Better Distributed Graph Query Planning With Scouting Queries

Tomáš Faltín*
Oracle America, Inc.
United States
tomas.faltin@matfyz.cuni.cz

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

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

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

Abstract

Query planning is essential for graph query execution performance. In distributed graph processing, data partitioning and messaging significantly influence performance. However, these aspects are difficult to model analytically, which makes query planning especially challenging. This paper introduces *scouting queries*, a lightweight mechanism to gather runtime information about different query plans, which can then be used to choose the "best" plan. In a pipelined, depth-first-oriented graph processing engine, scouting queries typically execute for a brief amount of time with negligible overhead. Partial results can be reused to avoid redundant work. We evaluate scouting queries and show that they bring speedups of up to 8.7× for heavy queries, while adding low overhead for those queries that do not benefit.

CCS Concepts

• **Information systems** → **Query planning**; *Graph-based database models*; *Parallel and distributed DBMSs*.

Keywords

query planning, distributed query planning, graph databases

ACM Reference Format:
Tomáš Faltín, Vasileios Trigonakis, Ayoub Berdai, Luigi Fusco, Călin Iorgulescu, Sungpack Hong, and Hassan Chafi. 2023. Better Distributed Graph Query Planning With Scouting Queries. In *Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES & NDA '23), June 18, 2023, Seattle, WA, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3594778.3594884

GRADES & NDA '23, June 18, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0201-3/23/06…$15.00
https://doi.org/10.1145/3594778.3594884

1 Introduction

Graph queries enable the interactive exploration of graphs, similar to what SQL offers for relational databases. Graph queries focus on edges, i.e., the connections between vertices, enabling users to submit queries with any pattern, filter, or projection. For example, the following simple SQL/PGQ [22] query:

```
SELECT
  p_brand, p_type, p_size,
  COUNT(DISTINCT supplier) AS supplier_cnt
FROM GRAPH_TABLE (tpch
  MATCH (p IS part)-[ps IS partsupp]->(s IS supplier)
  WHERE
    p.p_brand <> 'Brand#45'
    AND p.p_type NOT LIKE 'MEDIUM POLISHED%'
    AND p.p_size IN (49, 14, 23, 45, 19, 3, 36, 9)
    AND s.s_comment NOT LIKE '%Customer%Complaints%'
  COLUMNS(p.p_brand AS p_brand, p.p_type AS p_type,
    p.p_size AS p_size, s.s_suppkey AS supplier)
)
GROUP BY p_brand, p_type, p_size
ORDER BY supplier_cnt DESC, p_brand, p_type, p_size
```

counts the number of suppliers per part brand, type, and size with some rather complex constraints. Graph queries are a highly challenging workload. The number of edges traversed by a query can easily cause a combinatorial explosion of the intermediate and final results. Therefore, efficient query planning is key to improving graph query performance. The query plan dictates the order of pattern matching operations, i.e., which vertex or edge is to be matched first, second, and so on. An initial suboptimal decision by the graph query planner can negatively impact the entire query execution. Queries over large distributed graphs in particular can suffer from significantly worse performance. In our simple query example above (which represents TPC-H [7] query 16, see Section 4 for more details), matching from the `PART` first results in almost 4× worse performance than starting from `SUPPLIER`.

*This project was completed while the author was a research assistant for Oracle Labs. Current affiliation of the author is Charles University.
†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.

However, computing a performant query plan is notoriously difficult. The problem is exacerbated in the case of distributed query engines due to the need to additionally account for data partitioning, for messaging and communication costs. Properly modeling partitioning and the cost of networking for query planning is fairly complex (if not outright impossible).

The classic techniques for query planning look similar to the ones used in classic relational databases and primarily use data statistics in order to compute the potential cardinality of each of the matches. Computing the cardinality depends highly on the query; typically, the more complex the query (e.g., long patterns and extensive filtering), the harder it is for the query planner to produce a good estimate.

In this paper, we introduce *scouting queries* as a pragmatic solution to improve query planning and enhance the overall performance of distributed graph pattern matching. Scouting queries are short exploratory executions of the actual query used to benchmark the performance of different query plans in order to find the best-performing plan. Scouting queries follow these steps:

(1) Take the top N plans with traditional query planning.
(2) Execute these N plans with a short timeout (e.g., 50ms) and record statistics of their execution. If the system detects that the plan is close to completion, it simply allows this plan to run to completion.
(3) Combine the per-plan cardinality metric of the traditional query planner with the scouting query statistics and choose the best candidate plan.
(4) Execute the selected plan and, if there are opportunities to reuse the work of scouting queries, merge the outputs.

Scouting queries are better suited for large graphs and queries, as are typical for distributed graph engines, with any potential overheads amortized by the gains of the improved query plan. Additionally, scouting queries best fit engines with pipelined execution of pattern matching, i.e., engines that eagerly push intermediate results out as final. On top of such engines, the scouting query metrics include the actual final-result matching throughput, which gives a very good indication of the actual performance of the engine. However, as we detail in Section 3, scouting queries can be applied to any graph-processing engine.

We prototype scouting queries on top of the PGX.D/Async engine [34] which uses distributed depth-first-oriented matching, eagerly pushing intermediate matches out as final. We evaluate our prototype on several LDBC-inspired queries [39] and 12 actual TPC-H queries expressed as graph queries. We find that the total workload execution time improves by 3.3× and 1.7× for LDBC and TPC-H, respectively, with a maximum speedup of 8.7× on a TPC-H query and only two queries where scouting selects a worse query plan than the default planner.

The rest of this paper is organized as follows: Section 2 discusses the background and related work. Sections 3 and 4 present the design and implementation/evaluation of scouting queries. Section 5 concludes this paper and highlights possible future work.

2 Background and Related Work

Property Graph. In a graph, the link between two entities is materialized and easily traversable when executing pattern-matching queries. This gives a solid advantage to graph-oriented workloads compared to the classic relational model, which relies on joins to express links between entities. Our solution is based on the property graph model (but could be easily ported to RDF [5]), which represents the graph topology as vertices and edges, and stores properties and labels separately. Properties can be associated to 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. Separating the topology from properties avoids the proliferation of edges and allows for quick traversals of the graph.

Graph Querying and Pattern Matching. Several graph query languages exist, such as PGQL [4], SPARQL [6], Gremlin [1], and Cypher [3]. In its simplest form, graph querying makes it possible to find patterns in graphs, with filters and projections. In this work, we use SQL/PGQ [22], an extension of SQL for graph queries. Projection and aggregation operations are the same as in SQL, including GROUP BY and ORDER BY, but SQL/PGQ adds support for graph patterns and vertex and edge labels. It matches homomorphic patterns and it projects or aggregates the requested data – including arbitrary expressions – out of the matched vertices and edges.

Graph pattern-matching traversals traditionally follow a breadth-first (BFT) or depth-first traversal (DFT) execution, or nowadays a mix of the two [41]. However, new database research trends point to adopting multi-way joins, termed worst-case optimal joins, as a way to avoid intermediate result explosion and to support graph pattern matching in relational data systems [25]. Additionally, recent research in graph mining [19, 20, 23, 26, 32, 40, 43] employs various techniques such as subgraph-centric or edge-centric matching to perform graph pattern matching.

2.1 Query Planning

Query planning is a fundamental step of all declarative query languages. Since the user only describes the computation logic, it is the duty of the data management engine to come up with an execution sequence that returns the requested data. These engines have a limited set of basic operations that can be composed together and executed one after the other. The process of coming up with the right sequence of operations is called query planning, where a query plan is a specific scheduling of operations.

Table 1 describes the set of basic match operators that any (vertex-centric) distributed query engine needs to support for query planning in one way or another. The performance of any graph engine is highly influenced by the query-plan selection. Query plans dictate the pattern-matching order, i.e., which vertex or edge is matched first, second, and so on, regardless of the engine's execution model.

Every query which involves more than a single vertex match has multiple query plans. For example, without inspection matches, a simple pattern of two directed hops – `MATCH (a)->(b)->(c)` – could be planned as `(a)->(b)->(c)` or as `(c)<-(b)<-(a)`. Adding inspection matches exponentially increases the number of possible plans introducing `(b)->(c)=>(b)<-(a)` and `(b)<-(a)=>(b)->(c)`.*

For this reason, traditional query-planner methods come up with different plans for the same query, and pick the best one according to a cost metric. The cost of a query plan QP ($cost(QP) = X$) represents

*`->`, `<-` and `=>` are the query planner operators, i.e., outgoing and incoming edge match, and inspection match, respectively.

the predicted computational "costs" of the query in arbitrary units of computations. The cost can be computed based on a number of factors. One of the most important in graph querying is the cardinality of the individual matches. The cardinality of a match is an estimation of the number of data points it needs to process.

Of course, different query planners can quantify different aspects of the query plan. Different operators can require different amount of work, and thus have different costs. Filters can be taken into consideration in order to reduce the selectivity of operations based either on fixed heuristics or static and runtime statistics. Choosing the optimal query plan based on cost-based analysis is a well-explored topic and is outside of the scope of this paper. Our solution, namely scouting queries, builds directly on top of the traditional query planning approaches.

2.2 Related Work

With data sets and queries becoming increasingly complex, a traditional static cost-based query optimization as introduced by Selinger et al. [35] can become inefficient. The statistics and assumptions on which this dynamic programming approach relies can be inaccurate or sometimes even invalid. The limitations of the optimize-then-execute paradigm have led to a plethora of new approaches that rely on runtime feedback to correct the query plan [21]. This class of techniques is called adaptive query processing.

Runtime statistics refinement [8, 14–18, 36, 37] is a technique where statistics collection is triggered during query execution, resulting in little to no overhead. The newly collected statistics can be used for current or future query execution. Proactive re-optimization is another technique where the query optimizer is invoked when estimation violations occur [11, 24, 27, 28, 31]. Further improvements such as leveraging intermediate results or strategically delaying re-optimization have also been proposed [24, 28, 45]. Multi-plan choices have the query optimizer concurrently run multiple query plans, sometimes on different data subsets [9, 10, 12]. This approach is the most closely related to our scouting queries, but our solution is applied on distributed graph query execution with emphasis on the engine's throughput within a specified time frame, rather than on a restricted subset(s) of the data. To the best of our knowledge, this particular emphasis has not been previously documented in the literature. Approaches such as Smooth Scan [13] avoid sensitivity to the quality of the statistics and estimations.

A hot topic in database research is embedding machine-learning models in the query optimizer to improve its efficiency. These vary from using supervised learning on previous execution plans to generating plans for future queries [33], to training recommender systems on textual similarities between SQL queries, with the assumption that textually similar queries should have similar query plans [44]. Using latency to reward a reinforcement learning model in the query optimizer [30] has also been proposed. Adjusting inaccurate statistics by learning from the query planner's past mistakes [37] complements those works. Somehow similar to scouting queries, Trummer et al. [42] propose learning about the best join order while training on slices of data until the best order is found. Other approaches consist of enhancing instead of substituting the query optimizer, like BAO [29], which learns the best execution plans for past queries and chooses from multiple query plans suggested by a traditional query optimizer. While these approaches suffer from limitations, they herald even greater improvement possibilities [33]. Scouting queries is a pragmatic approach to offer low-overhead optimized distributed graph query plans.

Table 1: Typical graph operators used in a distributed graph pattern matching engine.

Operator	SQL/PGQ	Description
vertex match	`(x)`	Matches a vertex
outgoing edge match	`(x)->(y)`	Matches an outgoing edge from the current vertex (`x`)
incoming edge match	`(x)<-(y)`	Matches an incoming edge to the current vertex (`x`)
inspection match	`(x)->...` `(y)->(x)`	Transports the computation to an already matched vertex (`x`)

3 Scouting Queries

Scouting queries aim to augment traditional query planning, solving its shortcomings by gathering information about the actual final runtime of the pattern matching part of graph queries. They do so by executing various query plans of the same query for short duration and collecting statistics to help decide which plan will be the best once fully deployed.

3.1 Query Planning to Scouting Query Execution

In what follows, we detail the steps to integrate scouting queries in a (distributed) graph engine.

Preselect top N plans. We can use any traditional planner to preselect the top N query plans. Most algorithms assign an explicit confidence value indicating how likely a query plan is to be the optimal one. This confidence is internally used for ranking the N query plans at the beginning, but can also be used at a latter step to combine the power of traditional query planning with scouting queries.

Run scouting queries. We create a scouting query for each of the N query plans. Scouting queries are short executions of a query, which run for a limited time (in the order of milliseconds) and collect runtime information about their execution. Note that graph queries typically include pattern matching followed by post processing, such as GROUP BY and ORDER BY. Scouting queries execute only the pattern matching part of the query. We use these executions to choose the best performing query plan – out of N – based on the collected data. In order to collect statistics along the whole matching pattern, it is important for scouting queries to execute on top of a depth-first-oriented engine (DFT), i.e., an engine that eagerly pipelines intermediate results out as final. Breadth-first-oriented (BFT) engines (e.g, Neo4j [2]) collect all intermediate results per each match and then proceed to subsequent matches.

On the one hand, scouting must run almost instantly compared to the actual execution of the query in order to not add high overhead. On the other hand, scouting should have enough time to explore the complete query pattern. One should ideally configure the time limit for each scouting query based on the expected performance of the graph engine. For example, if the engine is capable of delivering throughput in the million matches per second, running the scouting

query for a few milliseconds is enough for almost any query. One must also account for the overhead of starting the query. Altogether, scouting is best-suited for large graphs and/or large queries, where the benefits can easily outweigh the overhead.

Scouting query statistics policies. The best scenario is for scouting to find some final (output) results of pattern matching. The output throughput can be the main indication of how good the query plan of the scouting query is. Our general assumption is that the throughput of a limited execution of a query plan is roughly the same as that of the entire execution (the experimental results in Section 4 validate this assumption). This means that if the query plan QP1 returns more results than query plan QP2 in the same amount of time, we expect query plan QP1 to be better than plan QP2. Our assumption could fail in theory, as the engine could be lucky while executing a worse query plan. Imagine a query `SELECT COUNT(*) MATCH (a)->(b) WHERE ID(a) < 10` where the best query plan starts from matching `(a)` because of the filtering (assuming all IDs are > 0) and continues to the matching of `(b)`. In some extreme cases, the bad plan starting matching from `(b)` can have higher or equal throughput during scouting, e.g., it matches `(b)` and follows to `(a)` with $ID(a) = 1...5$.

In order to alleviate this problem, we can mix the scouting findings with the confidence values of the default query planner. The query planner returns the confidence for each of the *N* preselected query plans. The formula for picking the best query plan can be as follows: $argmax(confidence(QP) * (throughput(QP) + 1))$. Of course, depending on the engine and the traditional query planner, one can devise different policies for weighing the query-plan selection. (In our evaluation, Section 4, we use for instance only the scouting query performance as the selection metric.)

Of course, there are queries that could have a small number or no results, which means throughput 0 for all scouting queries. To solve that issue, the engine uses a secondary metric (which requires deeper integration to the execution engine). The engine records the number of visited vertices for each vertex match and combines those as a metric. Matches at latter parts of the query mean that results flow faster towards the output, i.e., the matching happens faster.

Additionally, depending on the engine and the optimization criterion, any other statistics/metrics can be deployed. The aforementioned metric optimizes for the engine performance. One can easily create a metric that aims for a different optimization criterion, e.g., when the engine is low on memory, it can prefer memory consumption over performance. In that case, while executing a scouting query, it can monitor the memory consumption and pick the query plan with the lowest consumption. Another policy, important in cloud environments, is the one for minimizing the overall engine cost. To minimize the cost for query execution, the engine can monitor the usage of different modules (at their price) during scouting execution and pick the query plan with the potentially lowest price for the user.

Execution of scouting queries. Every scouting query executes starting from a set of vertices. If the graph vertex is fixed by the query, e.g., `(a)->(b)->(c) WHERE id(a) = 0`, we use the filter for bootstrapping the computation – for the plans that start with `(a)`. If there are multiple options for the starting vertex, we select the starting vertices randomly. By choosing the vertices randomly, the collected statistics are more representative compared to incremental vertex selection, e.g., in sorted order based on the vertex ID.

The actual execution of the scouting is up to the engine. As we mentioned above, in order for the queries to be short, the execution must be somehow limited. One way is to limit the execution time of the queries by the engine. To support this the engine needs an efficient support for execution cancellation. Using cancellation, the engine runs each of the scouting queries for the given amount of time and then the query is stopped.

Another approach at obtaining meaningful results in a short time is to limit the parts of the graph that are traversed. Instead of traversals navigating all edges, the engine randomly chooses edges followed for pattern matching at each step. In this case, the scouting queries are performing a random walk with the given query on the searched graph. This can be implemented by adding a random filter on every element of the scouting query that returns false with a certain probability, thus pruning further exploration of some paths. This approach is not as clean or effective as the time-capped method of the previous paragraph, but can be used to deploy scouting queries in BFT-based engines, which have no control of pushing output results out eagerly.

Furthermore, while monitoring the number of matches for each vertex match, the engine keeps track on whether the scouting query already traversed significant parts of the graph. In that case, it gives up on the execution of other scouting queries and lets the engine execute the current plan. To minimize the potential of running a worse query plan, we execute scouting queries in order of the default query-planner confidence. We predict when a query plan should continue by estimating the amount of remaining time after a scouting query. For example, if the time limit for running each scouting query is 10ms and we have 5 scouting queries, and if the first scouting query traverses 40% of the graph in that time, we then make the assumption that the query will continue with a similar pace, hence it could finish execution in another 10-20ms in comparison to running the remaining four scouting queries in 40ms.

3.2 Reusing Scouting Query Results

One potential overhead of scouting is when throwing away perfectly correctly-computed query matches. We can alleviate this issue by introducing scouting query result reusing. By definition, all query plans return the same output results, but they can differ in the order that intermediate results are expanded to generate final. DFT-oriented engines return output matches eagerly compared to BFT. DFT engines explore systematically all the matching subtrees with the same prefix and once they move to another prefix there are no more results with that same prefix.

We use this observation for building query-plan groups. If two query plans have the same matching prefix, they belong to the same group. For instance, if we set the prefix length to one (i.e., group query plans which start with the same vertex match) for the query example `MATCH (a)->(b)->(c)` we have following grouping of query plans, starting from different vertices:

- start from (a): **(a)**->(b)->(c), **(a)**->(b)=>(c)<-(b)
- start from (b): **(b)**<-(a)=>(b)->(c), **(b)**<-(a)=>(c)<-(b), **(b)**->(c)=>(b)<-(a), **(b)**->(c)=>(a)->(b)
- start from (c): **(c)**<-(b)<-(a), **(c)**<-(b)=>(a)->(b)

The query plans within the same group can directly share scouting results and the finally selected plan will reuse those results in its execution. The results can be shared if a query traverses a whole subtree for the given matched prefix. For our example with matched prefix `(a=1)->(b=2)` it means traversing and trying to match starting from root vertex `(2)`. Thanks to DFT, we know that there are no further matches after traversing the whole subtree and we do not need to visit that part of the graph with the same prefix again.

For reusing the results, we first split the query plans into groups according to their prefix. The length of the prefix can be set statically or dynamically after analyzing the top N scouting query plans. Because the engine needs to reuse results of scouting from the same group, it keeps track of which matched prefixes were fully traversed and the result for those prefixes. After running the next scouting query from the same group, the engine should avoid those specific traversed prefixes in order not to duplicate the same work. After finishing all scouting queries from the group, we have a set of traversed prefixes and their results. After selecting the winning query plan, we can easily continue the computation from the non-visited matched prefixes and the final output is the union of results from the executed query and all the partial results collected during scouting. Notice that even when each group contains a single query plan, we can reuse the results collected during scouting execution.

The above-mentioned approach is the preferred approach for reusing the results. Another possibility for avoiding duplicated work is to mark all the matched and visited paths in the graph and store results for each of them. One can notice that this approach is memory consuming. Nevertheless, compared to the prefix approach, this can be used together with the random-walk scouting queries or potentially other approaches.

It is worth noting that result sharing with prefixes further avoids re-traversing the non-matching paths of those prefixes. In our previous example, the `(a=1)->(b=2)` prefix could lead to 10 matched `(c)` vertices and 20M non matched, e.g., because of a filter `WHERE c.value = 43`. Having completed this prefix with scouting covers both the matching and non-matching `c`s.

4 Evaluation

In this Section, we evaluate the potential benefits, as well as the overhead of scouting queries in distributed graph queries.

4.1 Experimental Settings

Implementation and Configuration. We implement a prototype of scouting queries on top of the PGX.D/Async graph query and pattern matching engine [34], which is the perfect target, as it is a distributed engine with eager, DFT-style completion of pattern matching. We configure the solution to scout the top $N+1$ query plans, where N is the number of neighbor matches per query. For instance, pattern `(a)->(b)` has $N = 1$, hence we run two scouting queries (for `(a)->(b)` and `(b)<-(a)`). This way, we allow more scouting queries for longer patterns, which are expected to also result in longer queries, while maintaining the overhead relatively contained. Furthermore, PGX.D/Async uses intermediate-result buffering for remote edges, which can result in intermediate results flowing slower towards output. To reduce the effect of buffering in the short scouting execution (which could bias the scouting metrics), we reduce the size of buffers by $\frac{1}{16}$ for scouting queries. Alternatively, one could incorporate the number of buffered intermediate results in the scouting performance metric, weighing intermediate results in the later parts of the query plan more than those in earlier parts.

Additionally, we configure scouting queries with 50ms timeout and use only the scouting query throughput as the query plan selection criterion, such that we evaluate strictly the efficiency of scouting queries. Finally, we do not implement the result sharing solutions described in Section 3.2, thus the scouting query executions are strict overhead on top of the selected query plan execution.

Query Planning in PGX.D/Async. Planning uses the operators described in Table 1 to devise potential logical query plans. It then rewrites the logical query plan with a cost-based optimizer, implemented using dynamic programming, that is based on the following heuristics [34, 41]: (i) heavily filtered vertices are preferred for the earlier stages of the plan, (ii) inspection matches increase the plan's cost, (iii) the cost of an edge match is approximately *log* of the cost of a neighbor one, as it can be implemented with a binary search in the neighbor list of the source vertex, and (iv) forward and reverse neighbor matches are equally weighed.

Hardware. We use a cluster of eight 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 processor contains 384GB of DDR4-2400 memory and LSI MegaRAID SAS-3 3108 storage. Each machines includes a Mellanox Connect-X InfiniBand card, all connected to an EDR 100Gbit/s InfiniBand network.

Graphs and queries. Our experiments use two classic graphs and the corresponding queries. First, we use the latest LDBC graph [39] (scale factor 300, with 817 million vertices and 5.27 billion edges) and a set of 12 queries derived from the LDBC Business Intelligence (BI) standard queries [38]. This is not the official LDBC standard. The scope of this evaluation covers user-provided fixed-pattern queries, thus the latest LDBC BI queries are outside of the scope of this work. We use the older/first version of BI: Out of the 24 original queries, four represent simple path patterns (i.e., Q2, Q4, Q12, Q23) and are directly used in our experiments. The remaining ones either include regular path queries (e.g., `MATCH (a)-/:knows*/->(b)`), or include subqueries in projection or filters (e.g., `SELECT ... FROM (SELECT ...) ...`). We devise simplified variants of these queries in order to support the benchmark specification as closely as possible. For example, the original Q8 is:

```
SELECT
  relatedTag, COUNT(DISTINCT cmnt) AS count
FROM GRAPH_TABLE (ldbc
  MATCH
    (m IS Post|cmnt)-[IS postHasTag|cmntHasTag]->(tag IS Tag),
    (c IS Cmnt)-[IS cmntReplyOfC|cmntReplyOfP]->(m),
    (c)-[IS cmntHasTag]->(relatedTag IS Tag)
  WHERE
    tag.name = 'Genghis_Khan'
    AND NOT EXISTS
      (SELECT * MATCH (c)-[IS cmntHasTagTag]->(tag))
  COLUMNS(relatedTag.t_name AS relatedTag, c.aa_id AS cmnt)
)
GROUP BY relatedTag
ORDER BY count DESC, relatedTag
FETCH FIRST 100 ROWS ONLY
```

from which we remove the subquery. We plan to extend the SQL/PGQ support of scouting queries in future work.

Second, we express the classic TPC-H [7] relational database workload as a graph (both the data and the queries, scale factor 300, with 2.36 billion vertices and 6.14 billion edges) and optimize with scouting queries. For instance, the workload includes region vertices, which are connected with countries, in which customer vertices reside. With SQL/PGQ, Q3 is expressed as:

```
SELECT
    l_orderkey,
    SUM(l_extendedprice * (1 - l_discount)) AS revenue,
    o_orderdate,
    o_shippriority
FROM GRAPH_TABLE (tpch
  MATCH
    (l IS lineitem)-[IS lineitem_orders]->(o IS orders),
    (o)-[IS orders_customer]->(c IS customer)
  WHERE
    c.c_mktsegment = 'BUILDING'
    AND o.o_orderdate < DATE '1995-03-15'
    AND l.l_shipdate > DATE '1995-03-15'
  COLUMNS(l.l_orderkey AS l_orderkey,
    l.l_extendedprice AS l_extendedprice,
    o.o_orderdate AS o_orderdate,
    o.o_shippriority AS o_shippriority)
)
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY revenue DESC, o_orderdate
FETCH FIRST 100 ROWS ONLY
```

We rewrite and use the 12 TPC-H standard queries that require no subquery support.

Methodology. Our approach affects primarily the pattern-matching part of graph queries and is oblivious to the post-processing operators, such as GROUP BY and ORDER BY. Accordingly, in our experiments, we report the pattern matching execution time for both non-scouting and scouting and additionally the scouting execution overhead from all N queries for scouting.

We perform 10 runs of each query and report the median latency. Scouting is not deterministic: With some queries, scouting could dictate different query plans in different runs. We thus report the plans that are selected, and how often they did so, under the result bars. For each experiment set, we execute the queries in a per-graph round-robin fashion in order to reduce caching effects.

4.2 Results

Figures 1 and 2 include the results of PGX.D/Async with and without scouting queries on LDBC and TPC-H workloads, respectively.

Overall. Scouting improves the total workload execution time, including scouting overhead, by 3.3× for LDBC and 1.7× for TPC-H.

For both LDBC and TPC-H, we see that scouting queries result in the same query plan as the default query planner for only 4 out of 22 queries. Interestingly, even for these four queries the overall performance is slightly faster with scouting queries, even though we have not enabled result reuse. Our analysis shows that the scouting query warmup gives a good performance boost to the actual execution (especially for Q4 on LDBC that is tiny). The exact overhead from scouting queries is shown in orange and depends on (i) the query execution duration and (ii) the number of neighbor matches the query includes. As mentioned earlier, we run $N + 1$ scouting queries, where N is the number of neighbor matches in that query.

Table 2: Scouting query execution statistics of TPC-H Q7. *Visited* and *Matched* correspond to vertices.

	QP1	Visited	Matched	QP3	Visited	Matched
0	N1	225	18	N2	225	18
1	s	5488	5488	c	78683	78683
2	li	3179315	966944	o	746157	746157
3	o	663104	663104	li	2594222	787010
4	c	370814	370814	s	533522	533522
5	N2	370814	14877	N1	533522	21551

A second class of queries are the ones that marginally benefit from scouting queries (i.e., Q23 on top of LDBC and Q10 on TPC-H). For those, scouting queries cause the engine to use a different query plan, however, this plan is not so much faster than the original plan. For instance, query Q10 on TPC-H benefits by 40% (280ms faster) with the new query plan, but the scouting query overheads ($N = 4$) cancel this speedup.

A third category includes queries that have overall worse performance with scouting. This happens either because the queries are very short (Q2, Q4, Q15, Q17 on LDBC) and the scouting overheads are higher than any possible benefits, or because scouting queries choose a worse plan than the default query planer. The latter only happens to Q20 from LDBC and Q19 from TPC-H. In Q20/LDBC, scouting consistently in all 10 runs returns query plan 2, while in Q19/TPC-H, scouting takes the wrong decision 6 out of 10 runs, resulting in 1.6× slower execution time, 1.8× with the scouting overhead. In both cases, the scouting timeout is too short to select the best query plan. For Q19/TPC-H, the collected statistics are almost identical in both query plans, hence it is a matter of "luck", which plan is selected. When choosing the optimal timeout, one must strike a balance between collecting more precise statistics and the overhead of executing scouting queries.

The five LDBC queries that are altogether slower with scouting are relatively short (average execution time of 1.7s) and have an 1.8× average slowdown. With TPC-H, only three queries are slower (average execution time 0.9s) with 1.3× average slowdown.

Finally, the remaining queries (Q12, Q14, and Q24 on LDBC and Q5, Q7, Q8, Q9, and Q16 on TPC-H) represent the queries where scouting helps the most. In these cases, the default query planner cannot find the best plan, while the actual scouting-query execution successfully unveils a better plan. Intuitively, these tend to be longer queries with complex connections and filters. The result is speedups from 1.3 to 8.7× maximum, with 3.4× and 3.2× average speedups for LDBC and TPC-H, respectively.

In summary, the experimental results meet our intuition: Scouting is better suited for large complex queries where (i) traditional query planning has a hard time finding the best plan and (ii) the reduction of the long execution times outweighs any scouting overheads.

Deep Dive Q7/TPC-H. We now analyze Q7 from TPC-H in order to explain the complexities of query planning in distributed graph query engines that bring about the need for dynamic planning approaches such as scouting queries. We choose Q7 as it combines the complexities of long patterns with heavy filtering. Q7 calculates the total money transfers between France and Germany over two years:

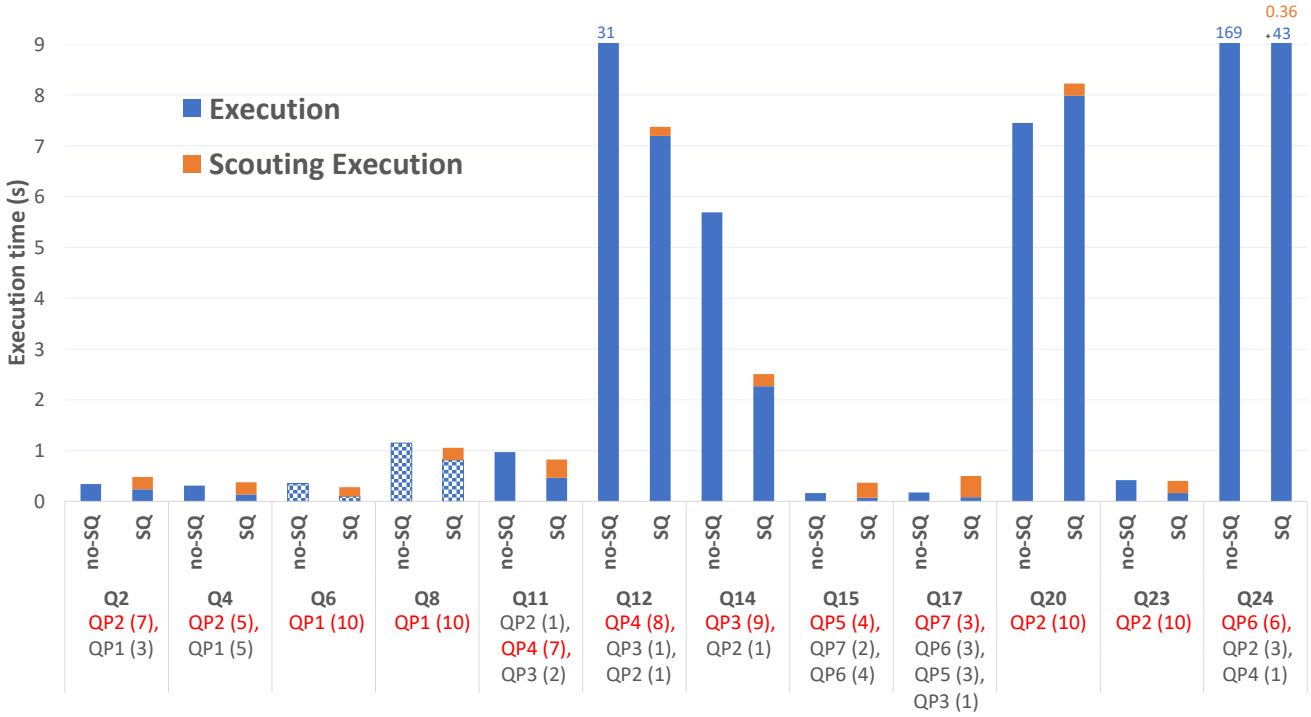

Figure 1: Query execution times for LDBC without (no-SQ) and with scouting queries (SQ). Mosaic-pattern bars represent queries where the default query planner and scouting queries give the same query plan. The query plans (QP*i*) and how many times each was chosen are listed beneath each query, sorted from best to worst, with the one preferred by scouting highlighted in red.

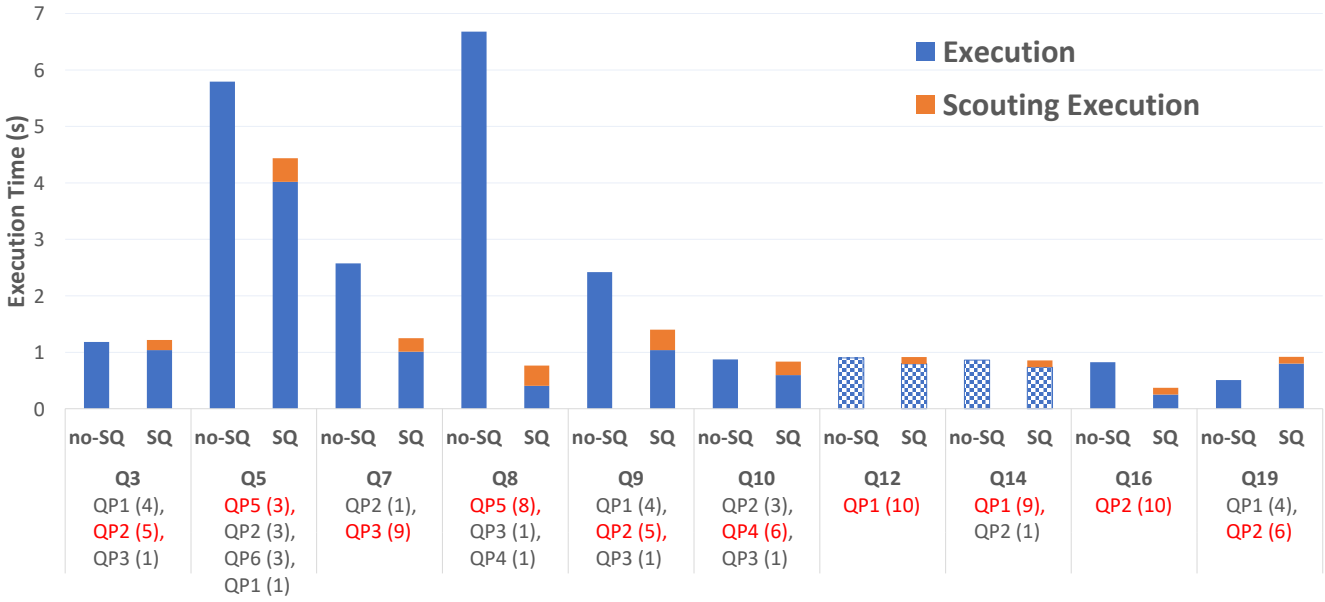

Figure 2: Query execution times for TPC-H without (no-SQ) and with scouting queries (SQ). Mosaic-pattern bars represent queries where the default query planner and scouting queries give the same query plan. The query plans (QP*i*) and how many times each was chosen are listed beneath each query, sorted from best to worst, with the one preferred by scouting highlighted in red.

```
SELECT
  supp_nation,
  cust_nation,
  EXTRACT(YEAR FROM l_shipdate) AS l_year,
  SUM(l_extendedprice * (1 - l_discount)) AS revenue
FROM GRAPH_TABLE (tpch
  MATCH
    (li IS lineitem)-[IS lineitem_supplier]->(s IS supplier),
    (li)-[IS lineitem_orders]->(o IS orders),
    (o)-[IS orders_customer]->(c IS customer),
    (s)-[IS supplier_nation]->(n1 IS nation),
    (c)-[IS customer_nation]->(n2 IS nation)
  WHERE
    ((n1.n_name = 'FRANCE' AND n2.n_name = 'GERMANY')
      OR (n1.n_name = 'GERMANY' AND n2.n_name = 'FRANCE'))
    AND l_shipdate BETWEEN
       DATE '1995-01-01' AND DATE '1996-12-31'
  COLUMNS(li.l_shipdate AS l_shipdate,
    li.l_extendedprice AS l_extendedprice,
    li.l_discount AS l_discount,
    n1.n_name AS supp_nation,
    n2.n_name AS cust_nation)
)
GROUP BY supp_nation, cust_nation, l_year
ORDER BY supp_nation, cust_nation, l_year
```

The query plan proposed by the default planner (QP1) starts from the supplier nation, goes to the supplier's line-items, to the orders, to the customers, and then to the second nation. Scouting queries instead choose the third query plan (QP3) that starts from the customer nation, moves to the customer, the orders, the line-items, the suppliers, and the suppliers' nation. The default query planner gives 3× lower cardinality to QP1 than QP3, thus it is certain about the choice of this particular plan.

However, the default query planner misestimates how selective the line-item filter is, bringing line-item matching as early in the plan as possible. As we see in Table 2, this choice leads to an early explosion of visited vertices in Stage 2 (`li`), compared to the smaller and later explosion in Stage 3 (`li`) for QP3. Looking at the query, this explosion makes a big difference, leading to a greater number of intermediate matches that need to extract data out of the line-item vertices for projections and to support the group-by operation, i.e., `EXTRACT(YEAR FROM l_shipdate)`, and `SUM(l_extendedprice * (1 - l_discount))`. This data needs to be carried across machines in a distributed engine, leading to overheads. In practice, expressing all these fine details in a default query planner is almost impossible.

5 Concluding Remarks

This paper introduces scouting queries, a mechanism built on top of traditional query planners to enable selecting the best query plan for pattern matching in (distributed) graph queries. An efficient query plan is a critical part of any query engine significantly affecting its performance. Our approach uses runtime statistics collected during the execution of candidate query plans and is thus able to correct any misestimations of the default query planner. Our evaluation shows that scouting queries can significantly improve performance on real graphs, while maintaining low overhead.

In future work, we intend to test BFT-style scouting matching, scouting-result reusing and different scouting metrics. Furthermore, we plan to use scouting queries in a feedback loop to improve query planners, both manually and with machine-learning techniques.

References

[1] Gremlin – A Graph Traversal Language. https://github.com/tinkerpop/gremlin.
[2] Neo4j. https://neo4j.com/.
[3] Neo4j Cypher Query Language – Developer Guides. https://neo4j.com/developer/cypher/.
[4] PGQL 1.5 Specification. https://pgql-lang.org/spec/1.5/.
[5] Resource Description Framework (RDF) Model and Syntax Specification. https://www.w3.org/TR/PR-rdf-syntax/.
[6] SPARQL Query Language for RDF – SPARQL Protocol and RDF Query Language. https://www.w3.org/TR/rdf-sparql-query/.
[7] TPC-H Decision Support Benchmark. https://www.tpc.org/tpch/.
[8] Ashraf Aboulnaga and Surajit Chaudhuri. Self-Tuning Histograms: Building Histograms without Looking at Data. SIGMOD, 1999.
[9] G. Antoshenkov. Dynamic query optimization in Rdb/VMS. ICDE, 1993.
[10] Gennady Antoshenkov and Mohamed Ziauddin. Query Processing and Optimization in Oracle Rdb. *The VLDB Journal*, 5(4), 1996.
[11] Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive Re-Optimization. SIGMOD, 2005.
[12] Pedro Bizarro, Shivnath Babu, David DeWitt, and Jennifer Widom. Content-Based Routing: Different Plans for Different Data. PVLDB, 2005.
[13] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. Smooth Scan: Robust Access Path Selection without Cardinality Estimation. *The VLDB Journal*, 27(4), 2018.
[14] Nicolas Bruno and Surajit Chaudhuri. Conditional Selectivity for Statistics on Query Expressions. SIGMOD, 2004.
[15] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: A Multidimensional Workload-Aware Histogram. SIGMOD, 2001.
[16] S. Chaudhuri and V. Narasayya. Automating statistics management for query optimizers. ICDE, 2000.
[17] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. A Pay-as-You-Go Framework for Query Execution Feedback. *Proc. VLDB Endow.*, 1(1), 2008.
[18] Chungmin Melvin Chen and Nick Roussopoulos. Adaptive Selectivity Estimation Using Query Feedback. SIGMOD, 1994.
[19] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-Miner: An Efficient Task-Oriented Graph Mining System. EuroSys, 2018.
[20] Soumyava Das and Sharma Chakravarthy. Duplicate Reduction in Graph Mining: Approaches, Analysis, and Evaluation. *IEEE KDE*, 30(8), 2018.
[21] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive Query Processing. *Foundations and Trends® in Databases*, 1(1), 2007.
[22] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. Graph Pattern Matching in GQL and SQL/PGQ. *SIGMOD*, 2022.
[23] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A General-Purpose Graph Pattern Mining System. SIGMOD, 2019.
[24] Kwanchai Eurviriyanukul, Norman W. Paton, Alvaro A A Fernandes, and Steven J. Lynden. Adaptive join processing in pipelined plans. EDBT, 2010.
[25] Per Fuchs, Peter Boncz, and Bogdan Ghit. EdgeFrame: Worst-Case Optimal Joins for Graph-Pattern Matching in Spark. GRADES-NDA, 2020.
[26] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A Pattern-Aware Graph Mining System. EuroSys, 2020.
[27] Navin Kabra and David J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. SIGMOD, 1998.
[28] Quanzhong Li, Minglong Shao, Volker Markl, Kevin Beyer, Latha Colby, and Guy Lohman. Adaptively Reordering Joins during Query Execution. ICDE, 2007.
[29] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making Learned Query Optimization Practical. SIGMOD, 2021.
[30] Ryan Marcus and Olga Papaemmanouil. Towards a Hands-Free Query Optimizer through Deep Learning. *CoRR*, abs/1809.10212, 2018.
[31] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. Robust Query Processing through Progressive Optimization. SIGMOD, 2004.
[32] Daniel Mawhirter and Bo Wu. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. SOSP, 2019.
[33] Mohamed Ramadan, Ayman El-Kilany, Hoda M. O. Mokhtar, and Ibrahim Sobh. RL_QOptimizer: A Reinforcement Learning Based Query Optimizer. *IEEE Access*, 10, 2022.
[34] 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. GRADES, 2017.
[35] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. SIGMOD, 1979.

[36] U. Srivastava, P.J. Haas, V. Markl, M. Kutsch, and T.M. Tran. ISOMER: Consistent Histogram Construction Using Query Feedback. ICDE, 2006.
[37] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. PVLDB, 2001.
[38] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter Boncz, Vlad Haprian, and János Benjamin Antal. An Early Look at the LDBC Social Network Benchmark's Business Intelligence Workload. GRADES-NDA, 2018.
[39] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter A. Boncz. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proc. VLDB Endow.*, 16(4), 2022.
[40] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. SOSP, 2015.
[41] 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. USENIX ATC, 2021.
[42] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. *ACM Trans. Database Syst.*, 46(3), 2021.
[43] Da Yan, Hongzhi Chen, James Cheng, M. Tamer Özsu, Qizhen Zhang, and John C. S. Lui. G-thinker: Big Graph Mining Made Easier and Faster. *CoRR*, abs/1709.03110, 2017.
[44] Jihad Zahir and Abderrahim El Qadi. A Recommendation System for Execution Plans Using Machine Learning. *Mathematical and Computational Applications*, 21(2), 2016.
[45] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic Plan Migration for Continuous Queries over Data Streams. SIGMOD, 2004.