undergoes multiple transformations, resulting in a distributed execution plan. These transformations are depicted in Figure 1.

*PGQL Query ⇒ Logical Plan.* This step converts the submitted query into a logical plan using operators from Table 1.

Multiple equivalent plans with different performance characteristics can exist, e.g., an alternative plan for our example query could swap the last two operators. The engine employs a standard cost-based query planner that selects the best plan using the following heuristics: (i) Prefer single-match vertices, e.g., `ID(v) = 123`, as starting points. (ii) Prioritize vertices with heavy filtering in the early stages. (iii) Prefer edge matches over neighbor matches since the cost of using the edge operator is logarithmic compared to a standard neighbor match. (iv) RPQ match is preferred over neighbor match because RPQ matching is slower and needs to run earlier due to potential match explosion. In future work, we plan to incorporate the scouting queries technique [28] with RPQs to improve planning.

*Logical Plan ⇒ Distributed Query Plan.* This step adds important distributed-specific information into the plan. All operators are transformed into a finite automaton consisting of *stages* (states) and *hops* (transitions between them). The behavior of stages and hops is described in Table 1.

The *inspection hop* is a special distributed operator designed to support non-linear patterns. In the Figure 1 example, Stage 5 sends the computation back to the machine of the matched vertex `b`.

To support RPQs, a special *RPQ control stage* is added to the plan. This stage incorporates RPQ-specific logic (e.g., flow control, termination protocol) and data structures (e.g., reachability index). *Transition hops* are used to connect the RPQ path pattern with the RPQ control stage, as well as the RPQ control stage with normal stages, enabling support for 0-hop matching.

*Distributed Query Plan ⇒ Execution Plan.* The engine proceeds to materialize the execution plan based on the distributed query plan. This involves allocating data structures, (filter) expressions, and execution contexts. The execution context refers to preallocated intermediate result storage for each stage on a per-thread basis. It has a fixed layout and, for non-RPQs, a fixed length. RPQ context is preallocated up to a predetermined depth and dynamically allocated if further needed. In Figure 1, expressions are depicted in grey, while the data inserted into the context is shown in blue.

## 3.2 RPQd Execution Runtime

The engine uses an execution plan automaton for performing the graph depth-first traversals. Each worker thread on each machine is assigned a distinct local set of vertices and sequentially executes the initial stage of the automaton on those vertices (known as the bootstrapping process). If a vertex satisfies all its filters, the computation can transition (*hop*) to the following stage by matching a graph edge. If the edge is local, i.e., the destination vertex is stored locally, the next stage is recursively applied to the destination vertex. In case of a remote edge, the engine serializes the working context into a message and continues computation with another edge. If there are no more edges to process from a matched vertex, RPQd *backtracks* one stage and continues with the subsequent neighbor. Eventually, the engine returns back to the initial stage after processing the entire graph sub-tree.

Back to the Figure 1 example: For simplicity, we will focus on the execution of a single worker on a single machine, assuming all properties fulfill the query filters. We refer to `Stage i` as `Si`, and to graph `vertex i` as ①.

The worker starts matching by applying ① on `S0`. It collects the properties `a.name` (projection) and `a.id=v1` (edge hop), *transitions* to `S1`, creates `rpid` (see Section 3.5) and stores `depth=0` into the context.

The `depth` controls the number of RPQ iterations. It is incremented on each matching iteration of the control stage. If `depth` < `min_hop`, the RPQ path matching continues. If `min_hop` ≤ `depth` ≤ `max_hop`, (i) it atomically checks and updates the *reachability index* for duplications, then (ii) it *transitions* to non-RPQ stage moving the DFT matching closer to the output, and also (iii) *transitions* to RPQ path stages for large depths. If `depth` > `max_hop`, it declines the match and backtracks.

In our example, `S1` checks the depth limit (`0` < `min_hop=1`) and transitions to `S2` (① match). Afterwards, the worker takes the first output neighbor edge to ②, evaluates edge filters, and hops on `S3` (② match) and transitions to `S1`. `S1` increments `depth=1`, and successfully checks the limit (`depth≥1`). It checks the reachability

index, creates a new entry for path {①,②}, and transitions to S4 (② match). It tries to continue using incoming neighbors (none), so it backtracks back to S1 and transitions again to S2 (② match), hops to S3 (④ match), and returns back to S1 which increments depth=2. After that the worker checks and creates an entry for {①,④}.

It transitions to S4 (④ match) and stores b.id=v4 (inspection hop) and b.city (filters), then it hops to: (i) S5 (③ match: evaluate filters using b.city), (ii) S6 (④ match: inspection hop back using b.id=v4), (iii) S7 (① match: edge match using a.id=v1), and finally (iv) output (storing the projections from context into the output). After that, the worker backtracks and continues with further matching. Note that it cannot loop to ② because reachability index contains the {①,②} entry.

*Messaging.* For messaging, RPQd uses an internal network library that allows efficient message passing and handles any faults. RPQd batches multiple contexts for the same machine and stage into a single message, which is then sent asynchronously once it is full. Upon receiving a message, a dedicated receiver thread places it into the message queue of the corresponding stage.

The received messages are picked up eagerly, prioritizing the latest stages and depths, and processed by the thread: (i) before bootstrapping new work from another vertex, (ii) after processing all the edges and backtracking, and (iii) when flow control prevents message sending. The RPQ control stage handles incoming messaging for path stages using a priority queue rather than the per stage queues of normal stages. Incoming messages are processed based on the larger depth first and, the later path stage first for the same depth.

## 3.3 Flow Control

In RPQd, memory consumption for pattern matching is effectively managed during depth-first traversal. The engine employs a local DFT for fixed pattern matching, ensuring constant memory usage. Messaging between machines utilizes a fixed number of preallocated buffers, partitioned equally among stages and machines. Each machine requires at least two buffers (for sending and receiving).

Flow control is employed to regulate buffer allocation, limiting the number of buffers sent to a destination machine. Once a buffer is processed, a special DONE message is sent back to the source machine, indicating that it was released and made available again.

For RPQ stages, RPQd equally partitions buffers among stages, machines, and up to the preconfigured depth D. Remaining buffers are partitioned across the path stages and are shared within a single path stage for all the depths $\geq D$. Additional overflow buffers are included to prevent livelocks that occur when a path stage is blocked at depth D, but buffers become available only after matching at a depth larger than D. The total number of consumed buffers for RPQs is $O(\#\text{machines} * (P * D + P) + \#\text{overflow\_buffs})$, where $P$ = #path_stages. In other words, unlike fixed-patterns, with RPQ, we cannot fully control memory consumption due to the need for overflow buffers. Still, as we detail in Section 4, the memory for few per-depth overflow buffers is negligible.

## 3.4 Termination Protocol

The engine executes the pattern matching until it visits all potential graph sub-trees. RPQd extends the lightweight distributed protocol of PGX.D/Async [4] to check termination conditions incrementally in stages 0 through N-1. The conditions for stage i are: (i) all the local work was finished, (ii) all the remote work was received and finished and (iii) the previous stage (i-1) terminated globally.

To support RPQs, we extend the protocol with additional conditions that check all the RPQ stages incrementally in depths from 0 through max_depth: (i) RPQ stage i terminates if it terminates on every depth d $\leq$ max_depth. (ii) RPQ stage i terminates on depth d if (a) all the path stages on depth $<$ d have terminated, and (b) all the path stages preceding stage i on the same depth d have terminated.

*Unbounded RPQs.* To address unbounded RPQs, RPQd proposes a consensus-like protocol for determining the *maximum observed depth*. Each machine independently tracks its maximum observed depth locally. When a machine M terminates stages S at depth D, it includes its maximum locally observed depth in the termination message broadcasted to other machines. By analyzing these termination messages, the engine deduces that there are no more remote messages from a specific machine, stage, and depth, indicating that the machine cannot extend the maximum observed depth further. Once all machines broadcast the same maximum observed depth D within the termination message for depth D, RPQd reaches a consensus on the maximum observed depth.

## 3.5 Reachability Index

The semantic of (homomorphic) reachability queries requires that, given a source, each destination is accounted only once, e.g., following query (a)->(b)-/:p+/->(c) with a graph {②->①<-③, ①->④->⑨, ①->⑤->⑨, ①->⑥->⑨} has only 2 matched results: {{②,①,⑨},{③,①,⑨}}.

Therefore, any alternative paths must be either avoided or eliminated. Additionally, any path exploration needs to ensure that cycles are not infinitely followed. To solve these issues, RPQd dynamically builds a *reachability index*.

The reachability index is a distributed data structure supporting atomic inserts and updates that behaves as a map with (i) keys being reachability path IDs (rpid) and (ii) values being the path depths.

*Path Encoding.* RPQd employs a specific encoding for source path ID: (machineId,workerId,seqId), where (machineId, workerId) (2 × 8-bits) is a unique worker identifier and seqId (48-bits) is a thread local sequence ID of the source matched path. RPQd uses the fact that every single path in DFT-based engine is processed by a single thread before entering the RPQ stage. As for the destination path ID, a simple vertex ID (64-bits) is used. rpid is constructed by combining source and destination path IDs (2 × 64-bits).

*Implementation.* RPQd implements the reachability index as a two-level map. The first level map is indexed by the destination vertex ID. Due to the continuous range of vertex IDs, it is implemented using an array of atomic pointers to the second level map. The second level map is a parallel map that stores the source path ID and the path's depth.

The reachability index is partitioned based on the vertex destination ID, i.e., the entry is stored on the path destination machine. This means that the index entries cannot be used for speeding up traversals by avoiding duplicated paths. This is left for a future work.

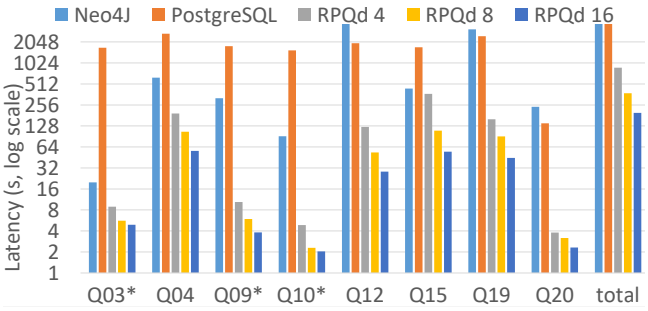


**Figure 2: Different configurations of RPQd vs. other systems.**

# 4 Evaluation

In this Section, we evaluate the performance of RPQd and compare to state-of-the-art graph and relational databases. We also analyze the behavior of RPQd with artificial queries.

## 4.1 Experimental Settings

*Methodology.* We run each query 10 times and report the median latency. Each experiment executes the queries in round-robin fashion in order to eliminate caching effects. We use 4 to 16 machines for running RPQd and a single machine for Neo4j and PostgreSQL.

*Hardware.* We use a cluster of 16 machines, each with two Intel Xeon CPU E5-2699 v3 2.30GHz CPUs with 18 cores (hyperthreads disabled/DVFS enabled), for 36 cores in total. Each machine contains 384GB of DDR4-2400 memory and LSI MegaRAID SAS-33108 storage. Each machine includes a Mellanox Connect-X InfiniBand card, all connected to an EDR 100Gbit/s InfiniBand network.

*Implementation and Configuration.* We implement RPQd on top of the PGX.D/Async engine [4]. We configure it to have 36 worker threads consisting of two dedicated to messaging. We use 8192 message buffers of 256KB per machine for flow control. This setting translates to approximately 2GB of intermediate results that can be produced by a single machine. This implies that the maximum worst-case memory consumption for messaging is $N * 2GB$, where $N$ is number of machines. For RPQs, flow control works with preallocated buffers (out of the 8192 allowance) up to depth four. For larger depths, it allows five shared messages per path stage with one extra overflow message per each depth. Contexts are preallocated up to depth three.

*Neo4j.* Neo4j [6] is a single machine graph disk-based database, but uses an in-memory cache for performance. We ignore the first slow run that brings data from disk. We run Neo4j Community Edition `4.0.10`, because it has the best performance with the same configuration compared to newer versions. To support all LDBC queries, we use the APOC [29] library, version `4.0.0.16`. Neo4j uses is a single cluster machine configured according to the Neo4j-admin Memrec [30] utility: 31GB of heap initial and max size, 320GB pagecache and 36 worker threads.

*PostgreSQL.* PostgreSQL [7] is a single-machine open-source relational database. We run version `15.2`. The database uses a single cluster machine configured with PGTune [31]: 99GB of shared buffers, 279GB effective cache size and 36 parallel workers.

*Graphs and Queries.* We use the latest LDBC graphs [5]: SF10 (27 million vertices, 170 million edges) and SF100 (255 million vertices, 1.7 billion edges). As a workload for comparison against the other engines, we take RPQs from the LDBC Business Intelligence (BI) queries [32] and adjust them to run on all engines – Neo4j supports reachability queries directly, PostgreSQL uses recursive queries for implementing reachability. In total, we use nine queries: Three queries are the original BI (Q3, Q9, Q10) and the remaining are adaptations of the original ones. In the adapted queries, we remove expressions unsupported by RPQd, such as correlated subqueries, and use the part related to reachability matching only. Note that this is not a standard implementation of the LDBC benchmark.

## 4.2 RPQd vs. Other Systems

Figure 2 includes the results for RPQd, Neo4j, and PostgreSQL on LDBC SF100. The queries marked with "$\star$" are the unmodified BI. In terms of total time (i.e., sum of all queries), RPQd with four machines is more than 18× and 16× on average faster than Neo4j and PostgreSQL, respectively.

RPQd performs the best on queries that explore tree-subgraphs of LDBC (all except Q10), e.g., with `Reply` labels. In these queries, a breadth-first approach brings no advantages compared to RPQd, which in turn is able to greatly contain memory usage.

RPQd delivers the worst scalability on Q03*, because of an intermediate-result explosion on depth one, which leads to flow control blocking the execution more than 82 million times, approximately 5× more than the number of matched vertices at that stage. Still, with the evaluated settings, RPQd delivers the purpose of consuming approximately 2GB per machine for pattern matching (excluding the reachability index) and better performance than the other engines.

## 4.3 Scalability

Figure 2 shows that RPQd scales very well overall. Comparing to the default configuration with four machines, using eight and 16 machines is 2.3× and 4.4× faster, respectively, meaning that the speedup is almost linear. Super linear speedups happen due to flow control: Every machine is configured with approximately 2GB of memory for pattern matching, thus, eight or 16 machines have more memory for computations. The limited scalability to 16 machines (Q3, Q10) happens because of (i) partitioning and narrow starting queries (Q3 filters `country.name='Burma'`, thus starting from a single vertex), and (ii) less local computations that can lead to excessive flow control for some queries (to the point that it cannot be compensated by the higher flow control allowance).

## 4.4 Deep Dive of Q9 and Q10 on LDBC SF100

We analyze the detailed statistics of two queries: Q9 and Q10, exhibiting fundamentally different behaviors.

Q9 uses reachability in order to find recursively all replies to messages. The query starts from a large number of messages (see Table 2) and traverses their comment trees, resulting first in an explosion of results (a message can have multiple answers), followed by an exponential decrease (few `Reply` chains are long). Due to nature of its filters and the graph, the reachability part of this query is always performed on a tree, making the reachability index superfluous. This query without reachability index on eight machines executes 3.4× faster than with the index.

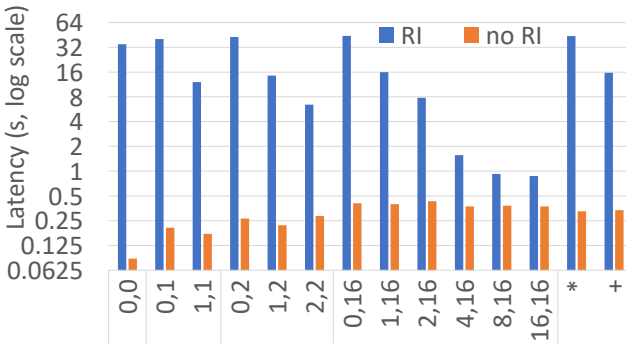


**Figure 3: Latency of `Reply` RPQs with different depths with and without reachability index on LDBC SF10.**

Q10, starting from a predefined single person, finds all persons in a within two or three `Knows` hops. The reachability index is heavily used, allowing traversal back and forth along `Knows` edges. Table 3 shows the number of visited vertices with number of eliminated (vertex already reached at a lower or equal depth before) and duplicated (the vertex already reached at a greater depth) vertices in reachability index. The duplication can occur due to prioritizing depth-first work: It can match at higher depths while some shallower computations are still pending. This essentially represents the main limitation of a DFT-oriented engine compared to BFT. The large number of depth three eliminations is due to that most vertices visited at depth three have more than one neighbor matched already at depth two. The elimination count would be the same in a BFT engine. Since RPQd favors deeper work first, many final result materializations happen before the RPQ search at depth two is complete. This property allows the engine to have a very low runtime memory usage.

None of these two queries triggers flow control: With eight machines the engine stayed below a total of 16GB memory. The number of elements saved in the reachability index is equal to the sum of the matched vertices in the RPQ control stage minus the eliminations and the duplications. Each entry in the reachability index occupies 12 bytes, resulting in a total dynamic size of 181MB for Q09 and 4.4MB for Q10 (compared to 100GB worth of data in LDBC SF100).

## 4.5 Effects of Reachability Index

Figure 3 shows the performance of simple artificial queries with and without the reachability index, highlighting the dynamic allocation of the index. We test a `Reply` pattern (worst-case for the index) while controlling the min and max exploration depth. Hops $\{0, 0\}$ represent 0-hop meaning RPQd inserts entries $\{v, v\}$ for each graph vertex `v`. This shows the overhead of dynamically allocated index. All patterns with 0-min hop includes this allocation overhead. Increasing inserts and updates into the index, by increasing the max-hop, has a negligible effect. By pre/bulk-allocating the index can trade memory for performance (left for future work).

**Table 2: Relevant statistics of the RPQ control stage of Q9.**

| depth | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| #matches | 3.1M | 5.9M | 4M | 1.5M | 375k | 62k | 7k | 658 | 52 | 1 | 0 |

**Table 3: Relevant statistics of the RPQ control stage of Q10.**

| depth | num. matches | eliminated | duplicated |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 1 | 35 | 0 | 0 |
| 2 | 19978 | 4036 | 12969 |
| 3 | 2700017 | 2334441 | 0 |

Increasing the min-hop with reachability index, leads to a somehow counter-intuitive performance improvement. Larger min-hop values reduce the number of reachability entries created, because any traversal with depth below min hop creates no entry. In future work, we plan to tune better the memory/performance trade-off of reachability indices.

# 5 Concluding Remarks

We presented RPQd, a regular path query (RPQ) algorithm to answer reachability queries on top of distributed asynchronous depth-first graph traversals. RPQd supports both bounded and unbounded repetition of arbitrary path patterns, including regular expressions over any labels, as well as advanced features like local and cross filtering. It seamlessly integrates with asynchronous depth-first traversals, extending its structure based on stages, and messaging with specialized RPQ stages and contexts. The runtime achieves strong guarantees on memory usage and outperforms state-of-the-art solutions.

*Limitations and Future Work.* RPQd builds upon the DFT engine, which implies aforementioned strengths, but also imposes limitations on the RPQ engine itself. Our approach excels in tree topology graphs and real-world workloads like social networks. However, as demonstrated in Section 4.5, when a graph-query combination generates numerous duplicated reachability paths, e.g., searching for long paths in complete graphs, the DFT algorithm reaches its limit. In such cases, more specialized algorithms like BFT, might be a better fit if sacrificing low memory consumption for a faster evaluation is acceptable. Alternatively, when provided with a generated reachability graph, RPQd can run a fast RPQ pattern matching without compromising performance and memory consumption. Additionally, RPQd can leverage the aDFS [11] improvements for better dynamic parallelization.

# References


[1] Property Graph Queries (SQL/PGQ). https://www.iso.org/standard/79473.html, June 2023.

[2] PGQL 1.5 Specification – Property Graph Query Language. https://pgql-lang.org/spec/1.5/.

[3] Neo4j Cypher Query Language – Developer guides. https://neo4j.com/developer/cypher/.

[4] Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine. *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, 2017.

[5] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter A. Boncz, Vlad Haprian, and János Benjamin Antal. An Early Look at The LDBC Social Network Benchmark's Business Intelligence Workload. In *GRADES Workshop*, 2018.

[6] Neo4j – Graph database platform. https://neo4j.com.

[7] PostgreSQL – The world's most advanced open source database. https://www.postgresql.org.

[8] GQL Standard – Graph Query Language. https://www.gqlstandards.org.

[9] SPARQL Query Language for RDF – SPARQL Protocol and RDF Query Language. https://www.w3.org/TR/rdf-sparql-query/.

[10] Brian Gallagher. Matching Structure and Semantics: A Survey on Graph-Based Pattern Matching. In *AAAI Fall Symposium*, 2006.

[11] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. aDFS: An almost Depth-First-Search distributed Graph-Querying system. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 209–224. USENIX Association, July 2021.

[12] Jacques Van Helden, Avi Naim, Renato Mancuso, Mattew Eldridge, Lorenz Wernisch, David Gilbert, and Shoshana J Wodak. Representing and analysing molecular and cellular function in the computer. *Biological chemistry*, 381(9-10):921–935, 2000.

[13] Klaus Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science*, 58(1-3):325–346, 1988.

[14] Haixun Wang, Hao He, Jun Yang, Philip S Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 75–75. IEEE, 2006.

[15] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[16] Isabel F Cruz, Alberto O Mendelzon, and Peter T Wood. A graphical query language supporting recursion. *ACM SIGMOD Record*, 16(3):323–330, 1987.

[17] Pablo Barceló Baeza. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 175–188, 2013.

[18] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):1–40, 2017.

[19] George HL Fletcher, Jeroen Peters, and Alexandra Poulovassilis. Efficient regular path query evaluation using path indexes. 2016.

[20] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. Rewriting of regular expressions and regular path queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 194–204, 1999.

[21] Wim Martens and Tina Trautner. Evaluation and enumeration problems for regular path queries. In *21st International Conference on Database Theory (ICDT 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[22] Lei Zou, Kun Xu, Jeffrey Xu Yu, Lei Chen, Yanghua Xiao, and Dongyan Zhao. Efficient processing of label-constraint reachability queries in large graphs. *Information Systems*, 40:47–66, 2014.

[23] Lucien DJ Valstar, George HL Fletcher, and Yuichi Yoshida. Landmark indexing for evaluation of label-constrained reachability queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 345–358, 2017.

[24] Xin Chen, You Peng, Sibo Wang, and Jeffrey Xu Yu. DLCR: Efficient Indexing for Label-Constrained Reachability Queries on Large Dynamic Graphs. *Proc. VLDB Endow.*, 15(8):1645–1657, 2022.

[25] You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Answering Billion-Scale Label-Constrained Reachability Queries within Microsecond. *Proc. VLDB Endow.*, 13(6):812–825, 2020.

[26] Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 122–133, 1997.

[27] Angela Bonifati, George Fletcher, Hannes Voigt, Nikolay Yakovets, and HV Jagadish. *Querying graphs*, volume 10. Springer, 2018.

[28] Tomáš Faltín, Vasileios Trigonakis, Ayoub Berdai, Luigi Fusco, Călin Iorgulescu, Sungpack Hong, and Hassan Chafi. Better Distributed Graph Query Planning With Scouting Queries. In *Proceedings of the 6th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 1–9, 2023.

[29] APOC – Awesome Procedures On Cypher. https://neo4j.com/labs/apoc/.

[30] Neo4j Admin Memrec – Neo4j memory recommendations. https://neo4j.com/docs/operations-manual/current/tools/neo4j-admin/neo4j-admin-memrec/#_example.

[31] PGTune – PostgreSQL configurator. https://pgtune.leopard.in.ua.

[32] LDBC SNB BI workload. https://github.com/ldbc/ldbc_snb_bi.