Histogram domain ordering for path selectivity estimation

Citation for published version (APA):

DOI:
10.5441/002/edbt.2018.55

Document status and date:
Published: 01/01/2018

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 10. Jun. 2022
Histogram Domain Ordering for Path Selectivity Estimation

Nikolay Yakovets, Li Wang, George Fletcher
TU Eindhoven, Netherlands
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.
histogram, is an ordered label path sequence produced by an ordering of $L^k$. Then, given label path $\ell$ and its index $\text{index}(\ell)$, 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 $\text{index}(\ell)$ 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 $|L|$ index values. The exact amount of memory can also be used to store a base label set $B$ which places a base label with lower cardinality in front of the label with higher cardinality, i.e., $l_1 < \text{card} l_2 \iff f(l_1) < f(l_2)$.

An ordering method can be described by the following three components. First, we need a base label set $B$. Second, we define a (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 $L^k$. It is a bijection that maps between label path set $L$ and integer set $[0, |L^k|]$. 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 $A$-$B$ 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 higher cardinality in front of the label with lower cardinality, i.e., $l_1 < \text{card} l_2 \iff f(l_1) < f(l_2)$.

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: $\text{alph}$ and $\text{card}$. Let $\text{alph}(l)$ and $\text{card}(l)$ 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):

\[
\ell_1 < \ell_2 \iff \begin{cases} |l_1| < |l_2| & |l_1| \neq |l_2| (1) \\
\wedge_{j=1}^{i-1} (l_1^{(j)} = l_2^{(j)}) \land (l_1^{(i)} < l_2^{(i)}) & |l_1| = |l_2| (2)
\end{cases}
\]

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 $\forall l \in L, \text{rank}(\ell) > \text{rank}(l)$) to every $\ell$ 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 $\ell$, the idea of sum-based ordering is to use the sum of ranks of all base labels in $\ell$ to approximate the cardinality of $\ell$. While being conceptually simple, the implementation of this ordering method is not trivial. First, given a path label $\ell$ of length $k$, $\ell$ 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 $\ell$ is mapped to $\text{index}(\ell)$ 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):

\[
\text{sum}_n = |L|^n
\]

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:

\[
\text{sr}_m = \sum_{i=0}^{m-1} \text{rank}(l_i)
\]

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 $\text{sr}_m$. This question is the same as how many ways there are to distribute $\text{sr}_m$ 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:

\[
\text{dist}(\text{sr}_m, m, L) = \sum_{j \geq 0} (-1)^j \binom{m}{j} \left( \text{sr}_m - j \cdot |L| - 1 \right) / m - 1
\]
The third stage explores combinations inside each of the stage-
two partitions marked by length \( m \) and summed rank \( s_{rm} \). These combinations are all integer partitions of \( s_{rm} \) into exactly \( m \) parts, where each part is less than \( |L| \). Let integers \( v, b \) represent \( s_{rm} \) and \( |L| \) respectively. A general formula for integer partition \( ip(v, b, m) \) is as follows:

\[
ip(v, b, m) = \sum_{i=0}^{b} ip(v - i \cdot b, m - 1, b - 1, \ldots, b)
\]

Based on Formula 4, we present a partitioning algorithm which outputs all combinations in the desired cardinality-based order and has time complexity is \( O(\log(|L|)^3) \) [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:

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

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 \( i < 0 \lor i \geq \text{nop}(C) \) then
3. return null
4. end if
5. if \( |C| = 1 \) then
6. return \([C[0]]\)
7. end if
8. \( i \leftarrow 0 \)
9. while \( i < |C| \) do
10. \( S \leftarrow C \setminus [C[i]] \) \( \longrightarrow \) subset of \( C \)
11. if \( \text{index} \geq \text{nop}(S) \) then
12. \( \text{index} \leftarrow \text{index} - \text{nop}(S) \)
13. \( i \leftarrow i + \text{count}(C, C[i]) \)
14. continue
15. else
16. \( \text{sub} \leftarrow \text{unranking_permutation}(\text{index}, S) \)
17. \( \text{sub.add}(0, C[i]) \)
18. return \( \text{sub} \)
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 \( O(\log(|L|)^3) \).

**Algorithm 2** Unranking in sum-based order

1. procedure **unranking_in_sumbased**(index, \( L, k \)) \( \rightarrow \) index, edge label set, \( k \)
2. if \( \text{index} < 0 \lor \text{index} > |L|^k \) then
3. return null
4. end if
5. for \( len \in 1, \ldots, k \) do
6. if \( \text{index} \geq |L|^{len} \) then
7. \( \text{index} \leftarrow \text{index} - |L|^{len} \)
8. continue
9. end if
10. for \( sum \in \text{len} \) do
11. if \( \text{index} \geq \text{dist}(\text{sum}, \text{len}, |L|) \) then
12. \( \text{index} \leftarrow \text{index} - \text{dist}(\text{sum}, \text{len}, |L|) \)
13. continue
14. end if
15. \( P \leftarrow ip(\text{sum}, \text{len}, |L|) \)
16. for \( p \in P \) do
17. if \( \text{index} \geq \text{nop}(p) \) then
18. \( \text{index} \leftarrow \text{index} - \text{nop}(p) \)
19. continue
20. end if
21. \( p' \leftarrow (i - 1 | i \in p) \)
22. sort(p')
23. return unranking_permutation(index, p')
24. end for
25. end for
26. end for
27. end procedure

<table>
<thead>
<tr>
<th>Label Path</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summed Ranks</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 1: Summed ranks

| O | Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|-------|---|---|---|---|---|---|---|---|---|---|---|---|---|
| num-alph | 1 | 2 | 3 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 1.10 | 1.11 |
| num-card | 1 | 2 | 1.1 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 1.10 | 1.11 |
| lex-alph | 1 | 1.1 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 1.10 | 1.11 | 1.12 |
| lex-card | 1 | 1.1 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 1.10 | 1.11 | 1.12 |
| sum-based | 1 | 2 | 1.1 | 1.2 | 1.3 | 1.4 | 1.5 | 1.6 | 1.7 | 1.8 | 1.9 | 1.10 | 1.11 |

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

orders 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.

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.

495
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: \textit{num-alph}, \textit{num-card}, \textit{lex-alph}, \textit{lex-card}, and \textit{sum-based}. The total number of label paths is 55996. We run 7 experiments by varying the number of buckets ($\beta$) 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 time compared to the other 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.

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 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 $L^2$, towards capturing correlations between label paths.

REFERENCES