Conjunctive Path Query Generation for Benchmarking

Roan Hofland
Eindhoven University of Technology
r.w.p.hofland@student.tue.nl

21st of March, 2022

Supervisors
George Fletcher (g.h.l.fletcher@tue.nl)
Yuya Sasaki (sasaki@ist.osaka-u.ac.jp)
Seiji Maekawa (maekawa.seiji@ist.osaka-u.ac.jp)

Contents

1 Introduction **3**

2 Related Work **3**
2.1 Conjunctive Path Queries . . . 3
2.2 gMark . . . 4
2.2.1 Query Selectivity . . . 4
2.2.2 Graph Schema . . . 4
2.2.3 Selectivity Class Algebra . . . 4
2.2.4 Schema Graph . . . 5
2.2.5 Selectivity Graph . . . 6

3 Theory **6**
3.1 Selectivity Algebra Extension . . . 6
3.2 Edge Graph . . . 6
3.2.1 Extending the Edge Graph . . . 7

4 Algorithm **8**
4.1 Arguments . . . 8
4.2 Steps . . . 8

5 Pseudocode **9**
5.1 Arguments . . . 9
5.2 Skeleton generation . . . 9
5.2.1 Other called functions . . . 9
5.3 Path generation . . . 9
5.3.1 Other called functions . . . 10

6 Performance Evaluation **11**
6.1 Runtime Scaling . . . 11
6.2 Edge Graph Scaling . . . 11

7 Concluding Remarks & Limitations **13**
7.1 Future Work . . . 13
7.1.1 Runtime Improvements . . . 13
7.1.2 Theory Improvements . . . 13
7.1.3 Technical Improvements . . . 13

A Getting started with gMark **14**

Version History

Version	Date	Sections	Comment
v1.0	2021-09-30	4	Original concept
v1.1	2021-10-13	5	Pseudocode
v2.0	2021-11-23	1, 2, 3, 4 & 5	Updates & move towards a report style
v2.1	2021-11-24	4 & 5	Small fixes
v2.2	2021-12-20	1 & 2	Introduction & Related Work
v2.3	2021-01-11	1, 2, 3, 4, 5, 6 & 7	Extensions, Related Work & Future Work
v2.4	2022-02-10	1, 2, 3, 4, 5, 6 & 7	Improvements, Performance & Conclusion
v2.5	2022-02-16	1, 2, 3, 4 & A	Apply feedback from George
v2.6	2022-02-17	2	Correct CPQ example
v2.7	2022-03-10	2	Apply feedback from Yuya
v2.8	2022-03-21	1, 2, 3, 4, 5, 6, 7 & A	Appendix & Apply feedback from Mireille

Time spent: capita selecta 166 hours, gMark rewrite 119 hours

1 Introduction

Conjunctive path queries (CPQ) are one of the most frequently used queries for complex graph analysis. However tailored support for them in various domains is lacking. One such domain is benchmark generation. The goal of this report is to propose a scalable method for generating CPQs for benchmarking purposes, while also allowing the user to control the selectivity of the generated queries. To accomplish this we will build upon the query generation framework presented in gMark, an open-source query workload generator that makes use of regular path queries (RPQ) instead of CPQs.

Since CPQs are so common in complex graph analysis, being able to generate them for benchmarking purposes can aid in the development and optimisation of graph database systems. In addition, by being able to control the characteristics of the generated queries, it becomes possible to compare graph database systems on their suitability for a specific use case.

An open-source implementation of the algorithm presented in this report will be provided as part of a rewrite of the gMark software[1]. Detailed instructions for getting started with gMark for both normal usage and development work can be found in Appendix A.

2 Related Work

In this section some previous work will be presented that the proposed approach builds on. Special attention will be given to specific concepts and data structures that are also used by the proposed CPQ generation method.

2.1 Conjunctive Path Queries

As stated in Sasaki et al. [2] conjunctive path queries (CPQ) are a basic graph query language that covers more than 99% of query shapes that appear in practice. Hence optimisations specifically targeted at CPQs will benefit a large majority of real world use cases. To illustrate how CPQs work we will show how to construct a query to find people who know someone who knows them. However, before that we have to formalise the exact definition of a CPQ.

A conjunctive path query can be recursively constructed from the operations of identity 'id', edge label 'l', inverse edge label 'l^-', join '$\circ$' and conjunction '$\cap$' (also referred to as intersection in this report). This results in the following grammar:

$$CPQ ::= id \mid l \mid l^- \mid CPQ \circ CPQ \mid CPQ \cap CPQ \mid (CPQ)$$

The exact definition of these operations when evaluated on a graph $\mathcal{G}$ with vertex set $\mathcal{V}$, edge label set $\mathcal{L}$ and edge set $\mathcal{E} = \mathcal{V} \times \mathcal{V} \times \mathcal{L}$ is then given as follows:

$$\begin{aligned}
[\![id]\!]_\mathcal{G} &= \{(v,v) \mid v \in \mathcal{V}\} \\
[\![l]\!]_\mathcal{G} &= \{(v,u) \mid (v,u,l) \in \mathcal{E}\} \\
[\![l^-]\!]_\mathcal{G} &= \{(u,v) \mid (v,u,l) \in \mathcal{E}\} \\
[\![q_1 \circ q_2]\!]_\mathcal{G} &= \{(v,u) \mid \exists m \in \mathcal{V} : (v,m) \in [\![q_1]\!]_\mathcal{G} \wedge (m,u) \in [\![q_2]\!]_\mathcal{G}\} \\
[\![q_1 \cap q_2]\!]_\mathcal{G} &= \{(v,u) \mid (v,u) \in [\![q_1]\!]_\mathcal{G} \wedge (v,u) \in [\![q_2]\!]_\mathcal{G}\} \\
[\![(q_1)]\!]_\mathcal{G} &= [\![q_1]\!]_\mathcal{G}
\end{aligned}$$

This means that the result of evaluating a CPQ is a set of vertex pairs. In addition we define the concept of the diameter of a CPQ as the maximum number of edge labels to which the join operation is applied, essentially the longest path from source to target within the CPQ itself. More information about CPQs and their properties can be found in the paper by Sasaki et al. [2].

Given the formal definitions we now show how to answer our original question. Note that finding people you know who also know you is equivalent to finding a path consisting of 2 knows edges that starts and ends at the same node. We can write the following CPQ for this $(\texttt{knows} \circ \texttt{knows}) \cap id$, where we add the conjunction with identity to ensure we only return results that end and start at the same node. To make the example more concrete we show a simple social network in Figure 1. Evaluating the suggested CPQ then gives the following results $[\![(\texttt{knows} \circ \texttt{knows}) \cap id]\!]_\mathcal{G} = \{(\text{Alice}, \text{Alice}), (\text{Bob}, \text{Bob})\}$. Note that we can formalise the same query in a slightly more complex way to better showcase the conjunction and inverse operation. For this we will first intersect the `knows` predicate with the inverse `knows` predicate to get all pairs of nodes with a `knows` predicate between them in both directions. So we start with $[\![\texttt{knows} \cap \texttt{knows}^-]\!]_\mathcal{G} = \{(\text{Alice}, \text{Bob}), (\text{Bob}, \text{Alice})\}$. We can then extend the path back to the node we came from and possibly to other nodes via either the normal `knows` predicate

[1] https://github.com/RoanH/gMark

or the inverse version. To better highlight the issues with this approach we use the normal knows predicate to get $[\![(\texttt{knows} \cap \texttt{knows}^-) \circ \texttt{knows}]\!]_{\mathcal{G}} = \{(\text{Alice}, \text{Alice}), (\text{Bob}, \text{Bob}), (\text{Bob}, \text{Charlie})\}$. This shows the hinted issue that we possibly do not end up where we came from, but we can resolve this by adding a final conjunction with the identity operation. Thus the final alternative query becomes $[\![((\texttt{knows} \cap \texttt{knows}^-) \circ \texttt{knows}) \cap id]\!]_{\mathcal{G}} = \{(\text{Alice}, \text{Alice}), (\text{Bob}, \text{Bob})\}$.

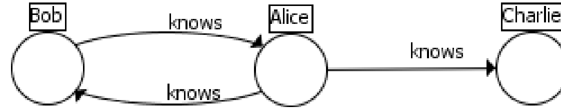

Figure 1: A simple social network graph $\mathcal{G}$.

2.2 gMark

In their paper Bagan et al. [1] introduced gMark, a domain- and query language-independent graph instance and query workload generator. For this report the main focus will be on the query workload generation, which heavily inspired the approach used to generate workloads with CPQs. As such this section will summarise some core concepts that were reused from gMark. The core concepts we will summarise are: query selectivity, the graph schema, selectivity class algebra, schema graph, and the selectivity graph.

Generating a query is a two step process in gMark. First, a top level query skeleton is generated made up of so called conjuncts. Second, each of these conjuncts is filled with a query of its own. In gMark RPQs are used to fill these conjuncts, in this report we will show how to replace the RPQs with suitable CPQs instead. We do not make any modifications to the skeleton generation, this means that we retain all of the existing features pertaining to skeleton generation. Due to this we support all the query shapes supported in gMark and even support having a Kleene star on a conjunct.

2.2.1 Query Selectivity

The selectivity of a query gives information about how the number of selected result set tuples grows as the graph it is being evaluated on grows in size. For example, selecting all nodes in a graph with n vertices returns n result set tuples. Similarly, selecting everything from the Cartesian product of a graph with n vertices with itself results in n^2 result set tuples.

Within gMark we classify queries with one of three selectivities: constant, linear, or quadratic. Constant selectivity queries select a number of results from the graph that remains essentially the same no matter the size of the graph. Queries like this are often associated with graph nodes that are present in fixed quantities in the graph, like the number of continents. Linear selectivity queries select a number of results from the graph that grows at approximately the same rate as the graph itself. These queries are often associated with graph nodes that are also present in a quantity that grows linearly with the size of the graph, such as the number of people in a social network. The quadratic selectivity is for queries that select a number of results that grows quadratically with the size of the graph. Quadratic queries often perform a Cartesian product, such as selecting all pairs of people in a social network.

2.2.2 Graph Schema

A graph schema in gMark is an integral component for both graph generation and query generation. The purpose of the graph schema is to describe the structure of a concrete graph instance that conforms to the graph schema. As such the graph schema is used to guide the random graph generation process. Similarly, it is used for query generation to ensure that queries actually match some part of the graph and to estimate the selectivity of queries. A graph schema is itself a graph and made up of nodes that represent graph node types and edges that model which node types can be connected by an edge. In addition, all edges have an in degree and an out degree distribution, making it possible to encode information about the relationship between node types. For example, we could encode that the number of authors of a single paper follows a normal distribution. Finally, for each node type it is also specified if the node type is present in a fixed quantity (and if so the exact quantity is also specified) in the graph, or if the number of nodes of the type grows together with the overall size of the graph.

2.2.3 Selectivity Class Algebra

To estimate the selectivity of a given query we essentially break it up into pieces and then combine these pieces into larger pieces while keeping track of the selectivity. In order to do this we need a construct to keep track of

this selectivity and a base case smallest piece we can directly derive this construct from.

The construct we introduce for this is selectivity classes and we can directly derive the selectivity class from an edge in the graph schema. To do this we look at the source and target node types and the in and out distribution of the edge. For the node types we specifically check if these nodes are present in a constant quantity (1) or growing with the size of the graph (N). This results in triplets of the form ($src \in \{1, N\}, o, trg \in \{1, N\}$) where o is an operation between types denoted by $o \in \{=, <, >, \diamond, \times\}$ as we will briefly describe next.

The $=$ operation usually occurs between constant types (1). The $<$ operation either occurs when the out-degree distribution is Zipfian or when the source node is of a constant type (1) and the target node of a growing type (N). The definition of $>$ is symmetric to $<$. The $\times$ operation corresponds to a Cartesian product between two node sets that are both of a growing type (N). Finally, the $\diamond$ operation is commonly seen between two hub nodes in a graph (there are a lot of paths between two hubs, but overall the number remains linear). More details and examples are described in the original paper by Bagan et al. [1].

To compute the selectivity of the query as a whole we then take edges of the path specified by the query and compute the selectivity class by adding one edge at a time. A special selectivity class algebra was developed in Bagan et al. [1] to compute the result of the conjunction[2] and disjunction of two selectivity classes. The latter is used for regular path queries (RPQ). To support CPQs we only need conjunction. Table 1 shows the result of extending a query with a specific selectivity class with an edge with a certain selectivity class.

$\circ$	$=$	$<$	$>$	$\diamond$	$\times$
$=$	$=$	$<$	$>$	$\diamond$	$\times$
$<$	$<$	$<$	$\times$	$\times$	$\times$
$>$	$>$	$\diamond$	$>$	$\diamond$	$\times$
$\diamond$	$\diamond$	$\diamond$	$\times$	$\times$	$\times$
$\times$	$\times$	$\times$	$\times$	$\times$	$\times$

Table 1: Conjunction selectivity class algebra (read in column-row order).

2.2.4 Schema Graph

The schema graph is a labelled directed graph between selectivity types derived from the graph schema. This graph shows how the selectivity class of a query path through the graph changes when extending it with an edge with a specific (inverse) label. A selectivity type refers to the combination of a graph node type and a selectivity class. An example of a schema graph is given in Figure 2. The graph schema for this example has 3 node types `T1`, `T2`, `T3` and 2 predicates `a` and `b`. In this schema graph we can see that for example we can extend a path ending at a node with type `T1` and selectivity class `(N,=,N)` with an edge with label `a` to end at a node with type `T1` and selectivity class `(N,<,N)`.

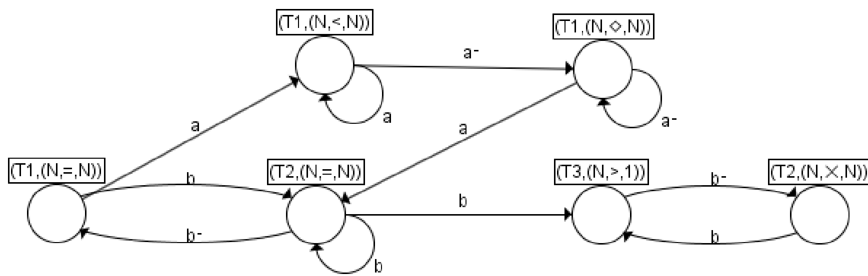

Figure 2: Schema graph.

[2]Conjunction in gMark is not the same operation as conjunction for CPQs, it is instead similar to the join operation.

2.2.5 Selectivity Graph

The selectivity graph is an unlabelled directed graph between selectivity types. Given a maximum path length, this graph shows between which selectivity types a path of at most the given maximum length exists. This graph is used to select the selectivity types to draw a path between in the schema graph when generating queries. An example selectivity graph is shown in Figure 3.

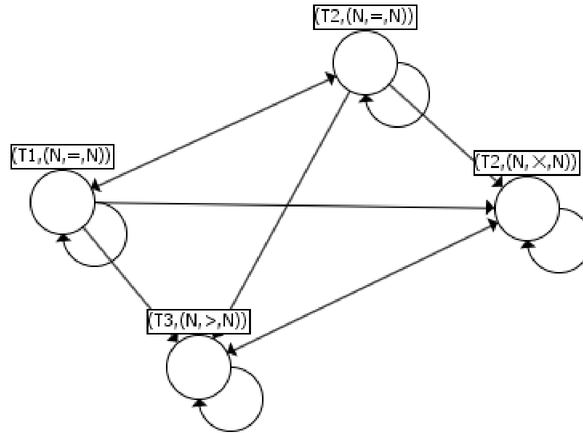

Figure 3: Selectivity graph.

3 Theory

In this section new theory and data structures will be introduced that are used to generate CPQs.

3.1 Selectivity Algebra Extension

The selectivity class algebra originally proposed in Bagan et al. [1] does not cover all the operations required to support generating CPQs with a specific selectivity. We need to extend it with two extra rules to support the conjunction and identity operations:

- **Conjunction:** When intersecting two paths, the final selectivity is at most the selectivity of the least selective path in the conjunction.
- **Identity:** Extending a path with identity does not change the path in a meaningful way, thus the only case that needs to be accounted for is intersecting a path with identity. When this is done we can follow the normal selectivity algebra. However, the cycle that is created in this way can never yield a quadratic selectivity.

3.2 Edge Graph

The edge graph is a graph constructed from an input graph such that edges in the original input graph are nodes in the edge graph. In addition, nodes are connected when a node exists between them in the original graph. Although there may be more applications of the general technique, the input graph that will be used for CPQ generation is the schema graph. The purpose of the edge graph is to encode all possible paths between two preselected nodes in the input graph. For this purpose two special nodes are added to the edge graph to represent the `source` and `target`. All paths in the edge graph must originate from the source node and end at the target node. For the algorithm proposed in this report, the `source` and `target` nodes represent specific nodes chosen from the schema graph. We will also impose a maximum length on paths used to construct the edge graph. The primary reason for this is to deal with cycles in the input graph, which may cause an infinite number of distinct paths to exist between two nodes. In the context of CPQ generation we will use the edge graph to model all paths that exist between two nodes of the schema graph.

Example: Consider the excerpt from a schema graph shown in Figure 4 and assume that we have preselected `(journal,(N,=,N))` as our `source` node and `(paper,(N,×,N))` as our `target` node. Furthermore, assume we set a maximum path length of 4.

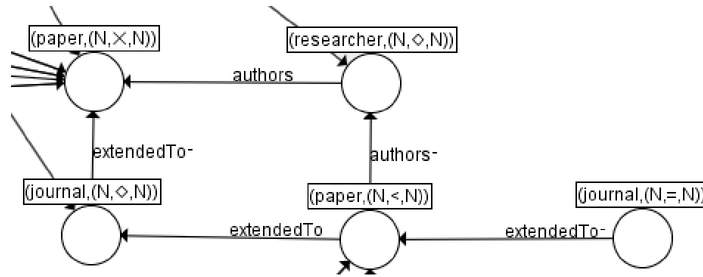

Figure 4: Schema graph excerpt.

The base edge graph we then obtain is shown in Figure 5. This is easy to verify as there are only two paths of length at most 4 between the selected `source` and `target` node in the schema graph.

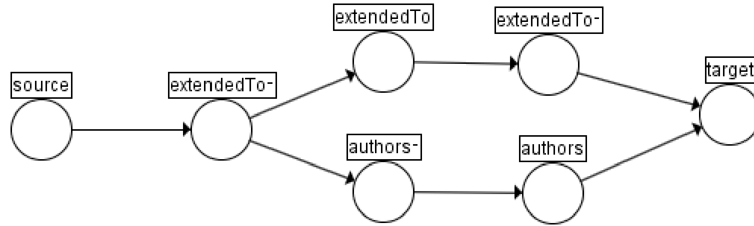

Figure 5: Base edge graph.

3.2.1 Extending the Edge Graph

The base edge graph is simply another way to encode the information that is already present in the original graph. The main goal when extending the base edge graph is to add new nodes that represent transitions with an intersection of multiple paths. The way this is done is by finding paths in the edge graph that run parallel between two nodes, for example, in Figure 5 there are two paths between the leftmost `extendedTo`$^-$ node and the `target` node. The resulting intersection node that would be added for this can be seen in Figure 6.

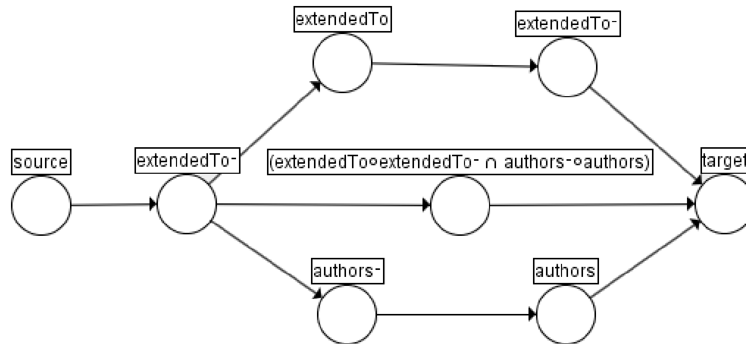

Figure 6: Edge graph with intersection (recursive depth 1).

This process can be repeated for a number of iterations, such that the newly added nodes can become part of intersections themselves. Similar nodes can also be added where the intersection is with identity, for this we simply take a path without looking for a parallel path.

For both operations it is important that we do not break the selectivity information originally captured in the schema graph. For the intersection between two parallel paths this is easy to see, these two paths already ended at the same node, meaning both paths result in the same selectivity and therefore their intersection still has the same selectivity. For identity there is more to consider. First of all, the edge graph nodes need to link back to nodes of the same type in the schema graph. Secondly, the target node needs to have a selectivity class that is at most linear. The translation of the edge graph nodes back to schema graph nodes should be done as follows. For the source node of the path the source node of the corresponding edge in the schema graph should be used and for the target node of the path the target node of the corresponding edge in the schema graph should be used.

4 Algorithm

In this section a structured approach to generate queries will be presented based on the theory explained in Section 3. Actual pseudocode for some of the steps explained in the subsections will be presented in Section 5.

4.1 Arguments

Query generation takes a number of arguments, with some being used for CPQ generation and others simply being passed on directly to the gMark skeleton generation logic.

- For skeleton generation: a minimum and maximum number of conjuncts $0 < c_{min} \leq c_{max}$, a list of desired query shapes $f \subseteq \{\text{star}, \text{cycle}, \text{star-chain}, \text{chain}\}$, a minimum and maximum query arity $0 \leq ar_{min} \leq ar_{max}$, a conjunct Kleene star probability $0 \leq p_{star} \leq 1$, and a graph schema.
- A desired query selectivity $s \in \{\text{constant}, \text{linear}, \text{quadratic}\}$ (also used for skeleton generation).
- A maximum diameter for the CPQs $0 < l_{max}$.
- A maximum recursive depth $0 \leq r_{max}$.

4.2 Steps

1) Compute the schema graph and the selectivity graph using the graph schema and c_{max}.

2) Draw a random query skeleton composed of c with $c_{min} \leq c \leq c_{max}$ conjuncts from the selectivity graph following the input skeleton generation arguments and with a desired selectivity s. This step is exactly the same as what gMark does to generate a skeleton.

- This gives us a sequence of conjuncts $C_1, C_2, \ldots, C_c$ with each conjunct having a source and target selectivity type originally corresponding to a node in the selectivity graph.

3) Now that we have the top level structure of the query we need to generate the individual CPQ queries that go in each conjunct. The following steps should be executed for every query conjunct C in the skeleton.

4) For every conjunct C we need to find paths that connect the selected nodes we picked from the selectivity graph, we can find these in the schema graph (see Figure 7). For example, assuming we picked node `(T1,(N,=,N))` as the first node and `(T3,(N,>,1))` as the next node and a maximum diameter of 4, then there are exactly 6 paths that are possible: `bb`, $\texttt{bb}^-\texttt{bb}$, $\texttt{bbb}^-\texttt{b}$, `bbb`, `bbbb` and $\texttt{aa}^-\texttt{ab}$. If paths we select follow the exact same schema graph edges then we can also have intersections deeper in the expression. For example, we could intersect $\texttt{aa}^-\texttt{ab}$ and `bb` by intersecting subpaths that start and end at the same nodes. In this case we would get $(\texttt{aa}^-\texttt{a} \cap \texttt{b}) \circ \texttt{b}$. This idea can be applied recursively, in some cases we can also add an intersection with id if the source and target nodes are of the same type and the result selectivity is at most linear. The edge graph introduced in Section 3.2 is designed to model these paths.

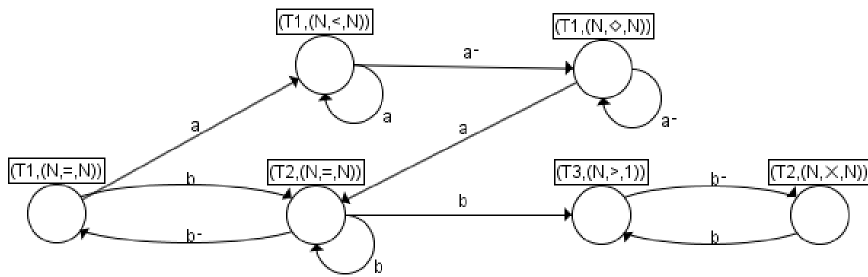

Figure 7: Reference schema graph.

5) The next step is then to create an edge graph (see Section 3.2) for every conjunct C with the source node of the conjunct as the `source` node of the edge graph and the target node of the conjunct as the `target` node of the edge graph. The path length and recursive depth are given by the input arguments l_{max} and r_{max} respectively.

6) After each edge graph is constructed we randomly draw one path in each from `source` to `target` with a length of at most l_{max}. The concatenation of the labels on the nodes in this path is the final CPQ for the conjunct.

7) Finally all conjuncts with their generated CPQs are returned as the final query.

5 Pseudocode

In this section pseudocode for the main algorithms required to implement the proposal in Section 4 will be presented.

5.1 Arguments

- C the graph configuration as given by gMark.
- s a desired query selectivity, this is either constant, linear or quadratic.
- W a collection of input arguments only relevant for skeleton generation such as the minimum and maximum number of conjuncts, list of allowed query shapes, minimum and maximum query arity, and conjunct Kleene star probability.
- l_{max} a maximum diameter, defined as the longest path through a generated CPQ by taking the longest path in every intersection.
- r_{max} the maximum recursive depth to go to when generating conjunct CPQs.

5.2 Skeleton generation

Algorithm 1 shows how the top level skeleton is created and used.

Algorithm 1 Query Generation

```
1: procedure GenerateQuery(C, s, W, l_max, r_max)          ▷ See Section 5.1.
2:     G_S ← ComputeSchemaGraph(C)
3:     G_sel ← ComputeSelectivityGraph(C)
4:     Q ← GenerateSkeleton(C, s, W, G_S, G_sel)
5:     for C ∈ Q do                                        ▷ For each conjunct.
6:         C.cpq ← SchemaCPQ(G_S, C.source, C.target, l_max, r_max)
7:     end for
8:     return Q
9: end procedure
```

5.2.1 Other called functions

- `GenerateSkeleton`(C, s, W, G_S, G_{sel}) generates a query skeleton as described in Bagan et al. [1] following the specifications given by C, W and s.
- `ComputeSchemaGraph`(C) computes the schema graph for the given graph configuration.
- `ComputeSelectivityGraph`(C) computes the selectivity graph for the given graph configuration.
- `SchemaCPQ`$(G_S, s, t, l_{max}, r_{max})$ see Algorithm 2.
- `C.cpq` refers to the inner CPQ of a conjunct.
- `C.source` refers to the source selectivity type of a conjunct.
- `C.target` refers to the target selectivity type of a conjunct.

5.3 Path generation

Algorithm 2 shows the pseudocode for the algorithm to generate random CPQs between two predetermined nodes on the schema graph. Algorithm 3 shows how the edge graph is constructed.

Algorithm 2 Generating schema graph CPQ's

```
1: procedure SchemaCPQ(G_S, s, t, l_max, r_max)   ▷ Schema graph, source, target, diameter, recursion.
2:     G ← ConstructEdgeGraph(G_S, l_max, s, t, r_max)
3:     p ← DrawPath(G)
4:     return ToSchema(p)
5: end procedure
```

Algorithm 3 Edge graph construction

```
 1: procedure ConstructEdgeGraph(G_S, s, t, l_max, r_max)   ▷ Schema graph, source, target, arguments.
 2:     G ← new empty graph
 3:     AddVertex(G,'src')
 4:     AddVertex(G,'trg')
 5:     for e ∈ G.edges do
 6:         AddVertex(G, e.id)
 7:     end for
 8:     P ← ComputeAllPaths(G_S, l_max, s, t)
 9:     for p ∈ P do
10:         AddUniqueEdge(G,'src', p_0)
11:         for i ← 0 to |p| − 1 do
12:             AddUniqueEdge(G, p_i, p_{i+1})
13:         end for
14:         AddUniqueEdge(G, p_{|p|−1},'trg')
15:     end for
16:     for r ← 1 to r_max do
17:         I ← ComputeParallelPaths(G)
18:         for i ∈ I do
19:             v ← AddUniqueVertex(G, i.id)
20:             AddUniqueEdge(G, i.source, v)
21:             AddUniqueEdge(G, v, i.target)
22:         end for
23:     end for
24:     return G
25: end procedure
```

5.3.1 Other called functions

- `ComputeAllPaths`(G, l_{max}, s, t) computes and returns all paths of length at most l_{max} between vertex s and vertex t in graph G.
- `AddVertex`$(G, label)$ adds a new vertex with the given label to the given graph G.
- `AddUniqueEdge`(G, s, t) adds a new edge between the given source s and target t vertices in the given graph G. If such an edge already exists then the edge is not added again.
- `e.id` for some edge e the value of $e.id$ uniquely identifies the edge.
- `ComputeParallelPaths`(G) computes and returns (all) pairs of paths that start and end at the same vertex. Effectively these paths run in parallel between two vertices in the graph. It is worth noting that pairs of paths are not returned twice. Assuming that path a and b run parallel, then only $\{a, b\}$ or $\{b, a\}$ is returned, but never both. In addition, paths that can be intersected with identity are also returned.
- `AddUniqueVertex`$(G, label)$ adds a new vertex with the given label to the given graph G. If a vertex with the same label already exists then the vertex is not added again. This subroutine also returns the vertex with the given label.
- `i.id` this is a unique id for two parallel paths and also distinct from `e.id`.
- `i.source` this is the shared source vertex of two parallel paths.
- `i.target` this is the shared target vertex of two parallel paths.
- `DrawPath`(G) computes and returns a random path between the edge graph source and target nodes consisting of at most l_{max} vertices (excluding the source and target node).
- `ToSchema`(p) converts the given path in the edge graph to its equivalent path in the schema graph. This means resolving elements back to their associated schema graph edges (and corresponding labels).

6 Performance Evaluation

In this section we will explore the scalability of the suggested approach by looking at the runtime scaling on a small and a large graph schema. All tests were executed on a system running an Intel Core i7-7700k (4.58 GHz) processor with 32GB of RAM (2933 MHz) and used only a single thread for computations. It is also worth noting that there is considerable variation between runs even when all parameters are the same due to randomisation in various algorithms involved in the query generation. Summing the data from a large number of runs mitigated this issue, but did not completely smooth out the variation for all tests.

Exact details for the graph schemas used for testing can be found in Table 2. The schemas themselves are available in the original gMark repository[3]. The workload used for runtime testing for both schemas had the number of conjuncts set to exactly 4, the multiplicity set to 0.5, an arity of 0 – 4, any selectivity (constant, linear or quadratic) and any query shape (chain, star, cycle or star-chain). Unless otherwise specified for a specific test the maximum recursive depth was 5 and the maximum CPQ diameter 4.

Name	Types	Predicates	Edges
Test Schema	4	4	4
Shop Schema	24	82	171

Table 2: Schema specification.

6.1 Runtime Scaling

We report the time it takes to generate 100 queries according to the given workload specification. Since every query contains 4 conjuncts this comes down to generating at least 400 CPQs. Only limited data could be collected for the diameter scalability on the large graph schema due to the prohibitively long runtime, even when lowering the recursive depth down from 5 to 1. The measured times are shown in Figure 8 and show a clear exponential trend for both parameters. Between the recursive depth and the maximum diameter, the maximum diameter seems to have a larger effect on the total runtime. The size of the graph schema also has a major influence on the total runtime.

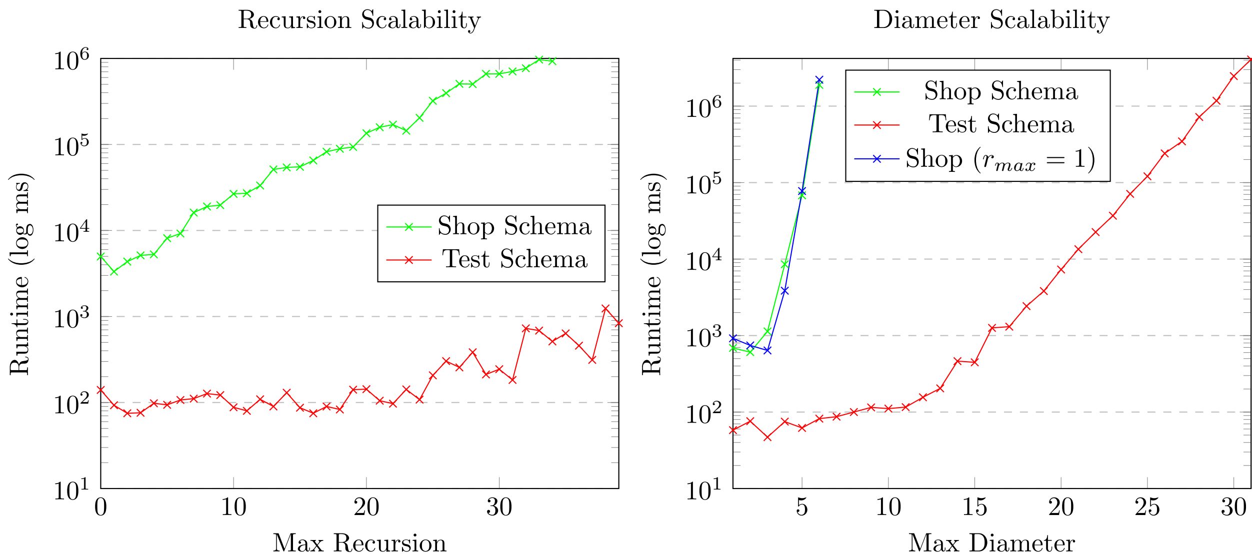

Figure 8: Runtime scaling when generating 100 queries.

6.2 Edge Graph Scaling

The most likely explanation for the increasing runtime is the construction time of the edge graph. Therefore some tests were executed to monitor the size of the edge graph. It should be noted that all edges in the edge graph are unique and application profiling suggests that ensuring this property remains correct takes up the majority of the runtime during query generation. For this reason all the graphs presented for this section list both the number of edges that were actually added to the edge graph ("Unique Edges") and the total number

[3] https://github.com/gbagan/gmark/tree/master/use-cases

of edges that was attempted to be added ("Edges"). In order to smooth out differences due to randomisation each data point represents the sum of 100 edge graphs constructed for the same query workload.

Figure 9 shows the size of the edge graph when varying the recursive depth for both graph schemas. It is worth noting that the edge graphs for the shop schema are significantly larger than the edge graphs for the test schema. In addition, it appears as if the growth rate of the edge graph for the test schema decreases as the recursive depth goes up. This effect is not visible for the shop schema, possibly because the prerequisite recursive depth was not tested.

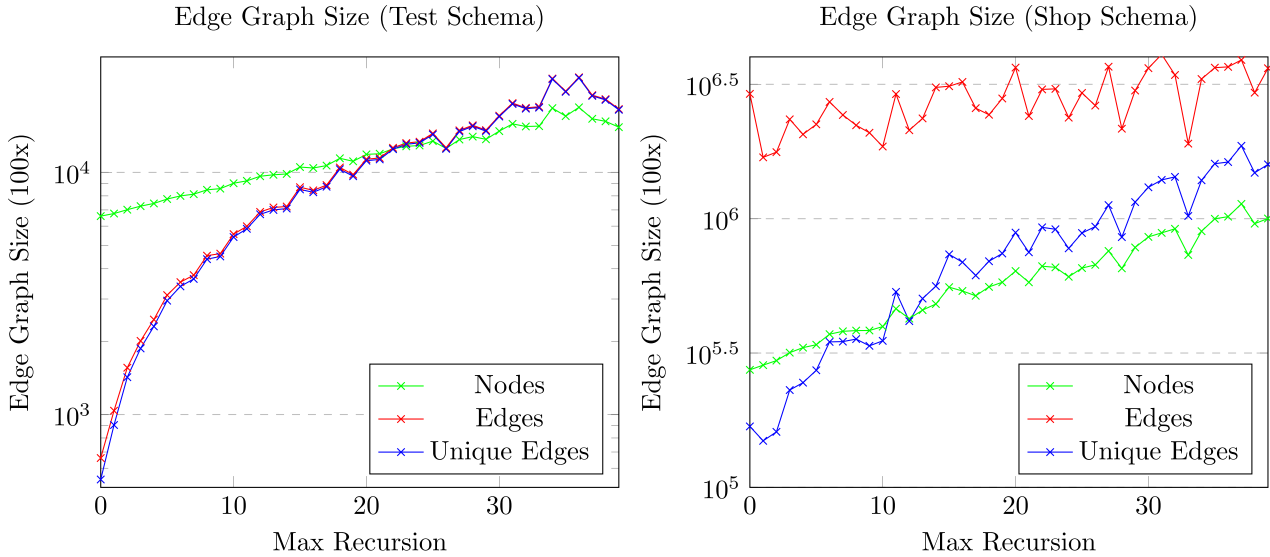

Figure 9: Edge Graph size scaling due to the maximum recursion.

Figure 10 shows the size of the edge graph when varying the maximum diameter. It is worth noting that due to the required runtime the shop schema could not be explored with as high a diameter as the test schema. In both graphs we can clearly see some saturation effect occurring. The total number of edges that is attempted to be added increases in an exponential fashion, however, there is a sharp decline in the number of edges that actually make it into the edge graph. There is therefore still considerable room for optimisation here to decrease the size of this gap.

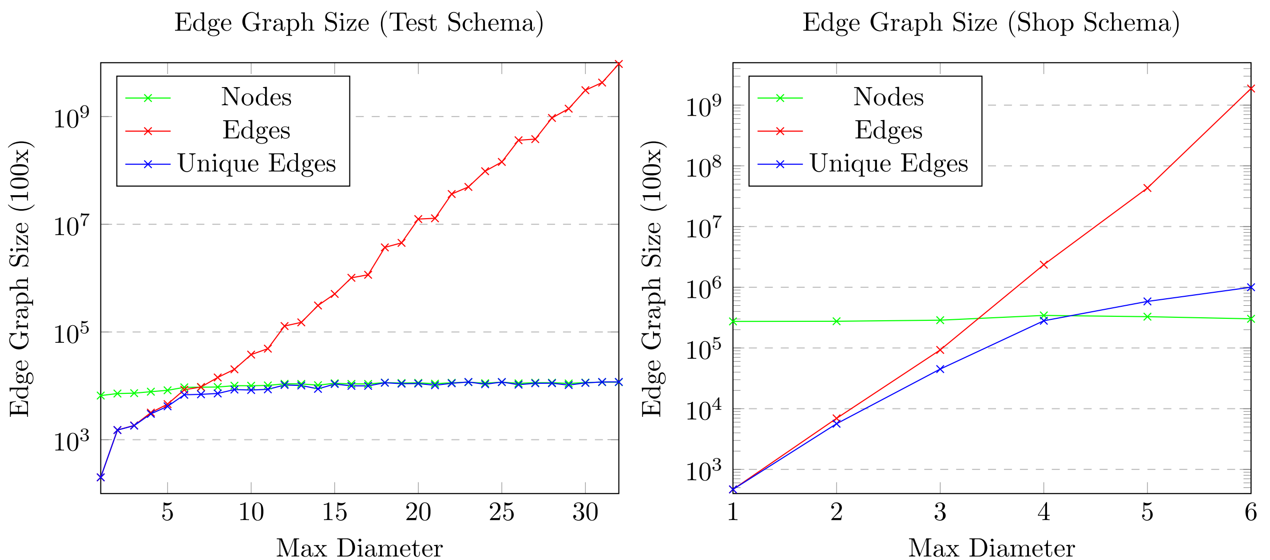

Figure 10: Edge Graph size scaling due to the maximum diameter.

7 Concluding Remarks & Limitations

To summarise, in this report we have presented a method for generating CPQs for benchmarking purposes. From the performance evaluation we can conclude that the method is sufficiently scalable for most use cases with the key limiting factor being the CPQ diameter. With the current implementation large graph schemas can only be explored with a maximum diameter of 6, going any higher results in prohibitively long runtimes. For recursion the practical limit is around 32, however slightly more is still possible with a large but still reasonable runtime.

7.1 Future Work

In the following sections we will discuss some possible directions for future work while also highlighting some limitations and properties of the current approach.

7.1.1 Runtime Improvements

Further investigation revealed that query generation performance is dominated primarily by the construction of the edge graph. The most important factors influencing the construction time are the number of schema edges, maximum path length and recursive depth. Currently the construction step computes all possible paths from all nodes. It would be possible to greatly speed up the construction step by instead only computing a random sample. Essentially we are moving the randomness of selecting the final path through the graph partially to the construction stage this way and this would not decrease the number of possible final paths that can be drawn. This is a possible optimisation related direction for future work.

An implementation related factor heavily influencing the runtime is the uniqueness check for edge graph operations. Instead of reducing the number of edges that are added to the graph, optimising this uniqueness check to be more efficient would also lead to a better runtime without the need to change the main algorithm. Options here are however more limited and unlikely to have a major impact given the exponential nature of the problem.

7.1.2 Theory Improvements

Next are some future work suggestions related to the theory and methods used to build up the edge graph.

Firstly, currently intersection nodes in the edge graph are generated from two paths that run parallel between the same source and target node. As established in Section 3.1 it is possible to take the conjunction with a path of a higher selectivity than desired and still have the result get the desired selectivity. This means that it should be possible to draw a second path from the schema graph starting at the corresponding edge graph source node and to a target node of the same type but with a higher selectivity. The exact implications this has on the generated conjunction should be investigated, most notably, it should be checked that the resulting source target pair sets are not disjoint. For a randomly generated graph this is likely not an issue, but it might be for a graph with more structure.

Secondly, currently intersections with identity are generated by scanning a path from the source to a specific edge graph node. If a node is found that matches the type of the end node of the path and the selectivity is at most linear, then an intersection with identity is added. However, currently this is always done for the first compatible node found along a path, whereas there may be other nodes later along the path that could also produce a valid intersection with identity.

Finally, a feature that would be nice to extend the algorithm with is the ability to generate CPQs with a given minimum diameter. The most obvious place to handle this bound would be when drawing paths in the edge graph. This however gives rise to a new issue where a path of the required minimum length may not exist or only be present in a very small quantity. The first issue could be solved by bounding the requested minimum length to be at most the length of the longest paths present in the edge graph. The second issue presents a potential performance issue that would need to be investigated.

7.1.3 Technical Improvements

Since query generation is highly parallelisable it would be possible to improve performance by adding support for multi-threaded query generation. It is also worth noting that there are cases in which generating a query fails. The primary reasons for this are the failure to draw a path from the selectivity graph and the absence of any paths between the nodes selected from the selectivity graph. Both issues are not that common, but are more often seen on very restrictive workload configurations (e.g., a maximum diameter of 1).

References

[1] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gMark: Schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, 29(4):856–869, 2017.

[2] Yuya Sasaki, George Fletcher, and Makoto Onizuka. Language-aware indexing for conjunctive path queries. *38th IEEE International Conference on Data Engineering (ICDE)*, 9-12 May 2022. (Virtual) Kuala Lumpur, Malaysia. IEEE Computer Society, in press.

A Getting started with gMark

This appendix will detail how to use and obtain a copy of the rewritten gMark software. The software is released on GitHub[4] and the README file of the repository will always contain the most up-to-date instructions on using the software. This appendix is written for release 1.0 of gMark and may not be fully correct for newer releases of gMark. To support a wide variety of of use cases gMark is a available in a number of different formats:

- As a standalone executable with both a graphical and command line interface
- As a docker image
- As a maven artifact
- For development

Command line usage

When using gMark on the command line the following arguments are supported:

```
usage: gmark [-c <file>] [-f] [-g <size>] [-h] [-o <folder>] [-s <syntax>] [-w <file>]
 -c,--config <file>    The workload and graph configuration file
 -f,--force            Overwrite existing files if present
 -g,--graph <size>     Triggers graph generation, a graph size can be provided (overrides the ones
                       set in the configuration file)
 -h,--help             Prints this help text
 -o,--output <folder> The folder to write the generated output to
 -s,--syntax <syntax> The concrete syntax(es) to output
 -w,--workload <file> Triggers workload generation, a previously generated input workload can
                       optionally be provided to generate concrete syntaxes for instead
```

For example, a workload of queries in SQL format can be generated using:

```
gmark -c config.xml -o ./output -s sql -w
```

An example configuration XML file can be found both in the gMark rewrite repository[5] and in the graphical interface of the standalone executable. The example RPQ workload configuration files included in the original gMark repository are also compatible and can be found in the use-cases folder[6].

Executable download

gMark is available as a standalone portable executable that has both a graphical interface and a command line interface. The graphical interface will only be launched when no command line arguments are passed. This version of gMark requires Java 8 or higher to run. Pre-compiled binaries of this version of gMark for all major operating systems can be found in the repository README and releases section[7]. The following commands show how to generate a workload of queries in SQL format on the command line using the standalone executable:

Windows executable: `./gMark.exe -c config.xml -o ./output -s sql -w`
Runnable Java archive: `java -jar gMark.jar -c config.xml -o ./output -s sql -w`

[4] https://github.com/RoanH/gMark
[5] https://github.com/RoanH/gMark/blob/master/gMark/client/example.xml
[6] https://github.com/gbagan/gmark/tree/master/use-cases
[7] https://github.com/RoanH/gMark/releases

Docker image

gMark is available as a docker image[8] on Docker Hub. This means that you can obtain the image using the following command:

```
docker pull roanh/gmark:latest
```

Using the image then works much the same as the regular command line version of gMark. For example, we can generate the example workload of queries in SQL format using the following command:

```
docker run --rm -v "$PWD/data:/data" roanh/gmark:latest -c /data/config.xml -o /data/queries -s sql -w
```

Note that we mount a local folder called 'data' into the container to pass our configuration file and to retrieve the generated queries.

Maven artifact

gMark is available on maven central as an artifact[9], so it can be included directly in another Java project using build tools like Gradle and Maven. This way it becomes possible to directly use all the implemented constructs and utilities. A hosted version of the javadoc for gMark can be found at gmark.docs.roanh.dev[10].

Gradle

```
repositories{
    mavenCentral()
}

dependencies{
    implementation 'dev.roanh.gmark:gmark:1.0'
}
```

Maven

```
<dependency>
    <groupId>dev.roanh.gmark</groupId>
    <artifactId>gmark</artifactId>
    <version>1.0</version>
</dependency>
```

Development of gMark

The gMark repository contains an Eclipse[11] and a Gradle[12] project with Util[13] and Apache Commons CLI[14] as the only dependencies. Development work can be done using the Eclipse IDE or using any other Gradle compatible IDE. Unit testing is employed to test core functionality. Continuous integration is used to run checks on the source files, check for regressions using the unit tests, and to generate release publications. Compiling the runnable Java archive (JAR) release of gMark using Gradle can be done using the following command in the gMark directory:

```
./gradlew clientJar
```

After which the generated JAR can be found in the `build/libs` directory. On windows `./gradlew.bat` should be used instead of `./gradlew`.

[8] https://hub.docker.com/r/roanh/gmark
[9] https://mvnrepository.com/artifact/dev.roanh.gmark/gmark
[10] https://gmark.docs.roanh.dev/
[11] https://www.eclipse.org/
[12] https://gradle.org/
[13] https://github.com/RoanH/Util
[14] https://commons.apache.org/proper/commons-cli/introduction.html