Towards an Algebraic Cost Model for Graph Operators

Alexander Singh and Dimitrios Tsoumakos

Department of Informatics, Ionian University, Corfu, Greece
{p13sing, dtsouma}@ionio.gr

Abstract. Graph Analytics has been gaining an increasing amount of attention in recent years. This has given rise to the development of numerous graph processing and storage engines, each featuring different models in computation, storage and execution as well as performance. Multi-Engine Analytics present a solution towards adaptive, cost-based complex workflow scheduling to the best available underlying technology. To achieve this in the Graph Analytics case, detailed and accurate cost models for the various runtimes and operators must be defined and exported, such that intelligent planning can take place. In this work, we take a first step towards defining a cost model for graph-based operators based on an algebra and its primitives. We evaluate its accuracy over a state of the art graph database and discuss its advantages and shortcomings.

1 Introduction

In recent years, we observe an increasing interest in Graph Data Analytics. Initially driven by the surge in social graph data and analysis, graph analytics can be utilized to effectively (and intuitively in many cases) tackle multiple tasks in bioinformatics, social community analysis, traffic optimization in IoT, optimization/robustness in power grids and large networks, RDF data, etc. In graph analytics, the primitives of a data graph G, i.e., its vertices and edges are mapped to problem entities and the respective relationships between them.

A continuously expanding set of approaches and tools have emerged, in order to assist in efficient storage and computation of various algorithms over big data graphs ((11; 2; 9; 1; 3), etc). These systems differ in one or more aspects, with the most important being their storage and computation (or execution) model (16). However, one size does not fit all: No single execution model is suitable for all types of tasks and no single data model is suitable for all types of data. Modern workflows consist of series of diverse operators over heterogeneous data sources and formats. *Multi-Engine Analytics* has been proposed as a solution that can optimize for this complexity and is gaining ground ever since (e.g., (6; 7; 12)).

One of the most critical challenges in such a multi-engine environment is the design and creation of a *meta-scheduler* that automatically allocates tasks to the right engine(s) according to multiple criteria, deploys and runs them without manual intervention. For such a meta-scheduler to function properly,

accurate *cost models* of the operators utilized by the underlying platforms or datastores must be defined or exported. The cost/performance tradeoffs will be then evaluated by the scheduler in order to adaptively "mix-and-match" operators to different engines in order to achieve a user-defined performance function (relating to time, cost, accuracy, etc). Yet, while cost models have been thoroughly studied in traditional systems (e.g, RDBMSs or traditional big data platforms like Hadoop), this is not the case for Graph Analytics runtimes.

In this work, we take a first step towards this direction. Specifically, using the algebra defined in (10), we describe the decomposition of common graph operations down to primitive operators. We then devise a cost model and empirically evaluate it using a state of the art graph database (3) loaded with varying sizes and types of directed graphs and queries based on the presented decompositions. We assess the strengths and weaknesses of the resulting cost model and identify factors that contribute to its accuracy. Defining an engine-based cost model, that still originates from a general algebra, is a decisive step towards multi-engine cost-based optimization of graph analytics workflows.

2 Related work

Inspired by the surging interest in graph databases, a number of works that focus on foundations of graphs and graph-databases, RDF and triple-stores have appeared.

An algebra for RDF Graphs was presented in (14), focused on querying large-scale, distributed triple-stores on shared-nothing clusters. The authors present the algebra as well as denotational semantics for it, while additionally presenting some preliminary experimental results. Another algebraic framework, also focusing on RDF data querying, was presented in (8). This work focuses on defining an algebra to be used in the formal specification of an RDF query language. It is presented as a set of extraction, loop, construction and other operators, focusing on both the constructive and the extraction-related aspects of an RDF query language.

A SPARQL algebra to be used as the foundation for optimizing SPARQL-queries was presented in (15). The authors define set-based semantics for SPARQL via a set algebra, further providing algebraic equivalence rules to be used for optimization purposes. They also identified fragments of SPARQL together with their complexity classes. Another SPARQL-related work was presented in (5), focusing on the transformations from SPARQL to the more traditional relational algebra framework. The author discusses this transformation framework, the mismatch between SPARQL and relational semantics and additionally outlines an SQL-based translation.

An algebra and related equivalence rules on transformations of graph patterns in Neo4j's property graph model is presented in (10). A set of operators for retrieval and selection of edges and nodes is discussed. These operators result in the output of *graph relations*. Additionally, a set of equivalence rules is presented to aid in query optimization. A demonstration shows how these equivalences

can be used to algebraically transform Cypher queries at the logical level, and a performance example is given, in terms of database hits, comparing the query evaluation plan found by Neo4j to the one given by the equivalences.

A presentation of a foundational framework for RDF databases is made by Arenas et al. (4), which includes a sound-and-complete deductive system focused on RDF-graph entailment. The work also includes discussions of algebraic syntax and compositional semantics for SPARQL, its expressive power, complexity considerations of evaluating various fragments of it and the optimization of SPARQL queries. A discussion on the issue of RDF-based queries in an RDFS framework is also included and an extension of SPARQL with navigational capabilities is presented.

3 Algebraic Framework

3.1 Data Model

The data model in this work is based on the notions of property graphs and graph relations (see (10)), defined as follows:

Definition 1. *Property Graph*
Let $G = (V, E, \Sigma_v, \Sigma_e, A_v, A_e, \lambda, L_v, L_e)$ be a property graph, where V is a set of nodes, E is a set of edges, Σ_v is a set of node labels, Σ_e is a set of edge labels. We also have that A_v is a set of node properties, while A_e is a set of edge properties. Let D be a set of atomic domains, then a property $\alpha_i \in A_v$ is a function $\alpha_i : V \to D_i \cup \{\epsilon\}$ assigning a property value from a domain $D_i \in D$ to a node $v \in V$, if v has property a_i - otherwise $a_i(v)$ returns ϵ. Accordingly, a property $\alpha_j \in A_e$ is a function $\alpha_j : E \to D_j \cup \{\epsilon\}$ which assigns a property value from a domain D_j to an edge $e \in E$, if e has property α_j, else $a_j(e) = \epsilon$. Finally, $\lambda : E \to V \times V$ is a function that assigns nodes to edges, $L_v : V \to \Sigma_v$ is a function that assigns labels to nodes, and $L_e : E \to \Sigma_e$ is a function assigning labels to edges.

Definition 2. *Graph Relation*
Let G be a property graph. Then, a relation R is a graph relation if the following is true:

$$\forall A \in attr(R) : dom(A) = V \cup E \tag{3.1.1}$$

where $attr(R)$ is the set of attributes of R (i.e., columns) and $dom(A)$ is the domain of attribute A.

3.2 Base Operators

Our algebraic framework is built on the definition of two primitive operators, *getNodes* and *expand* (10). These two operands can then be composed in order to produce higher-level graph operations. We proceed with their description, assuming a static graph G to operate on.

Definition 3. *getNodes Operator*
Consider a property graph G. The getNodes operator (denoted by $\bigcirc$) takes as argument a label x and outputs a graph relation with a single attribute x containing all the nodes of G:

$$val(\bigcirc_x) = V, \tag{3.2.1}$$

$$sch(\bigcirc_x) = \langle x \rangle \tag{3.2.2}$$

Where $val(R)$ is the set of tuples in R and $sch(R)$ is the schema of R.

Definition 4. *expand Operator*
Consider a property graph G. The expand operator, which takes as inputs a relation R, an attribute $x \in attr(R)$ and a new attribute label y, works by expanding the graph relation R to include the immediate neighbors of nodes under x reachable by an ingoing or outgoing edge. It does so by adding a new column y to R, containing the nodes than can be reached by an ingoing or outgoing edge (see below) from nodes of x. It comes in two variants, an expandIn (denoted by $\downarrow_x^y$) operator that selects neighbors that are reachable by ingoing edges, and an expandOut (denoted by $\uparrow_x^y$) which selects based on outgoing ones respectively:

$$val(\uparrow_x^y (R)) = \{\langle t, e, v\rangle | t \in val(R) \wedge e \in E \wedge \lambda(e) = (t.x, v)\} \tag{3.2.3}$$

$$sch(\uparrow_x^y (R)) = sch(R) \| \langle xy, y\rangle \tag{3.2.4}$$

where $val(R)$ is the set of tuples in R, $sch(R)$ is the schema of R, and $\|$ denotes concatenation of schema tuples. The semantics of $\downarrow_x^y$ can be defined in much the same way:

$$val(\downarrow_x^y (R)) = \{\langle t, e, v\rangle | t \in val(R) \wedge e \in E \wedge \lambda(e) = (v, t.x)\} \tag{3.2.5}$$

$$sch(\downarrow_x^y (R)) = sch(R) \| \langle yx, y\rangle \tag{3.2.6}$$

Apart from the aforementioned operators, we will also utilize the traditional relational algebra operators: select (σ), project (π) and join ($\bowtie$), which operate on graph relations with similar semantics to the traditional sense. As an example of the definitions above, consider the example of applying $\uparrow_x^y (\bigcirc_x)$ on a graph G, as shown in Figure 1.

3.3 Cost Model

We now define space and time costs for the *getNodes* and *expandIn*/*expandOut* operators. We note that the cost model is implementation-specific with multiple practical factors such as algorithmic implementation, the underlying data structures, relational operator implementation, etc having an effect on it. For simplicity, we consistently assume bare-bones implementations, i.e., graphs implemented as lists V, E containing nodes and edges (as node-tuples) respectively, joins using naive loop algorithms, etc. This assumption does not compromise our

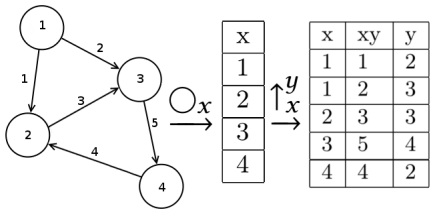

Fig. 1. Example of operations on graph

methodology, as factors can be modified and be "plugged in" to match the implementation at hand. We measure cost in terms of both (predicted) space/row count of the relations produced by each operator and (predicted) time/primitive operations taken to compute each query. The cost of the *getNodes* operator only scales with respect to the number of nodes present in our graph:

$$SpaceCost(\bigcirc_{label}) = O(|V|) \tag{3.3.1}$$

$$OpCost(\bigcirc_{label}) = O(|V|) \tag{3.3.2}$$

The costs of the *expandIn* and *expandOut* operators scale with respect to the size of the input graph relation, as well as the average number of ingoing/outgoing edges per node of our graph respectively:

$$\begin{aligned} SpaceCost(\uparrow^{targetLabel}_{sourceLabel} R) &= O(d_{out} \cdot SpaceCost(R)) \\ SpaceCost(\downarrow^{sourceLabel}_{targetLabel} R) &= O(d_{in} \cdot SpaceCost(R)) \end{aligned} \tag{3.3.3}$$

$$\begin{aligned} OpCost(\uparrow^{targetLabel}_{sourceLabel} R) &= O((|E| \cdot SpaceCost(R)) + OpCost(R)) \\ OpCost(\downarrow^{sourceLabel}_{targetLabel} R) &= O((|E| \cdot SpaceCost(R)) + OpCost(R)) \end{aligned} \tag{3.3.4}$$

where R is a series of operations resulting in a graph relation, and d_{in} and d_{out} is the average number of ingoing and ougoing edges per node in G, the property graph we operate on.

Note that computing the cost for compositions of operators is done via recursion. For instance:

$$SpaceCost(\uparrow^{y}_{x} (\bigcirc_x)) = (d_{out} \cdot SpaceCost(\bigcirc_x)) = d_{out} \cdot |V| \tag{3.3.5}$$

$$OpCost(\uparrow^{y}_{x} (\bigcirc_x)) = O(|E| \cdot SpaceCost(\bigcirc_x) + OpCost(\bigcirc_x) = O((|E| \cdot |V|) + |V|) \tag{3.3.6}$$

4 Graph Operator Decomposition

In the following subsections we describe decompositions of frequently-used graph operations using the operators defined above.

4.1 Finding cycles

Consider the task of querying a graph database for cycles. This is a popular operator found in many use cases (e.g., SPARQL query processing (13)). Expressing this query in terms of the base operators depends on the edge orientations we want to consider. For example, if we are to consider squares (cycles of size 4) formed with outgoing edges, the query can be expressed as seen in the left subfigure of Figure 2. Similarly, we can make use of *expandIn* operators for the ingoing case, or even mix the two operators to find cycles of mixed, but defined beforehand, edge orientations.

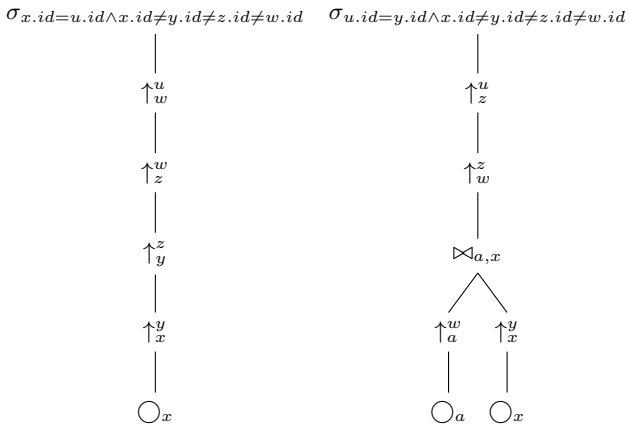

Fig. 2. Two alternative decompositions of the square query.

Another way to translate the above square query can be seen in the right subfigure of Figure 2. Note that both queries will return graph relations that potentially contain rows that describe the same cycle – each one traversing a cycle's nodes in a different order.

We can now compute the costs of these two queries. We know that both queries will return the same results after the final selection, namely all the squares found in our graph. We also cannot compute the space costs for such a selection without knowing more about the number of squares in our graph. Thus, we only consider the costs up to that selection. For the first query, the cost is[1]:

$$SpaceCost(\uparrow^u_w \circ \uparrow^w_z \circ \uparrow^z_y \circ \uparrow^y_x (\bigcirc_x)) = O({d_{out}}^4 \cdot |V|) \tag{4.1.1}$$

$$OpCost(\sigma_K \circ \uparrow^u_w \circ \uparrow^w_z \circ \uparrow^z_y \circ \uparrow^y_x (\bigcirc_x)) = O((|E| \cdot ({d_{out}}^3 \cdot |V|)) + (|E| \cdot ({d_{out}}^2 \cdot |V|)) + (|E| \cdot (d_{out} \cdot |V|)) + (|E| \cdot |V|) + |V|) \tag{4.1.2}$$

[1] we use the notation $\circ$ to denote composition of functions

where $K \rightarrow u.id = y.id \wedge x.id \neq y.id \neq z.id \neq w.id$. For the second query, we begin by observing that the two inputs of the join are essentially the same relation. From that we can deduce that the resulting relation has the same size as the first input multiplied by d_{out} (since for each row in the first argument, we'll get d_{out} rows in the joined relation), and so we can replace $SpaceCost((\uparrow_a^w \bigcirc_a) \bowtie_{a,x} (\uparrow_x^y \bigcirc x))$ with ${d_{out}}^2 \cdot |V|$. For the operations cost we use $OpCost(A \bowtie B) = (SpaceCost(A) \cdot SpaceCost(B)) + OpCost(A) + OpCost(B)$, based on the complexity of a nested loop join (for simplicity). We also set $OpCost(\sigma A) = |A|$. The costs are then the following:

$$SpaceCost(\uparrow_z^u \circ \uparrow_w^z ((\uparrow_a^w \bigcirc_a) \bowtie_{a,x} (\uparrow_x^y \bigcirc x))) = O({d_{out}}^2 \cdot SpaceCost((\uparrow_a^w \bigcirc_a) \bowtie_{a,x} (\uparrow_x^y \bigcirc x))) \quad (4.1.3)$$

$$OpCost(\sigma_K \circ \uparrow_z^u \circ \uparrow_w^z ((\uparrow_a^w \bigcirc_a) \bowtie_{a,x} (\uparrow_x^y \bigcirc x))) = O(({d_{out}}^3 \cdot |V|) + (|E| \cdot ({d_{out}}^2 \cdot |V|)) + (|E| \cdot (|V| \cdot d_{out})) + (({d_{out}}^2 \cdot |V|) \cdot ({d_{out}}^2 \cdot |V|)) + (|E| \cdot (d_{out} \cdot |V|)) + (|E| \cdot (d_{out} \cdot |V|))) \quad (4.1.4)$$

The above queries can be readily expanded to detect cycles of length greater than 4. In general, we could also consider squares with edges of any orientation by including both variants of the *expand* operator in our expression and filtering appropriately. Finally, we might be solely interested in cycles "centered" around a specific type of node, for instance nodes with a `name` attribute whose value is "`Alice`". In such a case, we can include an additional selection operation in the above trees, right after the bottom-most $\bigcirc_x$ operation.

4.2 Random Walk, Path, and Star-path

Another frequently used graph query is that of performing an n-step random walk, starting from a specified node x. Such a query can be used to detect $s-t$ connectivity using very small amounts of space. To this end, we can introduce a new operator $RandRow(R)$ which selects a random row from a graph relation R. An example of a 2-step random walk, starting from a node with $id = ID$ is depicted by the left subfigure of Figure 3.

Another interesting path operator is the *star-path*. In a star-path we wish to find a path between two nodes of interest and further want to expand the resulting graph relation to also include the neighbors of all nodes between the two terminals. We can make use of the random walk procedure to find n-length paths. For example, if we want paths of length $n = 2$, assuming the terminal nodes have ID_1, ID_2 respectively, there is only one inbetween node and our query can be seen in the right subfigure of Figure 3. If successful (i.e., there exists a path of length 2 between the two terminals), it would add all the neighbors of the inbetween node under the attributes "a", for those connected to it via outgoing edges, and "b", for those connected via ingoing ones.

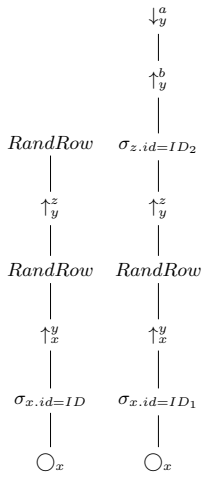

Fig. 3. A random 2-walk (left) and a star-path between ID_1 and ID_2 (right).

4.3 Grid query

By appropriately combining multiple square queries we can create a new operator that detects grids in a graph. To do this, we require access to a function that, given a graph relation R, adds a new attribute $uniqueValues$ to it, containing the number of unique values per row (see Figure 4 for an example). Once again, structuring our query appropriately is largely dependent on the specific edge orientations we need to consider.

5 Experiments

5.1 Experimental Setup

We have implemented the aforementioned algebra of graph operators and relations in Python using Neo4j (community edition 3.1.2) as the graph database and the Neo4j Python Driver `neo4j-driver` to facilitate communication with it. The platform was setup on a 8GB, Ubuntu 16.04 VM. Data graphs were generated using the NetworkX Python library. The queries benchmarked consist of a random 4-path and two square queries but modified to match a cycle with vertex/edge pattern $x \leftarrow y \rightarrow z \leftarrow w \rightarrow x$ (see Figure 5), so as to include both ingoing and outgoing edges and be of more interest than the plain query shown above. The actual time and space costs for queries presented in the benchmarks were obtained by averaging over 10 random graph relations samples for each of the queries executed. Queries that resulted in empty relations (i.e., happened to start on nodes that no cycles where centered on) were discarded. The projected time and space costs were obtained by applying our cost model, using

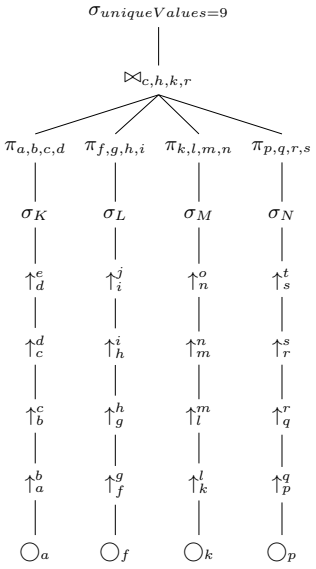

Fig. 4. Grid query, where $K \leftarrow a.id = e.id \wedge a.id \neq b.id \neq c.id \neq d.id$, $L \leftarrow f.id = j.id \wedge f.id \neq g.id \neq h.id \neq i.id$, $M \leftarrow k.id = o.id \wedge k.id \neq l.id \neq m.id \neq n.id$, $N \leftarrow p.id = t.id \wedge p.id \neq q.id \neq r.id \neq s.id$

node count, edge count, indegree/outdegree, etc., statistics provided by the NetworkX library (see Table 1). To compare the projected number of operations to actual time costs in seconds, the projections were multiplied by constant factors.

Table 1. Statistics for graphs used in our benchmark, grouped by type of graph.

Type	Node Count	Edge Count	Average Indegree/Outdegree
Small Random	35	[242-1190]	[6.9143-34]
Dense Random	[1K-10K]	[5046 - 49639]	5.06
Sparse Random	[1K-10K]	[1998 - 19887]	2.04
Scale-Free	[1K-10K]	[2138 - 21439]	2.16

5.2 Results and Discussion

We now present the results of our benchmarks that compare the predictions of our cost model against actual costs obtained from Python implementations of each query over a Neo4j database.

We start with an idealized situation: Consider a small graph of fixed node count $n = 35$, where each node has a probability p to connect to any other node in the graph. By steadily increasing p we slowly approach a regular graph,

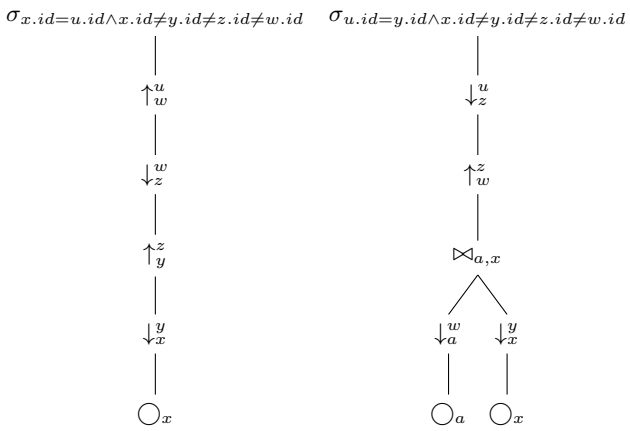

Fig. 5. The two square queries, thereafter referred to as SqrOne (left) and SqrTwo (right).

a situation in which our model's predictions will theoretically perfectly match the actual results. For the two square queries mentioned before, we showcase the results in Figure 6 (graphs generated using the `gnp_random_graph` function of the NetworkX Python Library, with increasing `p`). We note that the model accurately predicts size costs and also provides good results for the time costs.

Yet, a very small close-to-regular graph rarely reflects real life data. In the next benchmarks, we consider the projected and actual costs of two square queries on three sets of graphs: A pair of sparse (with average indegree/outdegree of around 2) and dense (average indegree/outdegree of around 5) random graphs, generated using the `gnp_random_graph` with increasing `n`, and a set of scale-free graphs generated using `scale_free_graph`[2]. For the two sets of random graphs, we can see in Figures 7 and 8 that our cost model reasonably describes the size behavior of the queries as well as the time behavior of the first square query, while it is not as accurate for the second one. For the scale-free graphs, we note in Figure 9 that the cost model we have described so far fails to capture the behavior of the actual queries. The structure of the graphs is largely responsible for this – the presence of large "hubs", i.e., nodes that have a very large number of ingoing and outgoing edges. This is especially true for the ingoing case, where such hubs heavily skew the indegree statistics from a median of 0 to an average of 2, the same as the outdegree average which however has a median of 1.

To deal with such a discrepancy in our model we need to modify it accordingly. An easy fix, assuming input graph structure was available, would be to replace the d_{out} and d_{in} factors with new ones that take hubs into consideration. We do this by replacing d_{out} with $OutGoing_{max}/d_{out}$ and d_{in} with $InGoingTrimmed_{max}/dTrimmed_{in}$, where $OutGoing_{max}$ is the maximum out-

[2] Using default arguments: `alpha=0.41, beta=0.54, gamma=0.05, delta_in=0.2, delta_out=0`

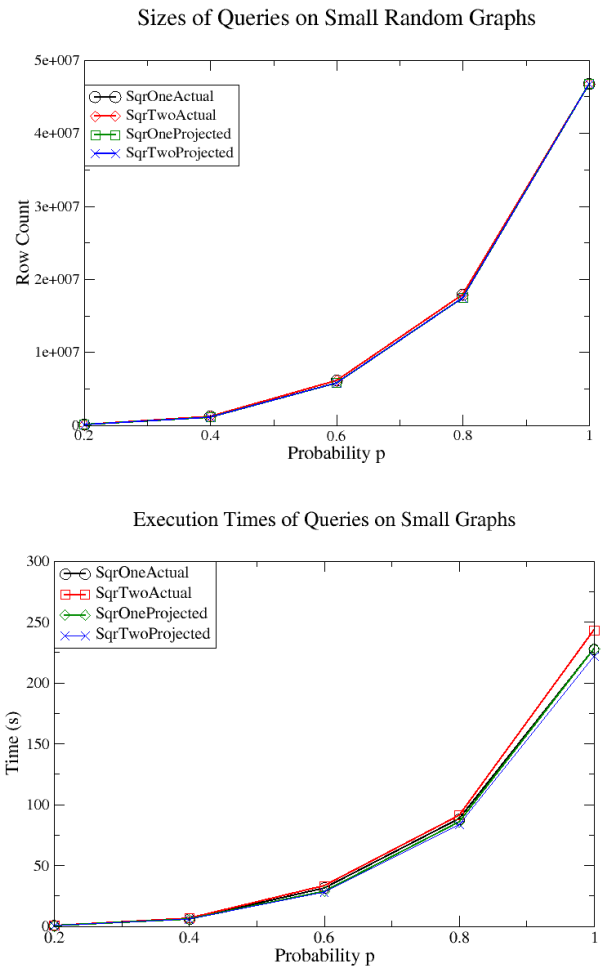

Fig. 6. Space and time costs (actual and projected) of the two square queries on small random graphs with increasing probability p of any two nodes connecting.

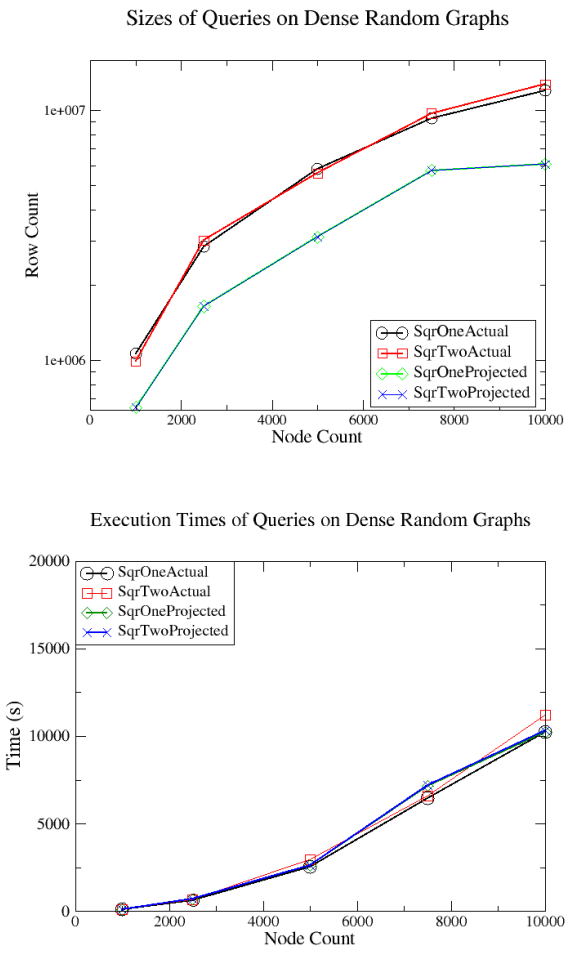

Fig. 7. Space and time costs (actual and projected) of the two square queries on dense random graphs with increasing N.

degree, $dTrimmed_{in}$ is the trimmed average of indegrees, and $InGoingTrimmed_{max}$ is the maximum indegree after *trimming*: We only need to "trim" the top 1% nodes (ordered by indegree) to obtain the predictions in Figure 9. We can see that the modified costs more closely model the behavior of the actual queries and capture the difference between the two queries: The first one consumes more time and space than the second one, due to its use of more *expandOut* operations with high d_{out} factors as opposed to the second one which uses more *expandIn* operations.

Respective results for random four-walks follow the same general trends and can be found in the Appendix.

5.3 Including label information in the cost model

So far, we have discussed queries that only make use of global, non label-specific, statistics such as the average in/out-degree. However, it is also of interest to consider graphs that embellish edges with "label" or "type" data. To include such information in our cost models we must consider the cases where the labels are used in a query. Such a case, perhaps the most common one, is an expansion followed by a selection based on the label type:

$$query = \sigma_{xy.label=\text{“}labelType\text{”}} \circ \uparrow_x^y R \tag{5.3.1}$$

In computing the costs of such a query, we can modify our cost model to include factors specific to the presence of labels. Note that the above query contains a selection operation, and as we discussed in paragraph 4.1, computing the space costs of a projection is very difficult unless we are supplied with relevant statistics which we can use to deduce costs. Fortunately, in this specific case, such information can be garnered from label-related statistics such as the in/out-going degree of a node per label type. As such, we can make use of this to modify our space cost as follows:

$$SpaceCost(query) = O(d_{out}^{labelType} \ |R|) \tag{5.3.2}$$

where we use a superscript to denote the average degree as it pertains to a specific label type. As an example, suppose we take the sparse random graphs used in the previous experiments and assign to each edge a label $l \in \{label0, label1\}$ with a 0.3 probability of labeling it $label0$, and 0.7 of labeling it $label1$. We can then compute the relevant statistics of in/out-degrees per label, whose sums should agree with the "global" degrees d_{in}/d_{out} we've used before. Now consider the following query which makes use of a projection on the labeled edges:

$$query = \sigma_{xy.label=\text{“}label0\text{”}} \circ \uparrow_x^y \circ \bigcirc_x \tag{5.3.3}$$

Then Figure 10 shows the relevant row-count benchmark of the actual query as well the projected costs from the default cost model and the cost model presented above. In general, it is quite more difficult to devise a cost model that includes label-related factors, as opposed to the label-agnostic one we have discussed

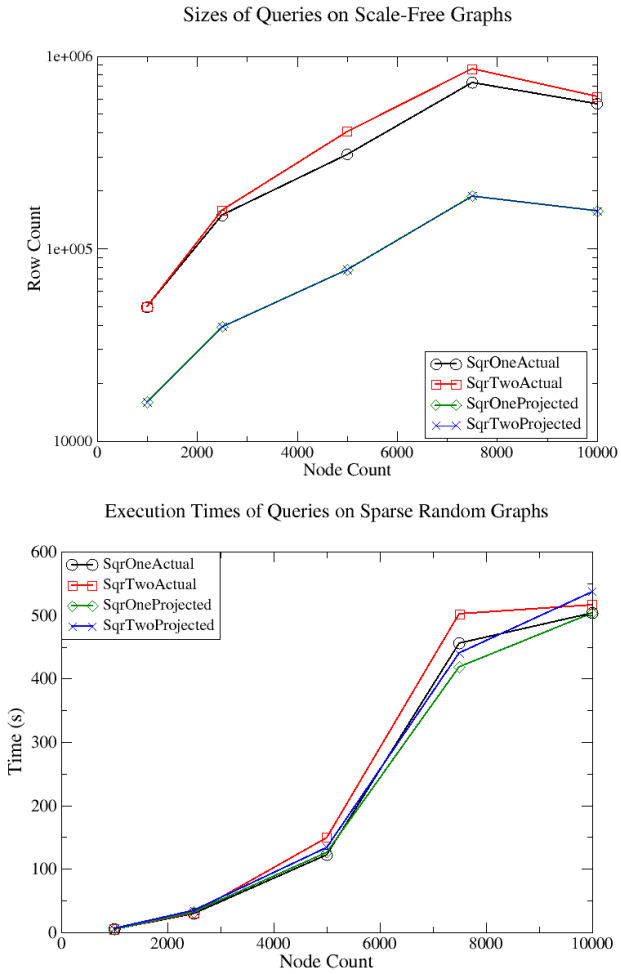

Fig. 8. Space and time costs (actual and projected) of the two square queries on sparse random graphs with increasing N.

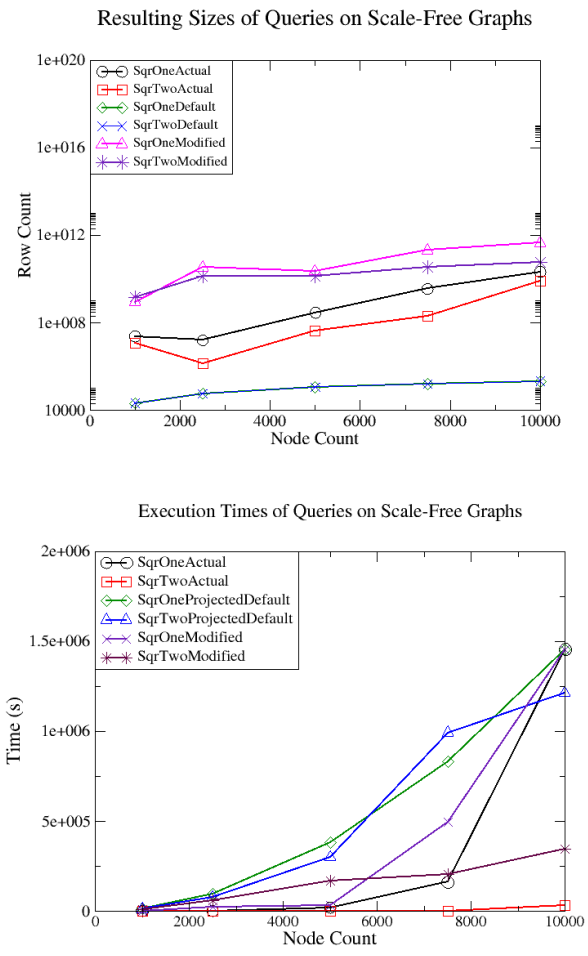

Fig. 9. Space and time costs (actual and projected via default and modified models) of the two square queries on scale-free graphs with increasing N.

earlier. One reason for that is the global behavior of the operators, in that they do not differentiate between labeled edges at all: *getNodes* fetches all nodes in a graph and *expandIn*/*expandOut* operate on all edges regardless of their label. Despite this difficulty, it is crucial to develop a cost model that includes such label-specific data, since much information about the characteristics of a graph can be found by examination of label-related metrics.

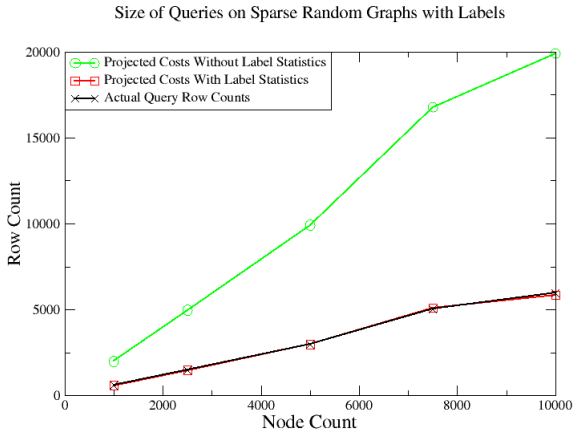

Fig. 10. Space costs (actual and projected) of query(5.3.3) using both the default cost model and the one modified to include label-related factors.

6 Conclusions

In this work we have presented an initial step towards creating algebraic cost models for various graph operators. Based on a simple model, we have demonstrated how various path operations can be decomposed and modeled utilizing the primitive operators, and how their costs can be computed. Our initial results show that, for a popular Graph DataBase, such modeling can quite accurately predict operator performance and cost. While implementation and internal computation models are important in devising model APIs, the structure and distribution of the graph itself can be of paramount importance. Thus, sampling, summarization techniques and sound knowledge in Network Science emerge as possibly critical factors towards the next steps in graph analytics modeling.

Bibliography

[1] Apache Hama. https://hama.apache.org/

[2] Apache Spark graphX. http://spark.apache.org/graphx/

[3] Neo4j. https://neo4j.com/

[4] Arenas, M., Gutierrez, C., Pérez, J.: Foundations of rdf databases. In: Reasoning Web. Semantic Technologies for Information Systems, pp. 158–204. Springer (2009)

[5] Cyganiak, R.: A relational algebra for sparql. Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170 35 (2005)

[6] Doka, K., Papailiou, N., Giannakouris, V., Tsoumakos, D., Koziris, N.: Mix 'n' match multi-engine analytics. In: 2016 IEEE International Conference on Big Data. pp. 194–203 (2016)

[7] Duggan, J., Elmore, A.J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., Zdonik, S.: The bigdawg polystore system. ACM Sigmod Record (2015)

[8] Frasincar, F., Houben, G.J., Vdovjak, R., Barna, P.: Ral: An algebra for querying rdf. World Wide Web 7(1), 83–109 (2004)

[9] Gonzalez, J., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In: Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12) (2012)

[10] Hölsch, J., Grossniklaus, M.: An algebra and equivalences to transform graph patterns in neo4j. In: EDBT/ICDT 2016 Workshops: EDBT Workshop on Querying Graph Structured Data (GraphQ) (2016)

[11] Kang, U., Tong, H., Sun, J., Lin, C.Y., Faloutsos, C.: GBASE: A Scalable and General Graph Management System. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '11 (2011)

[12] LeFevre, J., Sankaranarayanan, J., Hacigumus, H., et al.: MISO: Souping up big data query processing with a multistore system. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (2014)

[13] Papailiou, N., Tsoumakos, D., Karras, P., Koziris, N.: Graph-Aware, Workload-Adaptive SPARQL Query Caching. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. SIGMOD '15 (2015)

[14] Savnik, I., Nitta, K.: Algebra of rdf graphs for querying large-scale distributed triple-store. In: International Conference on Availability, Reliability, and Security. pp. 3–18. Springer (2016)

[15] Schmidt, M., Meier, M., Lausen, G.: Foundations of sparql query optimization. In: Proceedings of the 13th International Conference on Database Theory. pp. 4–33. ACM (2010)

[16] Yan, D., Bu, Y., Tian, Y., Deshpande, A., Cheng, J.: Big Graph Analytics Systems. In: Proceedings of the 2016 International Conference on Management of Data. SIGMOD '16 (2016)

7 Appendix

7.1 Random 4-walk benchmarks

For the random 4-walks, we first note that $SpaceCost(RandRow(R)) = 1$, so final space costs collapse to 1. For the time costs, we present Figure 11. We note that, in this case, both default and modified models fare well in their predictions – this can be attributed to the simplicity of the query and the fact that it uses only *expandOut* operations.

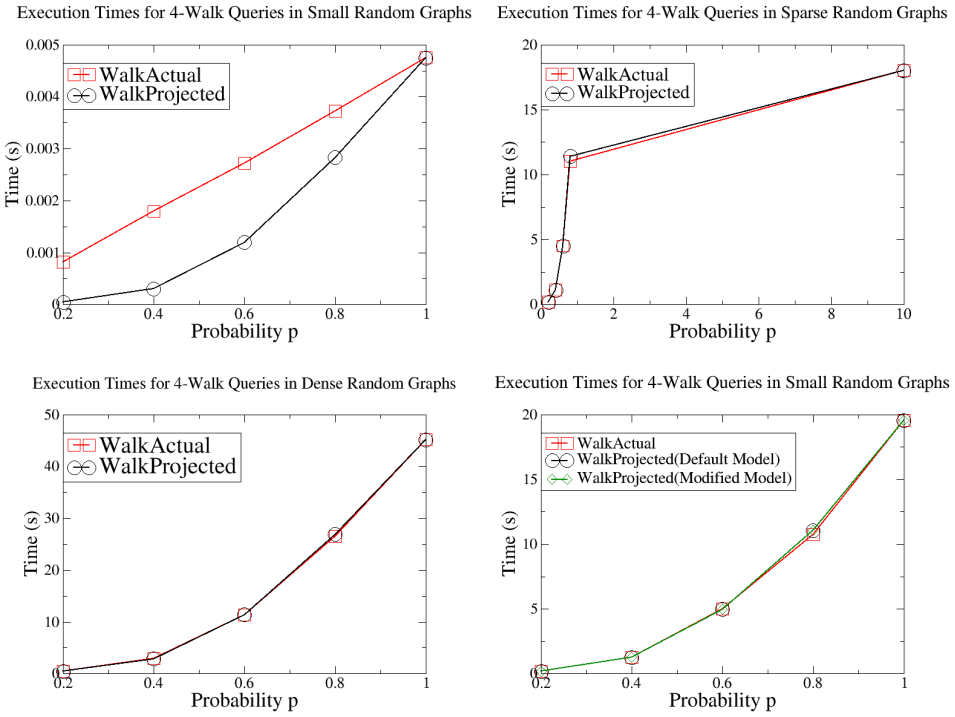

Fig. 11. Space and time costs (actual and projected via default and modified models) of 4-walk queries on graphs with various connectivity probabilities.