Subgraph Querying with Parallel Use of Query Rewritings and Alternative Algorithms

Foteini Katsarou
School of Computing Science
University of Glasgow, UK
f.katsarou.1@research.gla.ac.uk

Nikos Ntarmos
School of Computing Science
University of Glasgow, UK
nikos.ntarmos@glasgow.ac.uk

Peter Triantafillou
School of Computing Science
University of Glasgow, UK
peter.triantafillou@glasgow.ac.uk

ABSTRACT

Subgraph queries are central to graph analytics and graph DBs. We analyze this problem and present key novel discoveries and observations on the nature of the problem which hold across query sizes, datasets, and top-performing algorithms. Firstly, we show that algorithms (for both the decision and matching versions of the problem) suffer from straggler queries, which dominate query workload times. As related research caps query times not reporting results for queries exceeding the cap, this can lead to erroneous conclusions of the methods' relative performance. Secondly, we study and show the dramatic effect that isomorphic graph queries can have on query times. Thirdly, we show that for each query, isomorphic queries based on proposed query rewritings can introduce large performance benefits. Fourthly, that straggler queries are largely algorithm-specific: many challenging queries to one algorithm can be executed efficiently by another. Finally, the above discoveries naturally lead to the derivation of a novel framework for subgraph query processing. The central idea is to employ parallelism in a novel way, whereby parallel matching/decision attempts are initiated, each using a query rewriting and/or an alternate algorithm. The framework is shown to be highly beneficial across algorithms and datasets.

CCS Concepts

•Information systems → Database query processing;
•Mathematics of computing → *Graph algorithms;*

Keywords

Graph databases, graph query processing, subgraph isomorphism

1. INTRODUCTION

Graphs are ideal for representing complex entities and their relationships/interactions and subgraph querying is essential to graph analytics. In *subgraph querying*, given a pattern graph (query) and a graph DB, we want to know whether it is contained in each DB graph (the decision problem) and/or find all its occurrences within it (the matching problem). Subgraph querying entails the subgraph isomorphism problem (abbreviated as sub-iso), which is NP-complete. Subgraph querying has received a lot of attention. Related work is categorized in two major categories: the *filter-then-verify (FTV)* and the *no-filter, verify (NFV)* methods. Numerous methods have been proposed for the problem and three recent experimental analysis papers ([7, 9, 12]) compare and stress-test proposed methods.

In this work, we conduct a comprehensive analysis of this problem. Our analysis aims to (i) lead to interesting novel findings about the nature of the problem and existing solutions, (ii) analyse and quantify said discoveries and their effect on well-established existing solutions, and (iii) show that the findings can be used to develop a framework that can offer large performance gains. Specifically, we first recognize the existence of "straggler" queries; i.e., queries whose execution time is dramatically higher than the rest. This holds for all query workloads and all datasets examined and across all tested FTV and NFV algorithms. Subsequently, we reveal and quantify the interesting fact that isomorphic instances of queries can have a wild variation in querying times. Then we generate isomorphic instances of the original query using statistics on vertex-label frequencies and/or vertex degrees and we investigate their performance. Moreover, for NFV methods in particular, we additionally show that challenging queries are algorithm-specific, with a straggler query for one algorithm possibly being easy for others. Finally, we incorporate these findings in a novel framework, coined the Ψ-framework, that exploits parallelism for both FTV and NFV methods, achieving large performance gains. Specifically, instead of trying to come up with new algorithms for sub-iso testing, we utilize isomorphic query rewritings and existing alternative algorithms in parallel. Extensive experimentation shows that our framework can be highly beneficial across datasets and workloads, and for both FTV and NFV methods.

2. BACKGROUND

2.1 Related work

Related work is categorized in two major categories. In the first category, proposed methods typically address a *decision* problem, where given a dataset of many (typically small) graphs and a query/pattern graph q, the method decides whether q is contained in any graph in the dataset.

Most of the so-called *filter-then-verify (FTV)* or *indexed subgraph query processing* methods solve this decision problem, and work in 2 stages. In the index construction phase, stored graphs are decomposed into features which are then indexed, along with graph-id lists; i.e., lists of graphs that contain the feature. During query processing, query graphs are similarly decomposed into features; graphs from the dataset that do not contain one or more of these features definitely do not contain the query and are thus pruned away. The remaining graphs form the *candidate set*. At the verification stage, the query graph is tested for subgraph isomorphism against each graph in the candidate set to produce the final answer. The target of all these methods is to prune the candidate set and thus to reduce the number of sub-iso tests performed. Related works can be classified along 4 major dimensions: (i) type of indexed features (where "feature" refers to substructures of indexed graphs used to produce the index, independently of whether these are actually stored in the index or not): paths [1, 5, 30], trees [15, 25], simple cycles, or graphs [3, 20, 21, 22, 24, 29]; (ii) approach for extracting said features from indexed graphs: i.e., exhaustive enumeration [1, 10, 20, 30] or frequent subgraph mining techniques [3, 21, 22, 24, 25, 29]; (iii) index data structure: hash table, tree, trie; and (iv) whether the index stores location information or not. FTV methods are extensively discussed in [7, 9]. In [9] we concluded that Grapes[5] and GGSX[1] are the best solutions in terms of index construction and query processing time, and scalability limitations.

In the second category, proposed methods address a *matching* problem, whereby sub-iso testing is performed to find *all* the embeddings of the query graph q in a given large, stored graph g without performing any graph filtering in advance. We will call them the *no-filter, verify (NFV) methods.* Proposed methods, apart from the sub-iso test, additionally comprise of a pre-processing step where they maintain a feature-based index consisting of: (i) vertices and edges [15, 18], (ii) shortest paths [28] or (iii) subgraphs [8, 26] up to a certain size. The algorithms store vertex label lists along with additional information to facilitate the sub-iso test. A number of such methods were presented and compared in [12], concluding that (i) although there was no single algorithm to outperform all others in all occasions, GraphQL[8] was the only one that managed to complete all the tested query workloads; (ii) all three of GraphQL, sPath[28] and QuickSI[15] showed very good performance; but also that (iii) all existing algorithms have weaknesses in the way they apply their join selection and pruning heuristics, leading to the need for new graph matching algorithms.

There is nothing obstructing the NFV methods being applied for the decision problem and the FTV methods for the matching problem. FTV methods were originally proposed to work with datasets consisting of numerous, relatively small graphs, and their effectiveness relies on their achieved filtering, whereas NFV methods construct an index primarily to locate candidate vertices of the query in a large stored graph. For the current work, we opt to utilize all proposed methods for the originally proposed problems.

TwinTwig[11] and sTwig[16] deal with very large graphs, stored in a distributed infrastructure, and rely on parallel computing to perform sub-iso testing. Within FTV methods, iGQ[19] is a recent approach that employs caching on top of any proposed FTV method to improve performance. Semertzidis et al. [14] considered pattern queries over time-evolving graphs, which are beyond the scope of this study. Finally, there has been considerable work on the subject of *approximate graph pattern matching*. Related techniques (e.g. [10, 17, 20, 23, 27]) perform subgraph matching, but with the support for wildcards and/or approximate matches. All of these algorithms are not directly related to our work as we focus on exact subgraph matching.

As subgraph querying is an important problem, we expect that many researchers will keep focusing on trying to improve upon existing algorithms in the future. Indeed, since the publication just a few years ago of [12], comprehensively comparing the then state of art, newer algorithms have been proposed [6] with better performance. Nonetheless all algorithms show exponential execution times even at small query sizes (up to 10 edges)[13]. Our contributions aim to help this process in two ways. First, by revealing key insights, based on comprehensive experimentation, about the problem itself and how they affect well-known algorithms. Second, by shedding light onto a novel overall approach to the problem and its benefits. Namely, instead of focusing solely on developing new solutions by improving earlier algorithms, try to benefit from the wealth of ideas already existing within previous algorithms! Specifically, our findings show that different algorithms are appropriate for different queries. Furthermore, they show that different query rewritings are appropriate for different queries and for different algorithms! Finally, the existence of straggler queries poses new challenges for the performance comparison of different algorithms, needing more detailed performance metrics and experimenting with more challenging queries. All current works miss the above points: (i) they only consider one query rewriting, if at all, for all queries, (ii) they use only one algorithm for all workload queries, and (iii) they do not stress-test their algorithms with more challenging queries (e.g., larger sizes). Our framework shows that such misses also lead to misses of dramatic performance improvements.

2.2 Definitions

Definition 1 (Graph). *A graph $G = (V, E, L)$ is defined as the triplet consisting of the set $V = \{v_i\}, i = 1, ..., n$ of vertices of the graph, the set $E \subseteq \{(v, u) : v, u \in V\}$ of edges between vertices in the graph, and a function $L : V|E \rightarrow \mathcal{L}$ assigning a label $l \in \mathcal{L}$ ($\mathcal{L}$ being the set of all possible labels) to each vertex $v \in V$ and each edge $e \in E$.*

We assume that each node in a graph is assigned an integer in the interval $[1, n]$, so that no two nodes in a graph have the same number; we call this the node ID.

Definition 2 (Graph Isomorphism). *Two graphs $G = (V, E, L)$ and $G' = (V', E', L')$ are isomorphic iff there exists a bijection $I : V \rightarrow V'$ that maps each vertex of G to a vertex of G', such that if $(u, v) \in E$ then $(I(u), I(v)) \in E'$, $L(u) = L'(I(u))$, $L(v) = L'(I(v))$, and vice versa.*

Note that, given a graph G, a graph G' isomorphic to G can be trivially produced by permuting the node IDs in G.

Definition 3 (Subgraph Isomorphism). *A graph $G = (V, E, L)$ is subgraph isomorphic to a graph $G' = (V', E', L')$, denoted by $G \subseteq G'$, iff there exists an injective function $I : V \rightarrow V'$ such that if $(u, v) \in E$ then $(I(u), I(v)) \in E'$ and $L(u) = L'(I(u))$ and $L(v) = L'(I(v))$. Graph G is then called a subgraph of G'.*

Much like all of the works mentioned earlier, we focus on non-induced subgraph isomorphism.

3. EXPERIMENTAL SETUP

3.1 Short description of used Algorithms

3.1.1 FTV methods

Both Grapes[5] and GGSX[1] index the simplest form of features – i.e., paths – up to a maximum length. Paths are searched in a DFS manner and indexed in a trie or suffix tree respectively. Compared to GGSX, Grapes takes an additional step and maintains location information. Also, Grapes features multi-threaded design for both indexing and query processing. In query processing, maximal paths of the query are extracted to form the query index which is matched with the dataset index, pruning away unmatched branches. Subsequently, the search space is further pruned by the frequencies of indexed features. After this step, GGSX forms its candidate set of graphs that will undergo sub-iso testing. Grapes further exploits the maintained location information to extract relevant connected components of the dataset graphs, against which sub-iso testing is performed.

The underlying isomorphism algorithm for both Grapes and GGSX is VF2[4]. VF2 does not define any order in which query vertices are selected. Given a query graph q and a dataset graph g, the algorithm chooses a vertex from q to match to vertices in g, and proceeds by then trying to match still unmatched vertices adjacent to the matched ones in q. Given an unmatched vertex in q, the set of candidate vertices of g is defined as the set of all vertices in g with the same label as the unmatched vertex in q. VF2 then employs 3 pruning rules to reduce the number of candidate vertices. The first rule removes candidates that are not directly connected to the already matched vertices of g. The second rule removes all candidates for which the number of adjacent unmatched nodes which are also adjacent to matched nodes of g, is smaller than the corresponding figure for the matched vertex of q. The final rule removes all g candidates with less adjacent (matched/candidate) nodes than the corresponding figure in q.

3.1.2 NFV methods

In the sub-iso test of QuickSI[15] (QSI for short), priority is given to the vertices with infrequent labels and infrequent adjacent edge labels. In the indexing phase, QuickSI precomputes the frequencies of labels and edges and uses them to compute the "average inner support" of a vertex or an edge; i.e., the average number of possible mappings of the vertex or edge in the graph. The inner support is later used in the graph matching process to assign weights on the edges of the query graph and construct a rooted minimum spanning tree (MST). In case of symmetries, edges are added in such a way that will make the MST denser. The order in which vertices are inserted to the MST defines the order in which they are then matched in the sub-iso test.

In the indexing phase of GraphQL[8] (GQL for short), the labels of all vertices along with the neighbourhood signatures, which capture the labels of neighbouring nodes in a radius i in lexicographical order, are indexed. In the subgraph matching phase, the algorithm starts by retrieving all possible matches for each node in the pattern. Subsequently, 3 rules are applied in order to prune the search space. First, the indexed vertex labels and neighbourhood signatures are used to infeasible matches. Then a pseudo subgraph isomorphism algorithm is applied to the problem iteratively up to level l; i.e., for every pair of possible graph-query vertex matches, the nodes adjacent to the query node should be matched to the corresponding neighbours of the graph. Finally, the algorithm needs to optimize the search order in the query before proceeding with the actual sub-iso test, which in turn consists of a number of *joins* of the candidate node lists. This optimization is based on an estimation of the result-set size of intermediate joins, and as it would be very expensive to enumerate all possible search orders, only left-deep query plans are considered.

sPath[28] (SPA for short), similarly to GraphQL, also maintains a neighbourhood signature comprised of shortest paths organized in a compact indexing structure. Specifically, in order to reduce the storing space, shortest paths are not really maintained, but they are decomposed in a distance-wise structure. In the query processing, the query is initially decomposed in shortest paths that are then matched to the candidate shortest paths from the stored graph. From all possible candidate shortest paths, those that (i) can cover the query and (ii) provide good selectivity, i.e. minimize the estimated result-set size of each join operation, are selected as candidates. For each one of the selected paths, an edge-by-edge verification is then used to perform the sub-iso test.

3.2 Setup

Experiments with Grapes and GGSX were conducted on a small cluster consisting of 5 nodes, each featuring an Intel Core i5-3570 CPU (3.4GHz, 4 physical cores, 6MB cache), 16GB of RAM, 500GB disk per node, and running Ubuntu Linux 14.04. Experiments with QuickSI, GraphQL and sPath (i.e., the NFV methods) were conducted on a Windows 7 SP1 host, with 2 Intel Xeon E5-2660 CPUs (2.20GHz, 20MB cache) with 8 cores/16 vcores per CPU, 128GB of RAM, and 3.5TB disk. For practical purposes, we allowed a maximum limit of 10 mins for each query to be processed. Beyond that time, the execution is terminated and we proceed with the next query in the workload. Please note that this 10' limit does not apply in the indexing phases of the algorithms.

For Grapes and GGSX we used the implementations provided by their respective authors. However, in the case of Grapes, we had to alter the source code so that the VF2 verification step returns after the first match of the query graph, as opposed to the original implementation which was returning all possible matches. The reason for this is that FTV methods are mainly designed to retrieve the graphs that contain the query as an answer. For QuickSI, GraphQL and sPath, we used the implementation provided by [12].

We used the default values for the input parameters of the compared algorithms, as they were defined by their respective authors in the relevant publications and/or in their implementation code. More specifically:

- For GGSX and Grapes, we enumerated paths of up to size of 4.
- We ran Grapes with 1 and 4 threads; results for executions with 1 (resp. 4) threads are denoted by Grapes/1 (resp. Grapes/4).
- For GraphQL, we used a refined level of iterations of pseudo-subgraph isomorphism $r = 4$.
- For sPath, we used a neighbourhood radius of 4 and maximum path length 4.

		PPI	Synthetic
Dataset	# graphs	20	1000
	#disconnected graphs	20	0
	#labels	46	20
Per Graph	Avg #nodes	4942	1100
	StdDev #nodes	2648	483
	Avg #edges	26667	12487
	Avg density	0.0022	0.020
	Avg degree	10.87	24.5
	Avg #labels	28.5	20

Table 1: Dataset characteristics for FTV methods

	yeast	human	wordnet
#nodes	3112	4674	82670
#edges	12519	86282	120399
Avg degree	8.04	36.91	2.912
StdDev degree #nodes	14.50	54.16	7.74
Density	0.00258	0.0079	0.000035
#labels	184	90	5
Avg frequency labels	127	240	16534
StdDev frequency labels	322.5	430	152

Table 2: Dataset characteristics for NFV methods

- For QuickSI, GraphQL and sPath the number of searched embeddings of the pattern graph on the stored graph is capped at 1000; i.e., after finding the first 1000 matches, the algorithms terminate.

3.3 Datasets

We have chosen datasets which (a) have also been used by other studies, so as to enable possible direct comparisons, and (b) have key characteristics covering a large part of the design space (e.g., regarding graph size and density).

Table 1 summarizes the characteristics of the datasets that we used for the FTV methods. PPI (used in [5, 9]) is a real dataset representing 20 different protein-protein interaction networks. The majority of existing real datasets that were used for the FTV methods comprise of relatively small and sparse graphs. In [9] we showed that, for such datasets, both Grapes and GGSX perform adequately well. For our current study we are further interested in more challenging datasets and we thus employ an additional synthetic dataset generated with GraphGen[2], allowing various parameters of interest to be specified; namely, number of graphs, average number of nodes and density per graph, number of labels in the dataset). A more detailed description of how GraphGen constructs the dataset can be found in [9].

Datasets used for the NFV methods consist of only one graph as the primary task of these methods is to find all occurrences of the pattern graph in the large stored graph. Table 2 summarizes the characteristics of the three real datasets – namely yeast, human and wordnet — that we have used for the NFV methods. Yeast and human were previously used in [12], while Wordnet[1] was used in [16].

3.4 Query Workloads

To generate each of the queries, first we select a graph from the dataset uniformly and at random, and from that graph we select a node uniformly and at random. Starting from said node, we generate a query graph by incrementally adding edges chosen uniformly at random from the set of all edges adjacent to the resulting query graph, until it reaches the desired size. For the synthetic dataset, we used 100 queries of size 24, 32 and 40 edges for Grapes/1 and Grapes/4. We did not run GGSX against the synthetic dataset, because of excessive amount of time required for the experiments to complete. For the PPI dataset, we used 100 queries of size 16, 20, 24, and 32 edges. For the NFV methods, we used 200 queries of 10, 16, 20, 24 and 32 edges. Last, for QuickSI we only report results against the yeast dataset, as (i) it was the easiest NFV dataset to process, and (ii) QuickSI always had many more cases, compared to GraphQL and sPath, where query processing exceeded the 10' cap. For all used methods, the majority of the queries completed in under 2". We call them *easy* queries. Another portion of queries had processing times in the 2" to 600" range; we denote these *2"-600"* queries. We use the term *completed* to refer to all queries that finished within the 10' limit; those that did not are called *hard* or *killed.*

3.5 Performance Metrics

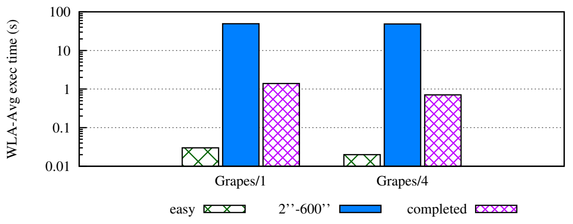
For every query against a stored graph, we measure the *Execution Time*, denoted *exec time*, for both FTV and NFV methods, while *avg exec time* denotes the average execution time. Specifically for FTV methods, this is the pure sub-iso time; i.e., excluding the index loading and filtering times, which add only a trivial overhead. For FTV methods reported times are in seconds, while for NFV methods times are in milliseconds, unless stated otherwise.

Let q_i be a given query and t_i^M the exec time of q_i over method M. Let also $q_{i,j}$ be the j-th isomorphic instance of q_i and $t_{i,j}^M$ the exec time of $q_{i,j}$ over method M. Finally, let $t_{i,j}^\Psi$ be the exec time of $q_{i,j}$ over our proposed Ψ-framework. We define the (*max/min*) metric as: $\frac{\max_j(t_{i,j}^M)}{\min_j(t_{i,j}^M)}$. The minimum value of this metric is 1, indicating that there are no variations between the min and max exec time. The higher the value of this metric, the higher the differences between the min and max exec time achieved by the isomorphic query instances. We also define the *speedup** metric as: $\frac{t_i^M}{T}$, where T is set to: (i) $\min_j(t_{i,j}^M)$, when comparing against the various isomorphic instances of q_i, (ii) $\min_M(t_{i,j}^M)$, when comparing against different methods, and (iii) t_i^Ψ, when comparing against our Ψ-framework. *speedup** represents what we lose in performance if we choose the original method over the various alternatives; i.e., *speedup** equals the maximum attainable speedup over the original method, if we chose the best of the examined alternatives. For comparison purposes, for queries that were killed at the 10' limit we use this time (i.e., 600") as their minimum execution time.

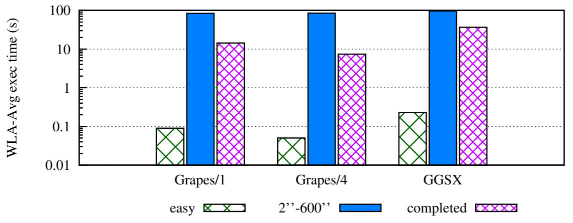
When comparing two sets of measurements $A = \{A_i\}$ and $B = \{B_i\}$, we can compute their average ratio in two ways:

- *Workload-Level Aggregation (**WLA**)*, given by $\frac{avg_i(B_i)}{avg_i(A_i)}$. When A and B contain query response times, the WLA computation would give the improvement in the overall average execution time. This metric is important from the *system* perspective as it encapsulates the overall performance change.
- *Query-Level Average (**QLA**)*, computed as $avg_i\left(\frac{B_i}{A_i}\right)$. When applied to query processing times, the QLA

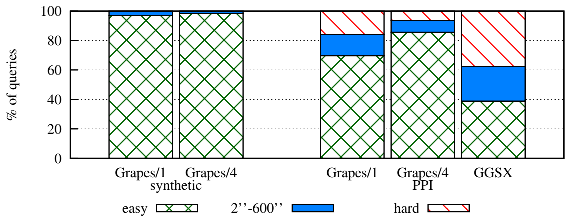
[1]http://vlado.fmf.uni-lj.si/pub/networks/data/dic/Wordnet/Wordnet.htm



(a) Synthetic dataset, WLA-Avg exec time (s)

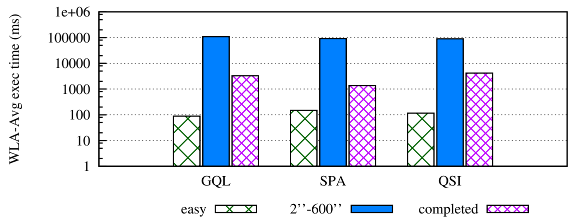


(b) PPI dataset, WLA-Avg exec time (s)

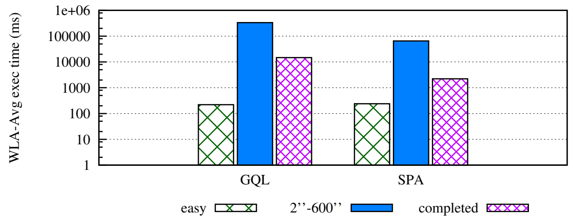


(c) Percentages of *easy*, *2"-600"*, and *hard* queries

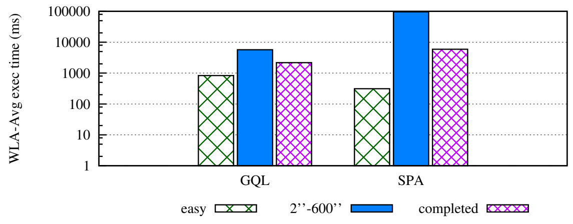
Figure 1: Stragglers in FTV methods



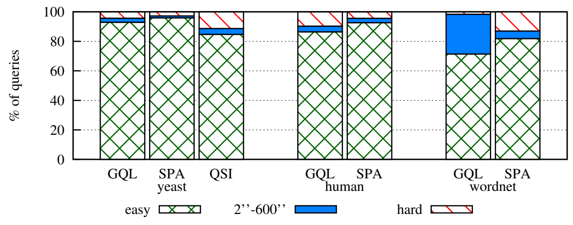
(a) yeast dataset, WLA-Avg exec time (ms)



(b) human dataset, WLA-Avg exec time (ms)



(c) wordnet dataset, WLA-Avg exec time (ms)



(d) Percentages of *easy*, *2"-600"*, and *hard* queries

Figure 2: Stragglers in NFV methods

computation would give the average of per-query improvements. This metric is *user-centric* in the sense that each user cares what the performance improvement for his query is using different methods.

In both cases, $avg_i(X_i)$ is the average over all items X_i in the set X. Based on this distinction, the aforementioned (max/min) and $speedup^*$ metrics can have a QLA or WLA version, denoted with a matching subscript; e.g., $speedup^*_{QLA}$. These two variants also carry over to other computations; for example, the standard deviation of the ratio of A and B would be computed as $\frac{stdDev_i(B_i)}{stdDev_i(A_i)}$ under WLA, and as $stdDev_i(B_i/A_i)$ under QLA. However, unless stated otherwise, we shall use QLA and WLA to denote averages.

4. STRAGGLER QUERIES

We know that as the dataset grows in terms of the size of graphs, query processing becomes harder; ditto as the size of the query graph increases [9]. But do these statements hold across all queries-dataset graph combinations? Running many queries against the whole dataset can hide the details of how much time is required per individual query-graph pair. In the case that a small portion of such pairs dominates the whole execution time, then by just looking at the whole query workload execution times it is easy to draw wrong conclusions about the algorithms' performance. Also, several related works choose to ignore queries whose execution is much higher compared to the rest. To investigate the above, in this study we execute each individual query against a single stored graph at a time.

Observation 1: In all of workloads generated by us or found in other papers, our experiments show "stragglers"; i.e., queries whose processing time is many orders of magnitude higher compared to the rest.

In order to back our observation, we present our results from the experiments on the aforementioned datasets against both FTV and NFV methods (fig. 1 and 2).

4.1 FTV methods

Fig. 1 presents the results from the query workloads on the FTV methods. Specifically, 1(a) and 1(b) show the average execution times for the corresponding algorithms for the synthetic and the PPI dataset respectively (GGSX/synthetic results omitted; see §3.4). 1(c) presents the percentage of the sub-iso tests that were *easy*, *2"-600"*, and *hard* for both the synthetic and PPI datasets. As expected, Grapes/4 has a much smaller percentage of *killed* queries compared to Grapes/1 and GGSX. A notable thing here is that although for both Grapes/1 and Grapes/4 the percentage of *2"-600"*

		GraphQL	sPath	QuickSI
10-edge q	AET *easy* (ms)	66.84	134.78	131.67
	% of *easy*	100	99.5	99
	AET *2"-600"* (ms)	-	2871.44	50367.40
	% of *2"-600"*	0	0.5	1
	% of *hard*	0	0	0
32-edge q	AET *easy* (ms)	130.66	120.71	96.62
	% of *easy*	80	91	67.5
	AET *2"-600"* (ms)	140812	140781	78917.2
	% of *2"-600"*	6.5	3	6
	% of *hard*	13.5	6	26.5

Table 3: Results for NFV methods on the yeast dataset (AET: avg exec time)

		GraphQL	sPath
10-edge q	AET *easy* (ms)	179.49	209.91
	% of *easy*	100	98
	AET *2"-600"* (ms)	-	182392
	% of *2"-600"*	0	1
	% of *hard*	0	0
32-edge q	AET *easy* (ms)	246.31	277.13
	% of *easy*	71.5	84.5
	AET *2"-600"* (ms)	93523.7	31817
	% of *2"-600"*	4.5	4.5
	% of *hard*	24	11

Table 4: Results for NFV methods on the human dataset (AET: avg exec time)

queries is $< 5\%$ in the synthetic dataset and $< 10\%$ in PPI, the avg exec time across all completed queries is significantly affected; that is, the most expensive queries dominate the execution time.

4.2 NFV methods

Fig. 2 presents the results from the query workloads on the NFV methods (QuickSI human/wordnet results omitted; see §3.4), while tables 3 and 4 give results for 10- and 32-edge queries for the yeast and human datasets. We can use the 10-edge query results to compare our findings with those presented in [12]. [12] used small query sizes (up to 10 edges) and showed that the best performing algorithm is GraphQL, because it managed to complete all tested query workloads. With our experiments, we confirm this for both the yeast and human datasets and for queries of size 10 edges. GraphQL performs better compared to sPath, having also 0% of *hard* queries. The same holds for the *easy* queries of 32 edges. However, the picture is reversed when looking at the rest of the queries. In this case, the percentage of *killed* queries is double for GraphQL compared to sPath.

We note that unlike yeast and human where sPath performs overall better than GraphQL having (i) smaller avg exec times on the completed queries and (ii) smaller percentages of *hard* queries, in wordnet this behavior is reversed. Based on our analysis, it's very difficult to claim that one algorithm is better than the other. In fact, in order to claim that, we need to define a performance metric of interest. Such a metric could be the percentage of *killed* queries, but note that it depends on the time limit imposed on query processing. For example, in wordnet, if the threshold was 2", then sPath would be better than GraphQL, but if we change this threshold the picture changes.

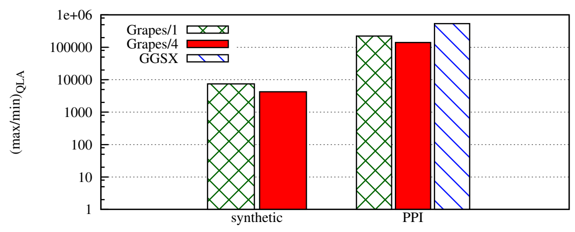


Figure 3: Avg $(max/min)_{QLA}$ for FTV methods

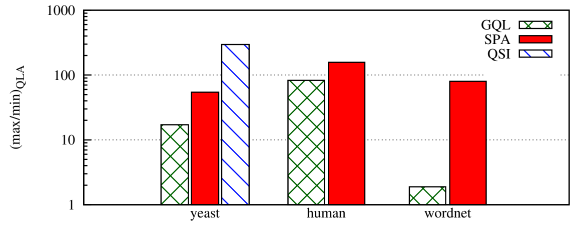


Figure 4: Avg $(max/min)_{QLA}$ for NFV methods

We summarize our results to the following 3 conclusions: (1) Some queries are *hard.* (2) Different algorithms have different percentages of *completed* queries; thus, different algorithms find different queries hard. (3) As the most expensive queries dominate the avg exec time, one must include a sufficient number of *hard* queries in order to draw conclusions about the relative performance of the algorithms.

5. ISOMORPHIC QUERIES

The proposed sub-iso methods ([8, 28, 15]), as well as [12] that compares them, claim that the search order on the query can have a huge impact on query processing time. We agree with this claim. In the current study, we take a further step and instead of relying on the order that the individual method imposes, we generated our own isomorphic query rewritings. To achieve this, we keep the structure of the query graph and the labels on the nodes unchanged, and permute the node IDs. Subsequently, we transform the query graph to an input format compatible with each individual method and perform the query processing. In the following experiments, we used a total of 6 different rewritings per query, leading to the following observation.

Observation 2: Queries which are isomorphic to the original query have widely and wildly different execution times.

We attribute this behavior to the fact that all proposed methods do not define a strict order in which the nodes of the query are matched, as computing a globally optimal join plan would be too computationally expensive. Thus, all methods rely on heuristics (see §3.1) in order to minimize the search space for the join plan.

5.1 FTV methods

Fig. 3 depicts the QLA average value of the (max/min) metric for the synthetic and PPI datasets, for the FTV methods (GGSX results omitted for the synthetic dataset; see §3.4). Table 5 additionally reports the stdDev, min, max and median values of $(max/min)_{QLA}$. In the calculations,

		Grapes/1	Grapes/4	GGSX
synthetic	stdDev	86,700.40	65,988.40	-
	min	1.06	1.02	-
	max	3,820,000.00	3,490,000.00	-
	median	3.90	4.45	-
PPI	stdDev	469,934	395,285	1,020,000
	min	1.03	1.02	1.01
	max	3,680,000	3,160,000	12,000,000
	median	1,186.51	11.19	109,086.00

Table 5: $(max/min)_{QLA}$ statistics for FTV methods

		GraphQL	sPath	QuickSI
yeast	stdDev	287.54	533.86	1685.71
	min	1.01	1.01	1.00
	max	7286.33	6695.85	15021.60
	median	1.40	1.36	1.61
human	stdDev	440.18	662.78	-
	min	1.00	1.04	-
	max	4115.06	4087.81	-
	median	1.82	1.96	-
wordnet	stdDev	20.55	396.87	-
	min	1.01	1.01	-
	max	646.44	3081.14	-
	median	1.21	1.34	-

Table 6: $(max/min)_{QLA}$ statistics for NFV methods

we did not include queries that were not helped by any of the isomorphic instances tried; i.e., queries that were *hard* on all tested isomorphic instances of the query. This behavior occurred in 0.0036% and 1.4% of queries for Grapes/1 on the synthetic and PPI datasets respectively, and 0.37% of queries for Grapes/4 and 1.96% of queries for GGSX for the PPI dataset. We note that the "max" and "average" values of $(max/min)_{QLA}$ are only lower-bound estimations, because of the 10' limit that we used instead of the actual verification time. In these results, we observe that there is an at least 6 orders of magnitude difference between the min and the max value of $(max/min)_{QLA}$, with the median (apart from GGSX) being closer to the min value. Along with the high stdDev, we can see that isomorphic instances of the same query can indeed have widely and wildly different verification times.

5.2 NFV methods

Fig. 4 reports the QLA-average values of the (max/min) metric for the yeast, human and wordnet datasets, for the tested NFV methods (QuickSI results omitted for the human and wordnet datasets; see §3.4). Table 6 reports the stdDev, min, max and median value of $(max/min)_{QLA}$. We report that 4.2%, 8.2% and 1.5% of queries were not helped by any tested isomorphic query instances for GraphQL and for yeast, human and wordnet respectively. For sPath the corresponding values are 2.1%, 1.4% and 11.8%. Finally, for QuickSI 8.6% of the queries were not helped for yeast.

The QLA-average (max/min) for the NFV methods is up to 3 orders of magnitude lower than that of the FTV methods. This is somewhat expected as the NFV methods define a more strict order in which the nodes of the query are matched and thus leave less space for wild variations. However, this order is still significantly affected by the initial node ids of the query, and thus we still see per-query (max/min) values of up to 2 orders of magnitude.

		Grapes/1	Grapes/4	GGSX
synthetic	stdDev	53,785.70	24,267.60	-
	min	1.00	1.00	-
	max	3,820,000	2,110,000	-
	median	1.36	1.24	-
PPI	stdDev	302,250	237,573	758,668
	min	1.00	1.00	1.00
	max	3,370,000	2,910,000	9,390,000
	median	3.71	1.67	1,751.22

Table 7: $speedup^{*}_{QLA}$ statistics for FTV methods across rewritings

		GraphQL	sPath	QuickSI
yeast	stdDev	235.61	422.56	1193.03
	min	1.00	1.00	1.00
	max	7286.33	6695.85	15021.60
	median	1.10	1.08	1.30
human	stdDev	259.93	492.45	-
	min	1.00	1.00	-
	max	4115.06	4087.81	-
	median	1.09	1.08	-
wordnet	stdDev	20.55	244.66	-
	min	1.00	1.00	-
	max	646.44	3081.14	-
	median	1.13	1.08	-

Table 8: $speedup^{*}_{QLA}$ statistics for NFV methods across rewritings

We summarize our overall results to the following conclusions: (1) For every isomorphic test to be executed, given a query graph q and a stored graph, there is an isomorphic version of q that can take anywhere from 2 to 6 orders of magnitude more time to execute compared to the least expensive version of the query. This holds across all algorithms and datasets tested. (2) Although the presented figures hide the details of the individual query sizes, we report that the harder the queries (higher query sizes), the higher these number are.

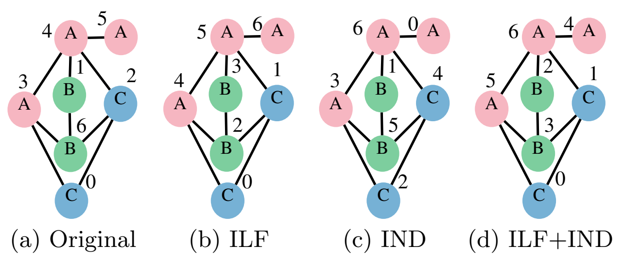
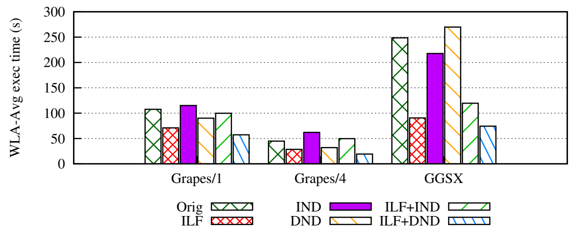


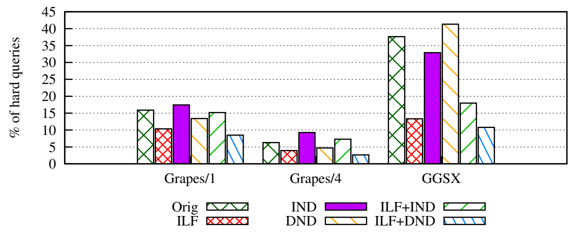
Figure 5: Isomorphic queries generated with different rewritings (assuming the label frequencies in the stored graph are: "A"=20, "B"=15, "C"=10)

6. GRAPH QUERY REWRITING

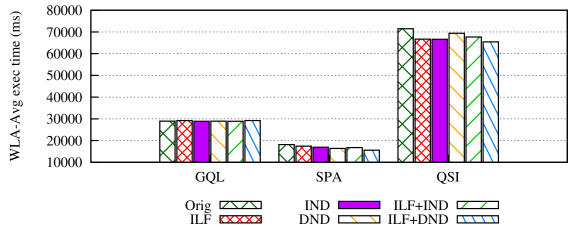
Having established that isomorphic versions of a query can have dramatically different execution times, we set out



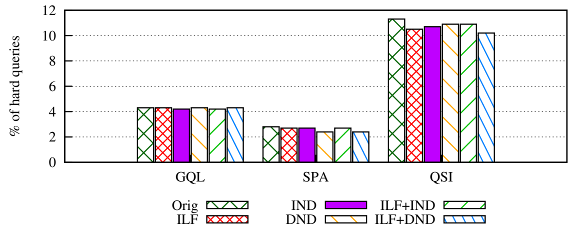
(a) PPI dataset, WLA-Avg exec time (s)



(b) PPI dataset, percentage of *hard* queries



(c) yeast dataset, WLA-Avg exec time (ms)



(d) yeast dataset, percentage of *hard* queries

Figure 6: Results for individual query rewrtings for both FTV and NFV methods

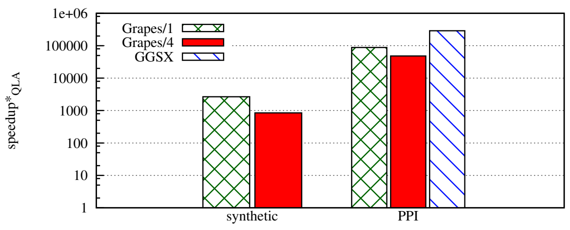


Figure 7: Avg $speedup^*_{QLA}$ **for FTV methods across rewritings**

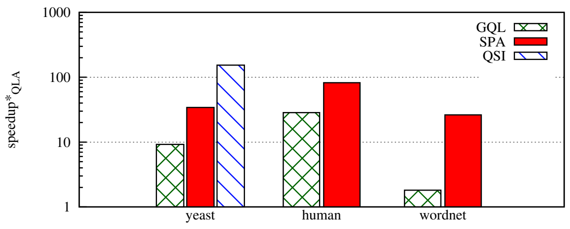


Figure 8: Avg $speedup^*_{QLA}$ **for NFV methods across rewritings**

to construct specific rewritings, constructing graphs isomorphic to the original queries, with the aim to capture these benefits. We have developed and experimented with several such query rewritings. We outline below five such rewritings, all performed by carefully permuting the node IDs in the query graph:

- `Query Rewriting ILF (`*`Increasing Label Frequency`*`)`: In a preprocessing step, we compute the frequencies of node labels in the stored graph, sorted in increasing frequency order. Given this order, we produce a rewriting of the query graph so if i, j are the node IDs of query graph nodes n_i, n_j, $L(n_i), L(n_j)$ are their labels, and $f(L(\cdot))$ is the frequency of a label $L(\cdot)$ in the stored graph, then $f(L(n_i)) < f(L(n_j)) \Rightarrow i < j$. Ties can appear in 2 cases: (i) two or more query nodes have the same label, or (ii) two or more query nodes have different labels but with the same frequency. These ties are broken arbitrarily.
- `Query Rewriting IND (`*`Increasing Node Degree`*`)`: The nodes of the query are sorted in increasing node degree order; i.e., if n_i, n_j are two query graph nodes, and $d(\cdot)$ is the degree (number of edges) of a node, then $d(n_i) < d(n_j) \Rightarrow i < j$. In the case of nodes with the same number of edges, ties are broken arbitrarily.
- `Query Rewriting DND (`*`Decreasing Node Degree`*`)`: This rewriting is similar to the IND but the nodes of the query are sorted in decreasing node degree and the nodes ids are assigned accordingly.
- `Query Rewriting ILF+IND`: This rewriting is the same as ILF above, with ties being broken in an IND manner: i.e., nodes with smaller outgoing degree get a lower node id.
- `Query Rewriting ILF+DND`: This rewriting is the same as ILF+IND, with ties being broken in a DND manner.

Fig. 5 presents an example of the above rewritings. Note that the ILF+IND rewriting in 5(d) is also a valid ILF rewriting. As we already mentioned, ties are (utterly) broken in an arbitrary way, and thus one may compute several different isomorphic graphs for the same rewriting.

Indicatively[2] and because of space restrictions, in fig. 6 we report the WLA average processing times of the original query and the 5 proposed query rewritings for the PPI and yeast datasets, as well as the corresponding percentages of the *hard* queries. For the FTV methods, the best performing rewritings are ILF and ILF+DND, with the percentage of *hard* queries being significantly improved. For the NFV methods, the picture is slightly different. GraphQL shows no considerable improvement with any individual rewriting; as a matter of fact, there are rewritings leading to higher

[2]We obtained similar results for the synthetic dataset for the FTV methods and the human dataset for the NFV methods. The sole exception was sPath, whose percentage of *hard* queries increased slightly for the wordnet dataset.

avg exec times than the original query. For sPath, the DND and ILF+DND rewritings reduced the percentage of *killed* queries from 2.8% to 2.4%. For QuickSI, ILF+DND reduced the percentage of *killed* queries from 11.3% to 10.2%, but DND only brought it down to 10.9%. More importantly, note that there is no single rewriting that manages to improve all algorithms across all datasets and workloads.

Observation 4: "Stragglers" can have isomorphic counterparts which are not stragglers.

Please note that the max and average reported *speedup** represent a lower-bound estimation because of the value 600" that we use for the *hard* queries that were killed. Additionally, in our calculations we do not include the few queries that were killed for both the original instance and with all the rewritings of the query (see §5.1 and §5.2).

6.1 FTV methods

Fig. 7 presents the average *speedup*$^*_{QLA}$ for the FTV methods for the synthetic and PPI datasets (GGSX/synthetic results omitted; see §3.4). Additionally, table 7 reports the QLA stdDev, min, max and median of *speedup*$^*_{QLA}$. Moreover, as we increased the size of the queries, *speedup*$^*_{QLA}$ increased by up to 3 orders of magnitude (not visible in the figure as results are aggregated). For the presented results, median *speedup*$^*_{QLA}$ is close to min *speedup*$^*_{QLA}$, evidencing again a wide variation in the benefits of the isomorphic query rewritings. Keeping in mind that the majority of the queries are *easy* (fig. 1), we conclude that large performance gains can come from improving the *hard* queries.

6.2 NFV methods

Fig. 8 presents the average *speedup*$^*_{QLA}$ for the NFV methods for the yeast, human and wordnet datasets (QuickSI human/wordnet results omitted; see §3.4). Table 8 reports the stdDev, min, max and median of the *speedup*$^*_{QLA}$. The performance of sPath could seemingly be improved by one to two orders of magnitude across all datasets. The same holds for QuickSI on yeast. GraphQL could also be improved by more than a factor of 10× on the yeast and human datasets. However, no significant improvement was possible for GraphQL on wordnet. The reason why this is so, is somewhat subtle. Apart from what the algorithms are doing internally to match the query, other culprits are the characteristics of the actual stored graphs and the generated queries. Looking at the statistics of the graphs (table 2), yeast and especially wordnet are very sparse graphs with small average node degree. Thus, the majority of the generated queries are paths and the rewritings based on node degrees are not effective in this case. Additionally for wordnet, the small number of labels (only 5) and distribution of the frequencies of the labels being highly skewed leads to the generation of queries that in their majority contain only 1 or 2 labels, with the second label appearing only once. As a result, the rewritings are of little use in these cases.

7. ALGORITHM-SPECIFIC STRAGGLERS

As we already mentioned in section 4, we notice that for the NFV methods, different algorithms have different percentages of *hard* queries, leading to the conclusion that different algorithms find different queries hard. In this section we elaborate on this observation.

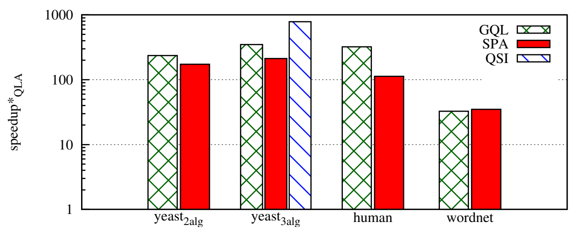


Figure 9: Avg *speedup*$^*_{QLA}$ when utilising different algorithms on NFV methods

		GraphQL	sPath	QuickSI
yeast$_{2alg}$	stdDev	1094.57	1051.65	-
	min	1.00	1.00	-
	max	9189.36	9129.60	-
	median	1.00	1.80	-
yeast$_{3alg}$	stdDev	1596.47	1255.34	2162.97
	min	1.00	1.00	1.00
	max	13060.10	12403.70	12312.70
	median	1.00	1.88	1.32
human	stdDev	1394.34	570.83	-
	min	1.00	1.00	-
	max	30873.80	4341.44	-
	median	1.00	1.04	-
wordnet	stdDev	253.56	104.42	-
	min	1.00	1.00	-
	max	3733.78	932.58	-
	median	2.47	1.00	-

Table 9: *speedup*$^*_{QLA}$ statistics when utilizing different algorithms on NFV methods

Observation 5: "Stragglers" are algorithm-specific; i.e., by evaluating the same query workloads with various algorithms, we have seen that a "straggler"-query for one algorithm can be a typical query for the other algorithms.

Fig. 9 presents the average *speedup*$^*_{QLA}$ for the yeast, human and wordnet datasets and for the tested algorithms. In table 9, we additionally report the stdDev, min, max and median of *speedup*$^*_{QLA}$. For the yeast dataset, we present the results with utilizing all 3 algorithms (noted as yeast$_{3alg}$), as well as with the pair of algorithms (GraphQL and sPath) that we utilize for the remaining datasets (noted as yeast$_{2alg}$). For the yeast dataset, all tested queries were helped by the use of different algorithms. In the human and wordnet datasets, only 0.8% and 0.1% of the queries were not helped by this scheme. Note that the *speedup*$^*_{QLA}$ values for using multiple algorithms are higher compared to the *speedup*$^*_{QLA}$ values achievable with multiple query rewritings (see §6.2). This leads to the conclusion that the use of multiple algorithms could be way more beneficial compared to the rewritings, which are not always effective (§6.2).

8. THE Ψ-FRAMEWORK

In this section we present how we incorporate our findings in a novel framework that exploits parallelism. The proposed framework is called Ψ-framework (**P**arallel **S**ubgraph **I**somorphism framework). Unlike recent related work [11, 16], by having different threads/machines working on different versions of the problem our Ψ-framework exploits parallelism in a novel way. We utilize Grapes and GGSX (as well as GraphQL and sPath) as well-established FTV (resp. NFV) methods. Within our Ψ-framework we have incorporated the original implementations of Grapes and GGSX as provided by their authors, and of GraphQL and sPath as found in [12].

In the FTV methods we leave intact the index construction and the filtering stages during query processing. In the verification stage, for every graph in the candidate set, we instantiate a number of threads equal to the number of the isomorphic-query rewritings we utilize. These threads run in parallel with each being assigned one rewriting of the initial query, and the first thread to finish is the "winner"; i.e., the rest of the threads are killed and the algorithm proceeds with the verification of the next graph in the candidate set.

Ψ-framework for the NFV methods works similarly to the verification stage of the FTV methods. However, we mentioned in observation 5 that stragglers disappear when using an alternative matching algorithm. We incorporate this finding in our Ψ-framework by running simultaneously two threads: one for sPath and one for GraphQL with the original query. Again after the completion of the fastest thread, the rest of them are killed.

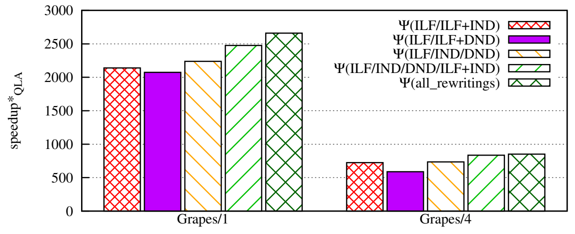
On one hand we have seen that the more the isomorphic instances we use, the better the speedup we gain in the graph matching process. On the other hand, the instantiation and synchronization of many threads come with a non-trivial overhead, impacting the overall speedup. To this end, in our performance evaluation we report on the speedup achieved by several beneficial combinations of rewritings. We note that our Ψ-framework is of course not the only solution to the straggler-queries' problem. Undoubtedly, it would be preferable to choose the right isomorphic query instance and/or algorithm to use to minimize the query execution time. However, given the complex nature of the sub-iso problem, we leave such design decisions for future work.

The cost of producing the query rewritings was measured from a few tens (for smaller query sizes) to a few hundreds (for the biggest query sizes) of μsecs; being a negligible overhead to the overall query processing time, we ignore it in the figures and omit any further discussion of this cost factor.

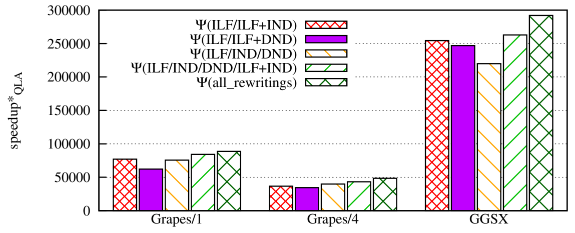
8.1 FTV methods

Fig. 10 and 11 present the avg $speedup^*_{QLA}$ and avg $speedup^*_{WLA}$ respectively for utilizing different versions of Ψ-framework on the FTV methods. Specifically, we present the avg $speedup^*_{QLA}$ and avg $speedup^*_{WLA}$ of the following versions of Ψ-framework: (a) ILF/ ILF+IND (2 threads), (b) ILF/ ILF+DND (2 threads), (c) ILF/ IND/ DND (3 threads), (d) ILF/ IND/ DND/ ILF+IND (4 threads) and (e) all 5 possible rewritings (5 threads). Our framework proves highly beneficial for all algorithms and datasets. Although not depicted in the figure, but as it was expected, by increasing the number of threads running multiple rewritings on the Ψ-framework, not only the avg execution time is significantly improved but also the percentage of hard queries is decreased, even leading to straggler-free executions. However, note that the Ψ-framework(ILF/ IND/ DND) (3 threads) is only 3-8% worse compared to Ψ-framework(ILF/ IND/ DND/ ILF+IND) (4 threads) for Grapes/1 and Grapes/4.

As Grapes is designed as a multi-threaded application, we additionally compare Grapes/4 against our Ψ-framework running Grapes/1 with the following four rewritings (for to-

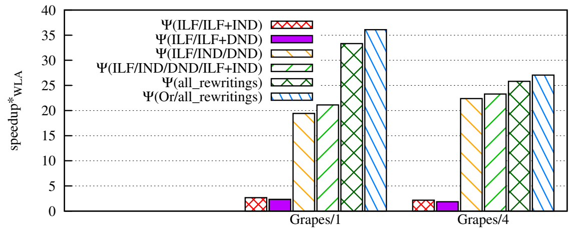


(a) Synthetic dataset

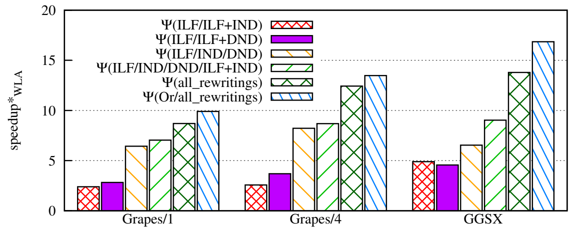


(b) PPI dataset

Figure 10: Avg $speedup^*_{QLA}$ across different versions of our framework on the FTV methods



(a) Synthetic dataset



(b) PPI dataset

Figure 11: Avg $speedup^*_{WLA}$ across different versions of our framework on the FTV methods

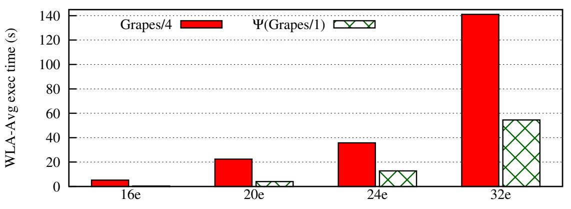
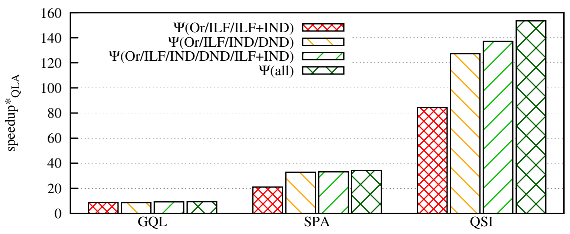
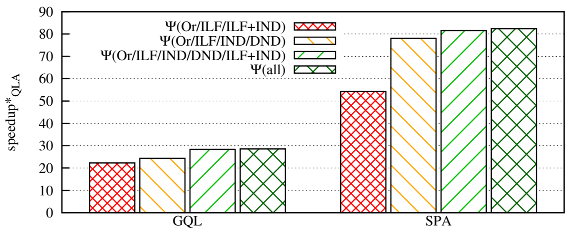


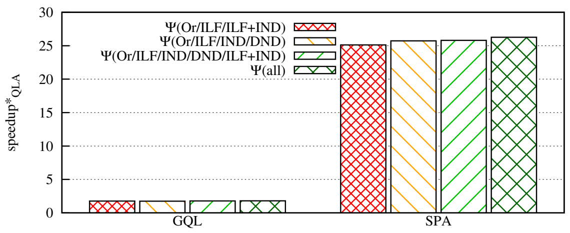
Figure 12: Comparison of avg exec time over the PPI dataset, for Grapes/4 against the Ψ-framework with 4 rewritings over Grapes/1



(a) yeast dataset



(b) human dataset



(c) wordnet dataset

Figure 13: Avg $speedup^*_{QLA}$ across different versions of Ψ-framework on the NFV methods

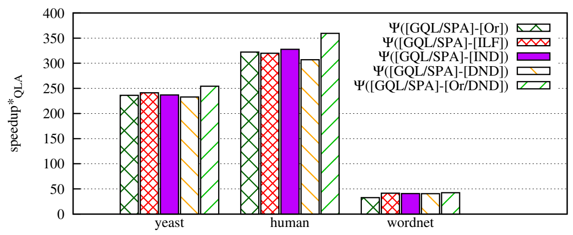
	PPI	yeast	human	wordnet
Grapes/4	6.29%	-	-	-
GraphQL	-	4.3%	10%	1.6%
sPath	-	2.8%	4.4%	13%
Ψ-fram	2.06%	0%	0.7%	0%

Table 10: Percentage of killed queries of FTV methods and our Ψ-framework

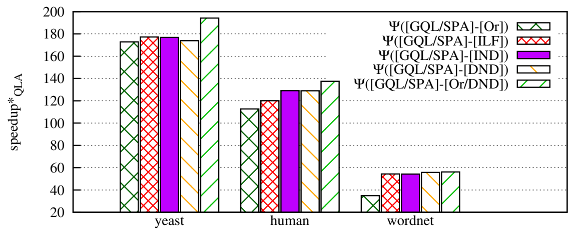
tal of 4 threads as well): ILF, IND, DND, ILF+IND. The results are presented in fig. 12 for the PPI dataset (results for the synthetic dataset were similar). Table 10 reports the percentage of killed queries for Grapes/4 and Ψ-framework on PPI. As is obvious, although both contenders have the same level of parallelism, Ψ-framework makes better use of its threads and leads to lower query processing times.

8.2 NFV methods

Fig. 13 presents the avg $speedup^*_{QLA}$ for utilizing different versions of Ψ-framework on the NFV methods (we omit the figures for avg $speedup^*_{WLA}$ due to space constraints). We utilize the following versions of Ψ-framework and the corresponding number of threads: (a) Orig/ ILF/ ILF+IND (3 threads) (b) Orig/ ILF/ IND/ DND (4 threads), (c) Orig/ ILF/ IND/ DND/ ILF+IND (5 threads), and (d) Orig + all-rewritings (titled as all) (6 threads). For all tested datasets and workloads, GraphQL benefited the least by the rewritings. The biggest improvements appear in the human

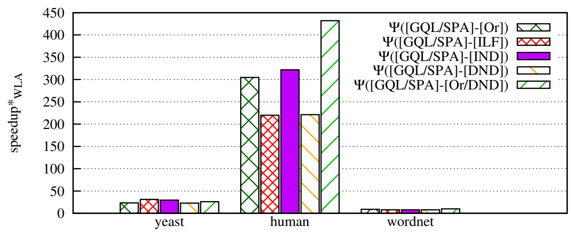


(a) $speedup^*_{QLA}$ for GraphQL

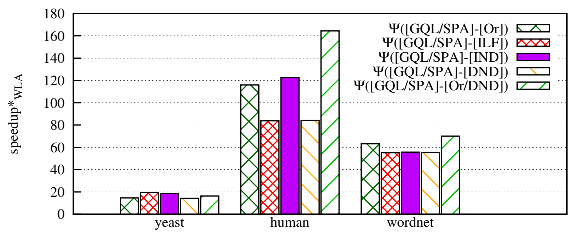


(b) $speedup^*_{QLA}$ for sPath

Figure 14: Avg $speedup^*_{QLA}$ for running multiple algorithms against NFV methods on Ψ-framework



(a) $speedup^*_{WLA}$ for GraphQL



(b) $speedup^*_{WLA}$ for sPath

Figure 15: Avg $speedup^*_{WLA}$ for running multiple algorithms against NFV methods on Ψ-framework

dataset. We attribute this to the fact that this dataset comprises a denser graph with more labels, thus a larger portion of "hard" queries benefited by our rewritings and framework.

Finally, fig. 14 and 15 depict the avg $speedup^*_{QLA}$ and the avg $speedup^*_{WLA}$ for utilizing different algorithms and different versions of Ψ-framework on the NFV methods and on yeast, human and wordnet, against vanilla GraphQL and sPath respectively. We instantiated the following versions of our Ψ-framework with the corresponding number of threads: (a) GraphQL-Orig/ sPath-Orig (2 threads), (b) GraphQL-ILF/ sPath-ILF (2 threads), (c) GraphQL-IND/ sPath-IND (2 threads), (d) GraphQL-DND/ sPath-DND (2 threads). (e) GraphQL-Orig /sPath-Orig/ GraphQL-DND/ sPath-DND (4 threads). For both GraphQL and sPath, we were able to achieve up to 3 orders of magnitude improvement with our Ψ-framework on both per-query and per-workload metrics. Also, with the Ψ-framework, the percentage of hard queries was reduced and, for yeast and wordnet, hard queries became extinct – see Table 10.

9. CONCLUSIONS

We have studied the subgraph isomorphism problem, in both its decision and matching versions, using well-established FTV and NFV methods respectively, and against several different real and synthetic datasets of various characteristics. Our research has revealed and quantified a number of insights, concerning (i) the existence and role of straggler queries in a method's overall performance, (ii) the dramatically varying performance of isomorphic queries, (iii) the impressive impact that query rewriting can have when used before executing the query with several algorithms, and (iv) the fact that straggler queries are algorithm-specific. We suggested and used both WLA and QLA metrics to fully appreciate the performance of algorithms in the presence of stragglers. A number of query rewritings were proposed and our results showed that in many cases there existed one rewriting that could offer great performance advantages – with different rewritings being best for different queries. We showcased that, for the NFV algorithms, when a query was proved to be very expensive with one algorithm, another algorithm would actually manage to compute its answer very efficiently. These findings then naturally culminated into a novel framework, which employs in parallel different threads, each using a different well-known algorithm and/or a specific query rewriting, per query. This introduced dramatic improvements (up to several orders of magnitude) to FTV and NFV algorithms. We hope that our findings will open up new research directions, striving to find appropriate, per-query, isomorphic rewritings, in combination with alternate per-query sub-iso algorithms that can yield large improvements. Using machine learning models to predict which version of our framework (algorithms, rewritings) to employ per query is of high interest.

10. REFERENCES

[1] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha. Enhancing graph database indexing by suffix tree structure. In *Proc. IAPR PRIB*, 2010.

[2] J. Cheng, Y. Ke, and W. Ng. GraphGen. http://www.cse.ust.hk/graphgen/.

[3] J. Cheng, Y. Ke, W. Ng, and A. Lu. FG-index: towards verification-free query processing on graph databases. In *Proc. SIGMOD*, pages 857–872, 2007.

[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE TPAMI*, 26(10):1367–1372, 2004.

[5] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha. GRAPES: A software for parallel searching on biological graphs targeting multi-core architectures. *PloS One*, 8(10):e76911, 2013.

[6] W.-S. Han, J. Lee, and J.-H. Lee. Turbo$_{iso}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proc. SIGMOD*, 2013.

[7] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. iGraph: a framework for comparisons of disk-based graph indexing techniques. *PVLDB*, 3(1-2):449–459, 2010.

[8] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proc. SIGMOD*, pages 405–418, 2008.

[9] F. Katsarou, N. Ntarmos, and P. Triantafillou. Performance and scalability of indexed subgraph query processing methods. *PVLDB*, 8(12):1566–1577, 2015.

[10] K. Klein, N. Kriege, and P. Mutzel. CT-index: Fingerprint-based graph indexing combining cycles and trees. In *Proc. ICDE*, pages 1115–1126, 2011.

[11] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *PVLDB*, 8(10):974–985, 2015.

[12] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2), 2012.

[13] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *PVLDB*, 8(5):617–628, 2015.

[14] K. Semertzidis and E. Pitoura. Durable graph pattern queries on historical graphs. In *Proc. ICDE*, 2016.

[15] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

[16] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.

[17] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *Proc. ICDE*, 2008.

[18] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the (JACM)*, 23(1), 1976.

[19] J. Wang, N. Ntarmos, and P. Triantafillou. Indexing query graphs to speedup graph query processing. In *Proc. ACM EDBT*, pages 41–52, 2016.

[20] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *Proc. ICDE*, pages 976–985, 2007.

[21] Y. Xie and P. Yu. CP-Index: on the efficient indexing of large graphs. In *Proc. CIKM*, 2011.

[22] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proc. SIGMOD*, pages 335–346, 2004.

[23] X. Yan, F. Zhu, P. S. Yu, and J. Han. Feature-based similarity search in graph structures. *ACM TODS*, 31(4):1418–1453, 2006.

[24] D. Yuan and P. Mitra. Lindex: a lattice-based index for graph databases. *VLDBJ*, 22(2):229–252, 2013.

[25] S. Zhang, M. Hu, and J. Yang. TreePi: A Novel Graph Indexing Method. In *Proc. ICDE*, 2007.

[26] S. Zhang, S. Li, and J. Yang. GADDI: Distance Index Based Subgraph Matching in Biological Networks. In *Proc. EDBT*, pages 192–203, 2009.

[27] S. Zhang, J. Yang, and W. Jin. SAPPER: subgraph indexing and approximate matching in large graphs. *PVLDB*, 3(1-2):1185–1194, 2010.

[28] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1-2):340–351, 2010.

[29] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: tree + delta >= graph. In *PVLDB*, pages 938–949, 2007.

[30] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *Proc. EDBT*, pages 181–192, 2008.