Short Paper

Histogram Domain Ordering for Path Selectivity Estimation

Nikolay Yakovets, Li Wang, George Fletcher
TU Eindhoven, Netherlands
{hush@,l.wang.3@student.,g.h.l.fletcher@}tue.nl

Craig Taverner
Neo4j, Sweden
craig.taverner@neo4j.com

Alexandra Poulovassilis
Birkbeck, University of London, UK
ap@dcs.bbk.ac.uk

ABSTRACT

We aim to improve the accuracy of path selectivity estimation in graph databases by intelligently ordering the domain of a histogram used for estimation. This problem has not, to our knowledge, received adequate attention in the research community. We present a novel framework for the systematic study of path ordering strategies in histogram construction and use. In this framework, we introduce new ordering strategies which we experimentally demonstrate lead to significant improvement of the accuracy of path selectivity estimation over current strategies. These positive results highlight the fundamental role that domain ordering plays in the design of effective histograms for efficient and scalable graph query processing.

1 INTRODUCTION

Analytics on graph-structured data is increasingly important in a variety of domains, e.g., role discovery in social networks, impact analysis in citation networks, functional analysis of biological networks, and querying knowledge graphs. Querying in graph query languages such as openCypher and PGQL is at the heart of these analytics tasks [1, 3, 11]. However, current graph database systems have difficulty in scaling query processing as the size and complexity of graph data collections continue to grow [4, 9].

Towards addressing this challenge, a crucial step in scalability of graph databases is the generation of effective query execution plans. Query optimizers rely on accurate data statistics for cardinality estimation during plan generation. Histograms are among the most widely used data structure for maintaining statistics for cardinality estimation, in particular for relational database systems [5]. However, there has been relatively little work on histograms for graph queries, even for the most basic graph query building block, namely, path queries [6–8, 10].

Our contributions. In this paper, we give an overview of findings in our ongoing investigations into histograms for path selectivity estimation [12]. We focus in particular on ordering strategies for path queries, i.e., how to order the domain over which histograms are built, with the goal of minimizing the variance within histogram buckets (and thereby improving estimation accuracy). We present a novel framework for systematically introducing ordering strategies, showing experimentally that the choice of domain ordering is a fundamental aspect of effective histograms. We introduce new ordering strategies which we demonstrate lead to significant improvement on the accuracy of obtained estimates, over current ordering approaches.

State of the art. The study and efficacy of histogram-based cardinality estimation are well-established [5], e.g., for path and twig query optimization in XML databases [2, 13]. Several studies have also considered path selectivity estimation on graph data with cycles (i.e., beyond trees and DAGs) [6–8, 10]. These works, however, have not investigated histograms or the impact of path ordering on estimation quality. To the best of our knowledge, we present here the first systematic study of this basic aspect of histogram construction and use in graph data management.

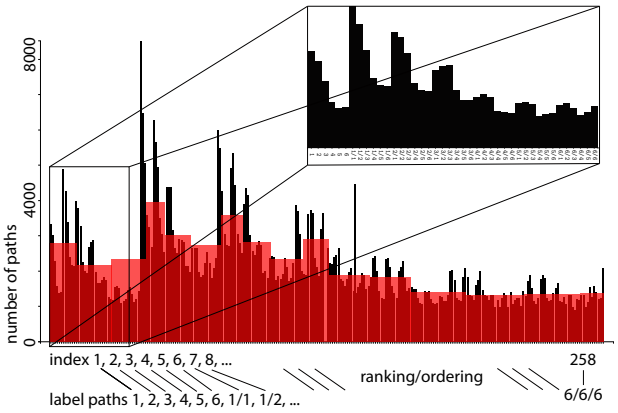

Figure 1: Visualization of a data distribution (black) and an equi-width histogram (red) of Moreno Health dataset with $k = 3$.

2 HISTOGRAMS ON LABEL-PATHS

We investigate selectivity estimation of path queries on graphs. A *graph* G is composed of a finite set of vertices V, a set of edge labels L, and a set of directed labeled edges $E \subseteq V \times L \times V$. A *k-label path* is a sequence $\ell = l_1/\ldots/l_k$, where $l_i \in L$, for all $1 \leq i \leq k$. We say $k = |\ell|$ is the *length* of ℓ. Viewing ℓ as a *path query*, the evaluation of ℓ on G returns the set $\ell(G)$ consisting of all pairs of vertices (v_s, v_t) in G such that there exist vertices $v_0, v_1, ..., v_k \in V$ where $v_s = v_0$, $v_t = v_k$, and for $0 < i \leq k$, $(v_{i-1}, l_i, v_i) \in E$. The total number of such pairs, i.e., the cardinality of $\ell(G)$, is called the *selectivity* of ℓ on G, which we denote by $f(\ell)$.

Let $\mathcal{L}^k$ be the set of all label paths over L with length up to k.[1] An *ordering* of $\mathcal{L}^k$ is a bijection from $\mathcal{L}^k$ to integer set $[0, |\mathcal{L}^k|)$. Once we establish an ordering on a label path set, a label path can be represented by its positional index in the ordering. For each label path ℓ, let $index(\ell)$ denote the index of ℓ in the ordering.

A *histogram* is a mechanism used to provide the approximation of frequency for a given value (point query) or value range (range query) without storing or accessing the complete original data distribution. More precisely, given an attribute $\mathbf{X}$, a histogram on this attribute is constructed by partitioning the data distribution of $\mathbf{X}$ into $\beta \geq 1$ mutually disjoint subsets called *buckets* and storing the statistics information and bucket boundaries for each bucket. In this work, attribute $\mathbf{X}$, also called the *domain* of the

[1] We will let $\mathcal{L}$ denote a label path set regardless of k when this does not cause ambiguity.

© 2018 Copyright held by the owner/author(s). Published in Proceedings of the 21st International Conference on Extending Database Technology (EDBT), March 26-29, 2018, ISBN 978-3-89318-078-3 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

histogram, is an ordered label path sequence produced by an ordering of $\mathcal{L}^k$. Then, given label path ℓ and its index $index(\ell)$, such a *label-path histogram* is used to compute an estimate $e(\ell)$ of the selectivity $f(\ell)$. An example of a label-path histogram is shown in Figure 1.

3 ORDERING FRAMEWORK

The purpose of histogram domain reordering is to ensure that label paths with similar cardinality are located close to each other, such that they can be allocated in the same bucket. This leads to lower variance, lower error rates, and overall better quality.

An intuitive and ideal way is to arrange the data distribution such that when $index(\ell)$ increases, $f(\ell)$ monotonically increases or decreases. The most straightforward, yet not feasible, approach is to sort the label paths by their selectivity and assign the *index* of each label path as its position in this sequence. This idea is not practical, however, as it requires extra memory to store $|\mathcal{L}|$ *index* values. The exact amount of memory can also be used to store the cardinality for each label path, such that instead of returning an estimation of selectivity, we can obtain the precise selectivity. We call such ordering an *ideal ordering*. Despite an ideal ordering being prohibitive, we can still construct an approximately monotonic sequence based on the awareness of precise cardinalities of a subset of $\mathcal{L}$.

For example, by looking at Figure 1, one can observe that the label *1* has the highest cardinality among all length-1 label paths while label *5* has the lowest. Similar trend repeats in the other 6-member groups with the same prefix {1/1, 1/2, ..., 1/6}, {2/1, 2/2, ..., 2/6}, and so on. Hence, we can assume that the label path that is composed of label paths with high cardinalities should also have high cardinality.

3.1 Concepts

We define a *base label set* as a $B \subseteq \mathcal{L}$ such that every label path in $\mathcal{L}$ can be decomposed into pieces which are all in B.[2] Then, a *splitting rule* defines how to decompose a label path. For example, $\mathcal{L}^6$ on Moreno Health dataset is {1, 2, 3, 4, 5, 6, 1/1, ..., 6/6/6/6/6/6}, if we choose B to be $\mathcal{L}^2$, with a *greedy splitting* rule which at each split step always cuts a piece in B as long as possible. For example, label path "4/4/3/3/6" is decomposed into "4/4", "3/3" and "6".

An ordering method can be described by the following three components. First, we need a base label set B. Second, we define *(un)ranking function* over the base label set that gives a rank for each base label and vice-versa. It is a bijection which maps between edge label set B and integer set $[1, |B|]$. Finally, we construct an *ordering rule* which is combined with a ranking rule to eventually determine the index of a label path (sequence of base labels) in $\mathcal{L}^k$. It is a bijection that maps between label path set $\mathcal{L}$ and integer set $[0, |\mathcal{L}^k|)$. A complete ordering method, therefore, is seen as the combination of a ranking rule and an ordering rule on a given dataset. We refer to an ordering method that is composed of ranking rule A and ordering rule B as *B-A* ordering.

We define two ranking rules in our study. *Alphabetical ranking* assigns ranks based on the alphabetical order of base labels. *Cardinality ranking* is ranking based on the cardinality of base labels, which places a base label with lower cardinality in front of the label with higher cardinality, i.e., $l_1 <^{card} l_2 \iff f(l_1) < f(l_2)$

[2]Naturally, $L \subseteq B$, otherwise there might exist label paths which cannot be decomposed into label paths in B.

In this work, we focus on the approach that takes the edge label set as the base label set, i.e., $B = L$. We define two bijections: *alph* and *card*. Let $alph(l)$ and $card(l)$ denote the index of edge label l, which will be referred to as the *rank* of l, in the set L totally ordered by alphabetical order and cardinality, respectively.

3.2 Numerical and Lexicographical Orderings

In numerical ordering, each rank is an integer, and a composition of ranks produces a number in $|B|$-based numeral system. For example, to compare two label paths $\ell_1 = l_1^1/l_2^1/\ldots/l_m^1$ and $\ell_2 = l_1^2/l_2^2/\ldots/l_n^2$, if one is shorter than the other then it has a lower ranking (rule (1) below), otherwise the two paths' labels are compared pairwise until a pair of different values is found at position i (rule (2) below):

$$\ell_1 < \ell_2 \iff \begin{cases} |\ell_1| < |\ell_2| & |\ell_1| \neq |\ell_2| \ (1) \\ \wedge_{j=1}^{i-1}(l_j^1 = l_j^2) \wedge (l_i^1 < l_i^2) & |\ell_1| = |\ell_2| \ (2) \end{cases}$$

Lexicographical ordering is the same as the ordering rule used in dictionaries; it is similar to numerical ordering with the following difference. Instead of comparing lengths of two label paths first, we append $k - |\ell|$ *blank* symbols (i.e., special symbols for which $\forall l \in L, rank(blank) > rank(l)$) to every ℓ to form a length-k sequence. We can then apply Formula 2 to compare the resulting label paths. The time complexity of both ranking and unranking functions for numerical and lexicographical orderings is $O(k)$.

3.3 Sum-based Ordering

Given label path ℓ, the idea of *sum-based* ordering is to use the sum of ranks of all base labels in ℓ to approximate the cardinality of ℓ. While being conceptually simple, the implementation of this ordering method is not trivial. First, given a path label ℓ of length k, ℓ is split into base labels and an integer rank is computed for each of the base labels to obtain a k-length integer *permutation*. Then, the integer permutation of ℓ is mapped to $index(\ell)$ by performing a *three-stage partitioning* of a histogram domain as follows.

The first stage partitions the histogram domain according to the length of the integer permutations, with shorter lengths being assigned partitions with lower indexes in the domain. Then, the size of each of the stage-one partitions can be computed by the following formula (where n is the length of the permutation):

$$sum_n = |L|^n$$

The second stage performs further division of stage-one partitions by grouping all m-length permutations by their *summed ranks*. Those permutations with lower summed rank will have a lower index within a stage-one partition:

$$sr_m = \sum_{i=0}^{m-1} rank(l_i)$$

To compute the boundaries of each of the stage-two partitions, we need to determine how many label paths are in the group with a certain m and sr_m. This question is the same as how many ways there are to distribute sr_m indistinguishable balls over m distinguishable bins of finite capacity $|L|$ with at least one ball in each bin. From combinatorics' *inclusion–exclusion principle* we have:

$$dist(sr_m, m, L) = \sum_{j \geq 0} (-1)^j \binom{m}{j} \binom{sr_m - j \cdot |L| - 1}{m - 1} \quad (3)$$

The third stage explores combinations inside each of the stage-two partitions marked by length m and summed rank sr_m. These combinations are all *integer partitions* of sr_m into exactly m parts, where each part is less than $|L|$. Let integers v, b represent sr_m and $|L|$ respectively. A general formula for integer partition $ip(v, b, m)$ is as follows:

$$ip(v, m, b) = \bigcup_{i=0}^{\lfloor v/b \rfloor} ip(v - i \cdot b, m - 1, b - 1), \underbrace{b, \cdots, b}_{\text{i } bs} \quad (4)$$

Based on Formula 4, we present a partitioning algorithm which outputs all combinations in the desired cardinality-based order and has time complexity is $O(log(|L|)^k)$ [12].

Finally, to compute the boundaries of each of the stage-three partitions, we need to determine how many permutations we skip when we skip a stage-three partition. This is equivalent to identifying how many permutations can be generated by a certain combination in which there might be duplicates. Let C denote the combination, d_i denote the number of times an integer i occurs in C, then the number of permutations is given by the following formula:

$$nop(C) = \frac{|C|!}{\prod_{i \in \{0,\ldots,|L|-1\}} d_i!} \quad (5)$$

Algorithm 1 finds the combination to which the target permutation belongs and has time complexity of $O(k^2)$.

Algorithm 1 Unranking permutation of combination

```
1:  procedure UNRANKING_PERMUTATION(index, C)
2:      if i < 0 ∨ i ≥ nop(C) then
3:          return null
4:      end if
5:      if |C| = 1 then
6:          return [C[0]]
7:      end if
8:      i ← 0
9:      while i < |C| do
10:         S ← C \ [C[i]]                          ▷ subset of C
11:         if index ≥ nop(S) then
12:             index ← index − nop(S)
13:             i ← i + count(C, C[i])
14:             continue
15:         else
16:             sub ← unranking_permutation(index, S)
17:             sub.add(0, C[i])
18:             return sub
19:         end if
20:     end while
21: end procedure
```

Algorithm 2 illustrates the complete version of unranking permutation in sum-based order and has time complexity of $O(log(|L|)^k)$.

3.4 Ordering Example

We illustrate the proposed ordering methods with examples on an artificial dataset which has 3 unique edge labels and its label paths set with k up to 2. Consider the cardinalities 20, 100, and 80 for edge labels "1", "2", and "3", respectively. Then, for the summed ranks shown in Table 1, label paths arranged in the corresponding orderings are shown in Table 2. Respectively, numerical ordering associated with alphabetical ranking, numerical ordering with cardinality ranking, lexicographical ordering with alphabetical ranking, lexicographical ordering with cardinality ranking, sum-based ordering with cardinality ranking are referred to as *num-alph*, *num-card*, *lex-alph*, *lex-card* and *sum-based*.

Algorithm 2 Unranking in sum-based order

```
1:  procedure UNRANKING_IN_SUMBASED(index, L, k)   ▷ index, edge label set, k
2:      if index < 0 ∨ index > |ℒ^k| then
3:          return null
4:      end if
5:      for len ∈ 1, ..., k do
6:          if index ≥ |L|^len then
7:              index ← index − |L|^len
8:              continue
9:          end if
10:         for sum ∈ len, ..., len ∗ |L| do
11:             if index ≥ dist(sum, len, |L|) then
12:                 index ← index − dist(sum, len, |L|)
13:                 continue
14:             end if
15:             P ← ip(sum, len, |L|)
16:             for p ∈ P do
17:                 if index ≥ nop(p) then
18:                     index ← index − nop(p)
19:                     continue
20:                 end if
21:                 p′ ← {i − 1|i ∈ p}
22:                 sort(p′)
23:                 return unranking_permutation(index, p′)
24:             end for
25:         end for
26:     end for
27: end procedure
```

Label Path	1	2	3	1,1	1,2	1,3	2,1	2,2	2,3	3,1	3,2	3,3
Summed Ranks	1	3	2	2	4	3	4	6	5	3	5	4

Table 1: Summed ranks

O \ Index	0	1	2	3	4	5	6	7	8	9	10	11
num-alph	1	2	3	1,1	1,2	1,3	2,1	2,2	2,3	3,1	3,2	3,3
num-card	1	3	2	1,1	1,3	1,2	3,1	3,3	3,2	2,1	2,3	2,2
lex-alph	1	1,1	1,2	1,3	2	2,1	2,2	2,3	3	3,1	3,2	3,3
lex-card	1	1,1	1,3	1,2	3	3,1	3,3	3,2	2	2,1	2,3	2,2
sum-based	1	3	2	1,1	1,3	3,1	3,3	1,2	2,1	3,2	2,3	2,2

Table 2: Ordered label paths according to different ordering methods O

4 EXPERIMENTAL STUDY

We implemented a k-path histogram construction and path selectivity estimation in Java. All experiments are conducted on an Ubuntu 16.04 machine equipped with an Intel i5 CPU with 4GB of RAM. We use the datasets shown in Table 3. The goal of our experiments is two-fold. First, we verify the impact of different domain ordering techniques on the estimation time. Second, we showcase the gains in estimation accuracy which can be obtained by using sum-based histogram domain ordering.

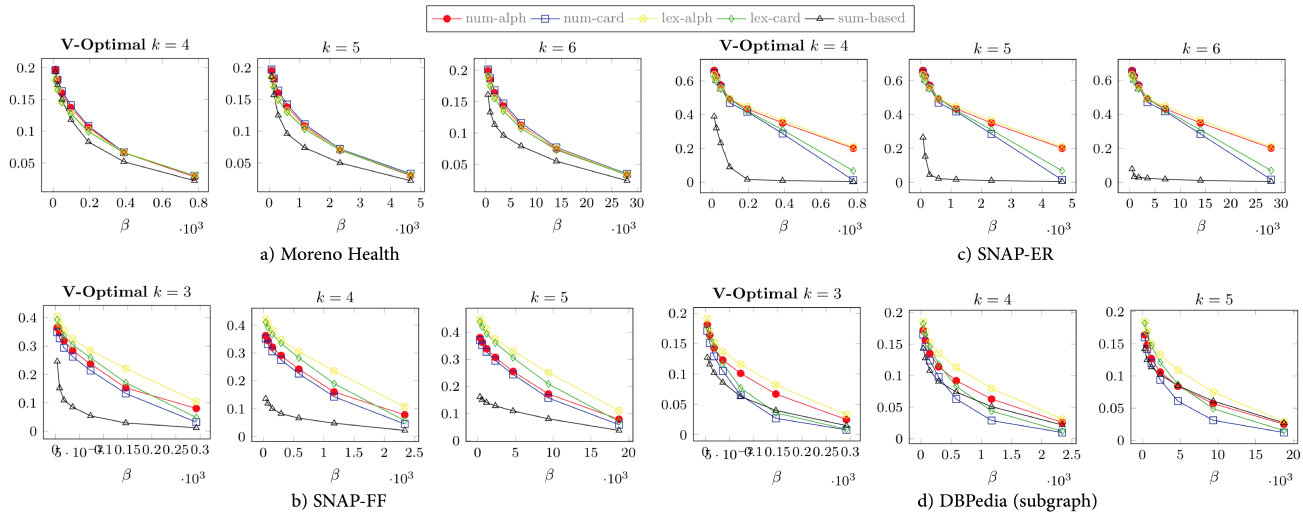

Figure 2: Mean error rate of estimation for different domain ordering techniques on V-Optimal k-path histogram

Dataset	#Edge Labels	#Vertices	#Edges	Real world data
Moreno health[3]	6	2539	12969	yes
DBpedia (subgraph)[4]	8	37374	209068	yes
SNAP-ER[5]	6	12333	147996	no
SNAP-FF	8	50000	132673	no

Table 3: Datasets

β	Average Estimation Running Time (in ms)				
	num-alph	num-card	lex-alph	lex-card	sum-based
27993	9.98	8.62	9.65	8.7	11.02
13996	7.69	7.23	7.79	7.3	9.39
6998	7.36	6.8	7.07	6.93	8.55
3499	6.4	6.52	5.97	6.31	7.42
1749	5.71	5.76	5.76	5.21	6.64
874	5.8	5.06	5.78	5.18	6.1
437	5.19	4.58	4.52	4.29	6.13

Table 4: Average estimation execution time in V-optimal histogram with different ordering methods (in ms)

Performance. We study the execution time of estimation associated with different ordering methods as follows. For $k = 6$, five V-optimal histograms are built, each of which is associated with an ordering method: *num-alph*, *num-card*, *lex-alph*, *lex-card*, and *sum-based*. The total number of label paths is 55996. We run 7 experiments by varying the number of buckets (β) in each histogram. All experiments are executed 100 times and the average estimation time is taken. The results (Table 4) demonstrate that sum-based ordering is approximately 20% slower in estimation than native ordering methods. This is explained by the higher complexity of the sum-based (un)ranking function.

Accuracy. We measure the average estimation accuracy by constructing a V-optimal histogram for each ordering method for varying k and β (Figure 2). We use the following $err(\ell)$ metric to measure the error of an estimation:

$$err(\ell) = \begin{cases} 0 & if \quad e(\ell) = f(\ell) \\ \frac{e(\ell) - f(\ell)}{max(e(\ell), f(\ell))} & else \end{cases} \tag{6}$$

[3]http://konect.uni-koblenz.de/networks/moreno_health
[4]http://wiki.dbpedia.org
[5]https://snap.stanford.edu/snappy/

We observe that, for the synthetic datasets, sum-based ordering provides accuracy which is far superior to other ordering methods, especially, for histograms with a low number of buckets. For the real-life datasets, the performance difference is not as significant, but still observable. This can be explained by the presence of edge-label cardinality correlations in real-life data.

5 CONCLUDING REMARKS

We have reported on initial findings in our ongoing study of domain ordering for improving histogram-based path selectivity estimation. Experimental study has demonstrated the promise of our framework, which facilitates the further systematic study of effective histogram design for graph databases. A primary future research direction is to expand the framework with additional ordering strategies, e.g., those built over richer base sets such as $\mathcal{L}^2$, towards capturing correlations between label paths.

REFERENCES

[1] 2017. openCypher. (2017). https://www.opencypher.org/
[2] A. Aboulnaga, A. Alameldeen, and J. Naughton. 2001. Estimating the selectivity of XML path expressions for internet scale applications. In *VLDB*. 591–600.
[3] Renzo Angles et al. 2018. G-CORE: A core for future graph query languages. In *SIGMOD 2018*. to appear.
[4] G. Bagan, A. Bonifati, R. Ciucanu, G. Fletcher, A. Lemay, and N. Advokaat. 2017. gMark: Schema-driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2017), 856–869.
[5] G. Cormode et al. 2012. Synopses for massive data: samples, histograms, wavelets, sketches. *Found. Trends Data.* 4, 1-3 (2012), 1–294.
[6] George H. L. Fletcher, Jeroen Peters, and Alexandra Poulovassilis. 2016. Efficient regular path query evaluation using path indexes. In *EDBT 2016*. 636–639.
[7] T. Neumann and G. Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE 2011*. 984–994.
[8] Yun Peng, Byron Choi, and Jianliang Xu. 2011. Selectivity estimation of twig queries on cyclic graphs. In *ICDE 2011*. 960–971.
[9] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *PVLDB* 11 (2017), 420 – 431. Issue 4.
[10] Silke Trißl and Ulf Leser. 2010. Estimating result size and execution times for graph queries. In *GraphQ 2010*. 11–20.
[11] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *GRADES 2016*.
[12] Li Wang. 2017. *On histograms for path selectivity estimation in graph data.* Master's thesis. Eindhoven University of Technology.
[13] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. 2003. Using histograms to estimate answer sizes for XML queries. *Inf. Syst.* 28, 1-2 (2003), 33–59.