MASTER

On histograms for path selectivity estimation in graph data

Wang, L.

Award date:
2017

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Web Engineering Research Group

On histograms for path selectivity estimation in graph data

MSc Thesis

Li Wang

Supervisors:

dr. Nikolay Yakovets
dr. George Fletcher

External Supervisors:

prof. dr. Alex Poulovassilis (Birkbeck University of London)
Craig Taverner (Neo Technology)

Assessment Committee Members:

dr. Nikolay Yakovets
dr. George Fletcher
dr. Wouter Duivesteijn

Eindhoven, the Netherlands, August 2017

Abstract

In this thesis, we study the problem of path selectivity estimation in graphs with the goal of low error rate, limited memory consumption, and high execution speed.

Specifically, we focus on the k-path index histogram approach. The research is carried out in two parallel directions. One implements and tests different histogram schemes. The other one brings in the ordering concept in describing a data distribution as well as its histogram. In this process, we introduce the *ideal* ordering which gives the lower bound of the error rate of estimation by arranging label paths by their frequencies. This method is used only to define the lower bound, and not as a suggested solution to ordering due to the need for high memory to maintain the ordering definition. We also present an innovative ordering framework, *data aware*, and one of its instances, *sum-based* ordering, to approximately construct a monotonic sequence which consumes much less memory than ideal ordering.

We carefully design our implementation such that the two parallel directions can be developed independently and merged to run experiments together seamlessly. We run experiments on various artificial and real-world datasets and compare the results to see the performance of combinations of schemes and ordering methods. Based on the results, we describe the characteristics of data set that gives the best estimation from the perspective of the histogram and the ordering method.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

As a result of the increase of connectivity and availability of computers, data across the Internet grows rapidly with respect to its volume, velocity, and variety. The optimization of databases has to take place in order to maintain the increased workload. Additionally, more and more data is presented in the form of graph data, such as social networks [13], biological networks [10], [9] and Linked Open Data [17]. Traditional data models associated with relational database management system that handles connections with foreign keys and join operations have trouble in processing graph data, as the deep traversals in processing the graph data cannot scale with such models. Consecutively in the recent decade, graph databases that deal with graph datasets are becoming increasingly important and irreplaceable in various scenarios. For example, Neo4j[15] is a widely used open source graph database.

However, the techniques in graph databases are not as mature as they are in the relational database, it is an active field of both academic research and industry. In this work, we are interested in query optimization techniques. When executing a complex query in the database, query optimization selects the most efficient query evaluation plan from among many possible strategies.

Typically, query optimization involves three steps: 1) parse the query into an abstract syntax tree of normalized expressions, 2) convert the abstract syntax tree into a set of possibly many alternative trees of logical plan operators (steps of solving parts of the plan), 3) estimate the cost of all operators, and from that the cost of all possible complete plan solutions, and select the plan with the least estimated cost [16]. Our study provides the estimation of a path query with high accuracy to query optimizer, such that it is within the scope of step 3 and focuses on the general idea of the path selectivity estimation in a graph database, which shares fundamental principals with traditional databases but also has its unique characteristics.

Estimation techniques generally follow this basic principle: the whole dataset is scanned beforehand, and the statistical data of the dataset is calculated and stored in a specially designed data structure. This statistical data is later used by the corresponding algorithm to estimate the cardinality of the query. The cardinality estimation structures and algorithms are expected to have the following features:

- Low memory consumption
- Short running time
- High accuracy in estimation

It is always the case that more accurate estimation can be obtained with more information captured in memory. Hence, there exists a trade-off which the user makes depending on the scenario at hand.

1.1 Contribution

The main contributions of this thesis are:

- We implement three types of histograms which can efficiently give an estimation of path query selectivity. Their results are compared and their features are discussed in depth.
- We introduce an *ordering* concept in describing the data distribution and using a histogram to estimate label path selectivity.
- We design a framework in which histogram scheme and ordering method are fully decoupled and transparent to each other. This allows them to be studied, developed, and tested separately in practice.
- We present a new ordering rule which produces better estimation than numerical order (Section 5.2.3) and lexicographical order (Section 5.2.4).
- We conduct an extensive empirical study and present and discuss its results in detail.

1.2 Outline

The rest of this thesis is organized as follows. In Chapter 2, we identify the research question. In Chapter 3, we summarize the existing approaches in the cardinality estimation field. In Chapter 4, we present the histogram framework and three histogram schemes. Chapter 5 first describes the motivation behind the ordering concept and discusses the optimal ordering. We then present the framework in describing an ordering method. Furthermore, we show the different ordering approaches in our research and the algorithms and formulas to construct them. In Chapter 6, we discuss more details in design and implementation phase. In Chapter 7, we talk about the dataset, present and discuss the experimental results, introduce the possible best practices from the perspective of histogram and ordering method. Finally, Chapter 8 presents conclusions and future work.

Chapter 2

Problem Formulation

2.1 Paths in a graph

The goal of our investigation is to estimate the selectivity of a **path query** in a **directed labeled graph**, which is composed of a set of vertices V, a collection of directed edges E and a set of unique edge labels L.

Each directed edge connects an ordered pair of vertices and is marked by a label, this can be denoted by $e = (v_s, l, v_t)$ with l is the label and (v_s, v_t) is the ordered vertex pair, indicating a one-step relationship goes from the source vertex v_s to target vertex v_t, i.e., $E \subseteq V \times L \times V$.

A **k-label path** is a sequence $\ell = \langle l_1, ..., l_k \rangle$, where $l_i \in L$, for all $1 \leq i \leq k$, We define k as the **length** of ℓ, denoted by $|\ell|$. The sequence itself will be referred to as the **label path identifier**. Given such a path, path query specifies the sequence of edge labels it takes along the path and returns all pairs of vertices $\{v_s, v_t\}$, for each pair there exists vertices $v_0, v_1, ..., v_k \in V$ and labels $l_1, l_2, ..., l_k \in L$ such that $v_s = v_0$, $v_e = v_k$. and for $0 < i \leq k$,

$$(v_{i-1}, l_i, v_i) \in E$$

We here clarify that cycles are allowed in this problem. The **selectivity** of a path query is the number of all possible pairs that satisfy such a query.

It is noted that the path query is categorized as a point query rather than a range query [2]. Although path query with wild card in a path identifier can be seen as a range query, our project focuses mainly on a path query of a single path length.

Let $\mathcal{L}^k = \{\ell | \ell \text{ is a sequence where length is at most } k\}$ be the set of all label paths with length up to k. From the definition, we know $L \subset \mathcal{L}^k, L = \mathcal{L}^1$. The size of $\mathcal{L}^k$ can be calculated by the following formula:

$$|\mathcal{L}^k| = \sum_{i=1}^{k} |L|^i \tag{2.1}$$

which is the maximum value of k-length integer of an $|L|$-based numeral system. In further discussion, if not specified, $\mathcal{L}$ is used to represent the label path set in a general state regardless of k .

2.2 Ordering and Data Distribution

An **ordering** of $\mathcal{L}^k$ is a bijection which maps between label path set $\mathcal{L}^k$ and integer set $[0, |\mathcal{L}^k|)$.

Given ranking method *rank* of an ordering, let ℓ_i denote the label path ℓ such that $rank(\ell) = i$, i is also its positional index in this ordered label path sequence. Once we establish an ordering on a label path set, a label path can be represented by its index. In Section 5.2.3, we introduce the *num-alph* ordering, i.e. *numerical ordering* rule associated *alphabet ranking* rule. This ordering approach is seen as the default ordering of a label path set.

For each label path ℓ, we define the following properties in order to discuss the data distribution in later sections:

- **value:** its index value in the label path sequence, i.e. $rank(\ell)$
- **frequency:** $f(\ell)$ is the corresponding cardinality of each label path, i.e., the selectivity of this query.
- **estimation:** $e(\ell)$ is the estimated frequency of label path ℓ.
- **error rate:** $err(\ell)$ is the metric used to measure the quality of an estimation mechanism. Its value is defined as follows:

$$err(\ell) = \begin{cases} 0 & if \quad e(\ell) = f(\ell) \\ \frac{e(\ell)-f(\ell)}{max(e(\ell),f(\ell))} & else \end{cases} \tag{2.2}$$

Definition 2.2 handles the situation of perfect estimation and avoids the division-by-zero problem, and gives a balanced range [-1,1] to all situations including under-estimated and over-estimated, and makes them distinguishable by marking them with the negative or positive sign, respectively. We use the error rate result directly when analyzing each case separately, and use its absolute value, i.e. $|err(\ell)|$ when investigating them together.

2.3 Example

We take the graph G_e in Figure 2.1 as an example, which consists of V_e={James, Kate, Jane, Kevin, Jason, Mark}, L_e={follows, is friend of, likes}.

A path query $\ell_1 = \langle$ likes, is friend of$\rangle$ on G_e gives {(Kate, James),(Mark, Kate)} as the source-target pairs, and $\ell_2 = \langle$ likes, is friend of, follows$\rangle$ on G_e gives {(Kate, Mark),(Mark, Jane)}, which means $f(\ell_1) = 2$, $f(\ell_2) = 2$. Suppose we have estimations of these two label path $e(\ell_1) = 1$ and $e(\ell_2) = 4$, by applying Formula 2.2, we have the corresponding error rate $err(\ell_1) = -0.5$ and $err(\ell_2) = 0.5$.

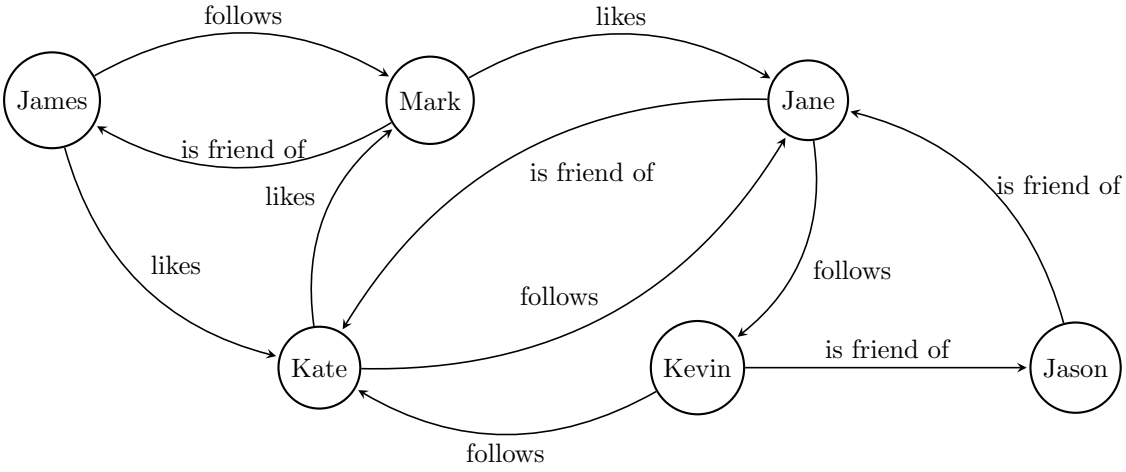


Figure 2.1: Simple social network example

2.4 Research Question

After having all necessary terms and concepts defined, we identify the following research question:

"Given a graph G and an integer k, how to use histogram to produce frequency estimations with lowest error rate for all paths with length up to k in an efficient way, by using no more than M bit memory?"

This description covers all three features that a good histogram-based estimation technique is expected to have.

In our study, to simplify things, all the properties in a graph will be represented as integers, namely: vertex name, source vertex and target vertex of an edge, and edge label.

Further, our research focuses on a k-path histogram described in [6] and aims to boost its performance in terms of its estimation accuracy.

Chapter 3

Literature Review

To start this research, we studied several existing cutting edge approaches in solving query estimation in different kinds of databases.

3.1 Histograms framework

In [7], Ioannidis discusses the histogram approach in depth, describes the origin, the evolution, and the future of histograms. That's everything we need to know about histograms. This is seen as our technical reference of the histogram approach. More importantly, it gives the framework of defining a histogram. We adopt this framework in Section 4.1.1 in describing three histogram schemas in our research.

3.2 Path indexing in graph databases

In [6], Fletcher, Peters, and Poulovassilis describe their research on the path index. Given a directed labeled graph $G(V, E, L)$ and a natural number k, let $paths_k(G)$ denote the set of all pairs of vertices $(v_s, v_t) \in V \times V$ such that there is an i-path from v_s to v_t, for some $i \leq k$. Set $paths_k(G)$ is indexed by using an ordered *k-path index* $I_{G,k}$ having search key $\langle labelpath, sourceID, targetId \rangle$. Given a non-empty prefix p of a search key, $I_{G,k}$ returns an ordered list of all matching entries.

The query estimation solution in their work is served by a basic k-path histogram instance. However, the low accuracy of its estimation remains as an issue.

3.3 Estimating the Selectivity of XML Path Expression

Path tree is introduced by Aboulnaga, Alameldeen, and Naughton in [1] in estimating the selectivity of XML Path Expressions queries. It is a tree that represents the structure of an XML document The root of the path tree is the root of this document, and all XML elements directly nested in root element with the same tag are merged into one child node of the root node. The tag name is used to label the node and the number of the elements is recorded as the *frequency* of the node. This keeps going until it reaches leaf elements which become the leaves in path tree.

In case a path tree is consuming too much memory, it needs to be summarized. Several methods are also introduced in this paper to cut a path tree while retaining as much original information as

possible. The first one is call sibling-*, which is to merge the siblings with low *frequency* into one node labeled by *. It will refer this kind of nodes as *-node and the nodes not labeled with * as regular nodes. The *frequency* will be the sum of the *frequency* of all merged nodes. Additionally, it will record the number of nodes that get merged, their children will be children of the *-node. After this operation, there could be more than one siblings share the same name. In this case, siblings with the same name will be merged into one which will keep its name. The *frequency* will also be updated to the *frequency* of all nodes that get merged. The number of nodes that get merged is recorded as well.

The structure is used in estimating as follows: Given a path expression, it tries to find all end nodes which a path with tags sequence that matches the sequence in the expression leads to, the selectivity estimation is the total *frequency* of all end nodes. In this process, the *-nodes can be seen as a wild card to match any tag. For example the path expression //A/B/C would match all of //A/*/C or //*/*/C, but not //*/*/*, generally it matches with any number of *-nodes as long as there's at least one regular node in a match, a match with nothing but *-nodes will be considered as invalid because of lack of confidence.

As the interest of our research mainly focuses on path query on a path query of a single path length, many optimizations in path tree are regarded as useless in our context.

3.4 Estimating Result Size and Execution Times for Graph queries

In [20], Tribl and Leser discussed estimation based on simple graph properties, namely, the number of nodes, the number of edges, the highest out-degree, the number of nodes with 0 out-degree. Two index structures are used in their paper, namely Travel Closure (TC) and GRIPP.

Travel Closure is essentially a matrix recording in which the crossing of a row and a column represents if a pair of vertices are connected. As it records no intermediate nodes and works only for unlabeled graphs, we think it's less related to our problem.

GRIPP is detailed described in [19]. Unlike TC, it does record intermediate nodes to solve path queries. Although it doesn't issue labeled graph either, the result of it turning a graph into a tree is still worth paying attention and thus has the potential to be our candidate of possible approaches.

3.5 Using Positional Histograms to estimate answer sizes for XML queries

In this paper[21], a novel histogram approach is proposed by Wu and Patel. In query estimating, the histogram is already a popular structure recording statistics data, in XML context, it is also possible to be used to record positional information.

To do that, the first step is to associate a numeric **start** and **end** label with each node, the tree is traveled in pre-order, each node labels the **start** with the pre-order number, in case the node is a leaf, the **end** will have the same number as its **start** label, if that's not the case, then it will be labeled with a number larger than the maximum of its descendant's **end** label. By doing this, the interval between labels is an area such that every node with **start** label within this area is a descendant of this node.

Once all nodes are labeled, it starts building this positional histograms which are essentially two-dimensional matrices, there will be one matrix for each predicate α in set **P** of predicates of

interest, denoted by $Hist_\alpha$. Each matrix is divided into n_2 grid cells where n is the parameter it uses to control the memory size consumed by the structure, each grid cell in this matrix represents ranges of start and end position values and each document node is mapped to a point that falls into some grid cell in a matrix, if a node satisfies α then the corresponding cell in $Hist_\alpha$ will increase the number it maintains by 1.

The feature **start** $\leq$ **end** leads to the fact that the bottom-right region of a matrix is always empty. Also, note that the **start** and **end** ranges of any two nodes can be either one range fully contains the other, or have no intersection at all, which means for a node satisfying α, the point to which it is mapped has the coordinates (x,y), the grid cell with bottom-left vertex (0,x), top-right vertex (x,y) and grid cell with bottom-left vertex (x,y), top-right vertex (y,0) is guaranteed to be empty. Both of the two features discussed above contribute to more accurate estimate by eliminating empty space, this could also be useful in our problem.

3.6 Sampling from Large Graphs

In [11] Leskovec, Faloutsos introduces several algorithms on how to do sampling on large graphs, and also describes several algorithms in each of the categories. Among the two different purposes described in that paper we figure the scale-down sampling is more related to our problem, there are 9 metrics it uses to measure how much resemblance the sample has to the whole graph, for example, in-degree and out-degree distribution, distribution of weakly connected components and strongly connected components, and so on, and D-statistics is used to measure the agreement between two distributions.

The algorithms can be categorized into three types, namely: random node selection, random edge selection, and random walk exploration. As the names suggest, generally, three methods are to randomly select nodes from the original graph, randomly select edges from the original graph, and firstly randomly select nodes as start points, then select edges that connect the nodes.

Chapter 4

Histograms

In the database context, a data distribution is a listing of value-frequency pairs of some attribute. *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 whole 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 each bucket we have the following attributes:

- start index
- end index, (or the total number of tuples. Given the start index, these two can be used to calculate each other)
- the sum of frequencies

Depending on the partitioning methods, a corresponding approximating algorithm is used to approximately restore the frequency and value for a given element.

The list below illustrates the possible key issues of a histogram and how can they be addressed:

- How to define a histogram?
  - Bucket scheme
- How to build a histogram?
  - Construct algorithm
- How to use a histogram?
  - Value approximation
  - Frequency approximation
- How good is a histogram?
  - Error guarantee
  - Construction running time
  - Dynamic update
  - Value approximation running time
  - Frequency approximation running time

4.1 The properties in our project

To use a histogram to estimate selectivity, we need to fulfill the memory constraint described in section 2.4. To cover this issue, we introduce several constants at the beginning:

- m is the memory consumed by each bucket. Assuming a 64-bit data type, we need $64+2*32$ bits per bucket, for two 32-bit integers as boundary information and a 64-bit long integer as statistical information.
- β is the total number of available buckets

$$\beta = M/m \tag{4.1}$$

4.1.1 Bucket schema

In this section, several important characteristics of the histogram and their default values are introduced. These properties have their origins in [7] and [3]. Each parameter or the combination of several parameters restrict, define and describe a histogram from a certain perspective, and all of them together define a unique histogram schema. The structure will be used in later sections as a skeleton of histogram definition. Furthermore, we identify the setting of those properties that have the same setting in all histograms implemented in our projects, namely, Partition Class, Sort Parameter, Source Parameter, Value Approximation and Frequency Approximation. **Partition Constraint**, **Construction Algorithm**, and **Error Guarantee** to be specified for each histogram schema.

Partition Class

This parameter defines the possible restrictions on buckets. In our project, all histogram are *serial* [7]. Together with the next parameter **Sort Parameter**, the restrictions require the buckets to be not overlapping each other with respect to Sort Parameter, and the Sort Parameter values within each bucket to form a contiguous range.

Sort Parameter

As specified in the last paragraph, this parameter defines how the sequence is ordered. In our project, this role is always served by *value* property, which is introduced in section 2.2.

Source Parameter

The cardinality of each label path, which is $frequency$.

Partition Constraint

This is a mathematical constraint on source parameter or sort parameter that uniquely identifies a single bucket. An ideal constraint groups source parameters similar in value into one bucket. This parameter varies for different schemes.

Construction algorithm

This is the concretization of **Partition Constraint**, usually in the form of description or pseudo-code according to which a histogram can be built.

Value approximation

The mechanism that approximately reconstructs the value distribution with information stored within a bucket. In our project, no approximation is needed as the values are all exact.

Frequency approximation

The mechanism that approximately reconstructs the frequency distribution with information stored in a bucket. In our project, we always make the *uniform distribution assumption* within each bucket, which calculates the estimation by the following formula:

$$e(\ell_j) = \frac{\sum_{\ell_i \in B(\ell_j)} f(\ell_i)}{|B(\ell_j)|} \tag{4.2}$$

where $B(\ell_j)$ is the set of label paths that locate in the same bucket as the jth label path does. Additionally, we will discuss *Label-based biased frequency approximation* in section 8.

Error Guarantees

This indicates whether partition rule or Frequency approximation provides an upper bound on the error rate of estimation.

4.2 Equi-width Histogram

Partition Constraint

The whole *value* range is serially and uniformly divided into β buckets. Each bucket stores the same number of tuples. The number is referred to as the *width* of the histogram.

Construction algorithm

Algorithm 1 Equi-width Histogram

```
procedure EQUI_WIDTH_HISTOGRAM(L, β)          ▷ distribution, number of buckets
    buckets ← {}
    n ← |L|/β                                 ▷ ignore the extra bucket for simplicity
    for i ∈ {0, ..., β − 1} do
        b ← Bucket()
        b.sum ← 0
        buckets.push(b)
        for j ∈ {i × n, ..., i × n + n − 1} do
            b.sum ← b.sum + f(ℓ_j)
        end for
        b.start ← i × n
        b.end ← i × n + n − 1
    end for
    return buckets
end procedure
```

The algorithm travels the label path set in order for once, hence the time complexity is $\mathcal{O}(|\mathcal{L}|)$.

Error Guarantees

No error guarantee, the quality depends on the structure of data distribution.

4.3 Equi-depth Histogram

Partition Constraint

The basic constraint for this approach is to arrange the label paths in such a way that the total frequency within each bucket, known as the $depth = \frac{\sum_{\ell \in \mathcal{L}} f(\ell)}{\beta}$, will be the same value.

Construction algorithm

Although it is designed in a neat and elegant way, the real situation is always more complicated, as it is rarely the case that the total frequency of a bucket is exactly the same value as *depth*. A bucket keeps taking in label path until its total frequency is equal to or larger than *depth*, that is, a bucket stores at most one more label path than it should.

During the implementation and test, we notice that sometimes it could be the case that the frequency of a single label path can exceed the *depth* of a bucket. This clearly produces estimation with a high error rate, as the high-frequency label path shares bucket with other ones with relatively low cardinality, it will be overwhelming and dominating within that bucket. This leads to an improvement of this algorithm, which allocates that single label path to a new exclusive bucket.

The construction time complexity is $\mathcal{O}(|\mathcal{L}|)$.

Algorithm 2 Equi-depth Histogram

```
procedure EQUI_DEPTH_HISTOGRAM(L, β)
    buckets ← {}
    depth ← (Σ_{ℓ∈L} f(ℓ)) / β
    for i ∈ {0, ..., |L| − 1} do
        b ← Bucket()
        buckets.push(b)
        b.start ← i
        b.sum ← 0
        while b.sum < depth do
            if b is not empty and f(ℓ_i) > depth then
                break
            else
                i ← i + 1
                b.end ← i
                b.sum ← b.sum + f(ℓ_i)i
            end if
        end while
    end for
    return buckets
end procedure
```

Error Guarantees

Since the total frequency for all buckets are the same, we can bound the maximum error by the following expression:

$$1 - \frac{\min_{j=1}^{|\mathcal{L}|} f(l_j)}{depth}$$

4.4 Variance Optimal (V-optimal) Histogram

Partition Constraint

Given a fixed number of β, limit the intra-bucket variance within each bucket as much as possible by varying the boundaries between buckets.

Construction algorithm

Algorithm 3 Greedy V-optimal Histogram

procedure GREEDY_V-OPTIMAL_HISTOGRAM($\mathcal{L}, \beta$)
  $buckets \leftarrow \{\}$
  for $i \in \{0, ..., |\mathcal{L}| - 1\}$ **do**
    $b \leftarrow Bucket()$
    $b.sum \leftarrow f(\ell_i)$
    $b.start \leftarrow i$
    $b.end \leftarrow i$
    $buckets.push(b)$
  end for
  $err \leftarrow 0$ ▷ starts with perfect estimation, the total error is 0
  while $|buckets| > \beta$ **do**
    $min \leftarrow MAX\_DOUBLE$
    $j' \leftarrow 0$
    for $j \in \{0, ..., |buckets| - 2\}$ **do**
      $b' = merge(buckets[j], buckets[j+1])$
      $err' \leftarrow err - \sum_{\ell \in buckets[j]} err(\ell) - \sum_{\ell \in buckets[j+1]} err(\ell) + \sum_{\ell \in b'} err(\ell)$[1]
      if $err' < min$ **then**
        $min \leftarrow err'$
        $j' \leftarrow j$
      end if
    end for
    for $t \in \{buckets[j'+1].start, ..., buckets[j'+1].end\}$ **do**
      $buckets[j'].sum \leftarrow buckets[j'].sum + f(\ell_t)$
    end for
    $buckets[j'].end \leftarrow buckets[j'+1].end$
    $buckets.remove(j'+1)$ ▷ see buckets as a list for simplicity
    $err \leftarrow min$
  end while
  return $buckets$
end procedure

To reduce the intra-bucket variance, the first thing we can think of is to assign an exclusive bucket for each label path, such that we have 0-variance within each bucket and perfect estimation for all label paths. Of course, this is not feasible, we need to fulfill the memory restriction by merging adjacent buckets into one. The most effective way in terms of reducing error rate is to make sure the merge operation produces the least error rate. A greedy version of construction algorithm can be built based on the idea above.

The algorithm is to start with N buckets with one label path in each bucket, we have $N - 1$ adjacent bucket pairs, compute and compare the error rates generated by all pairs when two buckets in each pair are merged into one bucket, we choose the pair that produces the least error and execute the merge, this is called a step. The number of buckets reduces by 1 each time a step is finished, we keep doing this until the number reaches β.

A drawback of this schema is it takes too much time to construct. It needs $\mathcal{L} - \beta$ steps to build the histogram, and in each step $n - 1$ pairs of buckets are merged and compared while n is the number of buckets at that time. Therefore the building time complexity of V-Optimal histogram is $\mathcal{O}(|\mathcal{L}|^2)$.

However, this disadvantage is not one of the measurements. In practice, this can be covered by using high-performance CPU and more memory for construction. Once the histogram is built, it can be delivered and used in producing estimation.

Error Guarantees

This type of histogram is expected to have better quality than both Equi-width and Equi-depth. Although we cannot calculate its boundary for error rate yet, the experiment results in section 7.4 shows that this is indeed the case.

4.5 Example

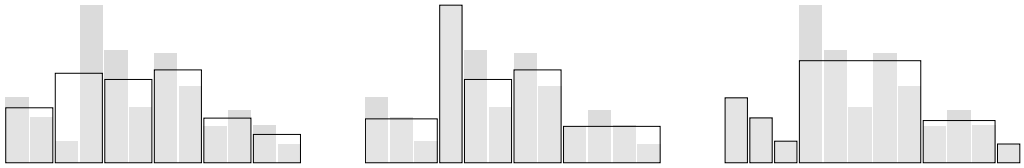
To understand these three histogram schemas more clearly, and to reveal a behavior that is not easily seen, we present a simple example with visualization. The dataset in this example has 3 unique edge labels. With $k = 2$, its data distribution has 12 label paths in total which are put into 6 buckets. Figure 4.1 illustrates the distribution and histogram constructions. The light bars represent the frequencies of label paths and transparent rectangles with dark borders reflect buckets. The estimation of each label path is given by the height of bucket in which it locates.

Equi-width histogram organizes buckets in a straightforward way. In this example, the *width* is 2. The *depth* property of Equi-depth histogram is represented by the size of the area of the transparent rectangle. It is not easy to identify V-optimal histogram at a glance. However, with a closer observation, it can be distinguished by the feature that label path with low cardinality can be assigned to an exclusive bucket, which will never occur in the other two histograms.

Another interesting fact we notice about Equi-depth histogram is it only uses 5 buckets instead of 6. This first draws our attention in our unit testing process when we specify the number of available buckets b, which consequently specifies the *depth*. After construction, we observe the phenomena that the number of actually used is usually less than b, i.e. there are buckets left unused.

According to Algorithm 2, there are two reasons for this. 1) When a label path whose cardinality is larger than the *depth* of a histogram, it is assigned to an exclusive bucket. This bucket takes more load than it should. This applies to all exclusive buckets. 2) For the other buckets, they

[1]Here three sums of $err(\ell)$s are given by the corresponding buckets, namely, two original and one merged



(a) Equi-width (b) Equi-depth (c) V-optimal

Figure 4.1: Simple example of histogram

always store one more label path than they should, which is, they are also overloaded. As all buckets are overloaded, it actually needs fewer buckets to store all the label paths. This feature of Equi-depth histogram will be referred to as *saving-bucket*.

Chapter 5

Ordering

5.1 Motivation

From Figure 7.2 we can see the results of naive equi-width and equi-depth histogram are both not good enough. In the process of analysis, by observing and examining the bar chart of a data distribution with small k, we notice that the high variance of the fluctuating frequencies within each bucket is the main reason for poor accuracy. We use the term **Intra-bucket variance** to denote this property.

The first solution we attempt is to use an advanced histogram schema (see Section 4.4) to produce the estimation, which indeed improves the quality to a certain extent (its result will be discussed in detail in Section 7.4). However, when we inspect the data distribution and the structure of bucket in more depth, we notice it is the case that when a data distribution is sharply fluctuating, no matter how good a scheme is, the estimation will always have poor quality in terms of error rate. But by re-arranging some of the sequences in an appropriate way, we can have a better result. This is how the *ordering* concept emerges into our vision.

For example, imagine we are processing a label path set $\mathcal{L}$, $f(l_i) = 10000, \forall\ index(l_i) \bmod 2 = 0$, and $f(l_j) = 1, \forall\ index(l_j) \bmod 2 = 1$, it can be easily seen that with equal number of 1's and 10000's within each bucket, the average value of frequency is always 5000.5. For label path whose frequency is 10000, it is a bad estimation with 0.5 error rate, for the ones whose frequency is 1, it is a disaster. To reorder this sequence, we can use a simple formula

$$r(i) = \frac{(i \bmod 2) * |\mathcal{L}| + i}{2}$$

to build a bijection between two identical integer set $[0, |\mathcal{L}|)$ to locate label path with even index first. Now let us analyze this reordering approach to see which features it has, and which changes it brings to the data distribution.

Although we rarely see such an extreme example in practice, it helps us to notice the **ordering** concept of a data distribution in this scenario. The ordering we currently use can be described as **Numerical Ordering**, which is formally defined in Section 5.2.3. Numerical Ordering naturally follows the process in which we evaluate estimations for all label paths up to length k. Producing all label paths is the same as generating all possible combinations of character set L.

5.1.1 Ideal Ordering

The purpose of reordering is to generate a sequence in which 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 in such a way that when $index(\ell)$ increases, $f(\ell)$ monotonically increases or decreases. The most straightforward, yet not feasible, approach is to sort the sequence by its cardinality and assign the position number to each label path as its *index*. This idea is not practical because 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 the estimation of selectivity, we can obtain the precise selectivity.

Though this method can not be applied in practice, it still gives theoretical significance to this problem. We can see the error rates it produces with all the constraints as the **Lower Bound** of the error rates that any type of histogram scheme produces with the same restrictions. This ordering method will be referred to as the **ideal ordering**.

5.2 A Novel Framework and Different Ordering Instances

In this section, we present an innovative framework in defining an ordering over a label path set $\mathcal{L}$ and describe several different ordering methods in this pattern.

5.2.1 Base Label Set

From above discussion, we establish that sorting label paths by cardinalities is not an option in practice because of the massive memory consumption. However, we can still construct an approximately monotonic sequence based on the awareness of precise cardinalities of a subset of $\mathcal{L}$. By looking at Figure 5.1, we notice that the label *1* has the highest cardinality among all length-1 label paths while label *5* has the lowest. Similar trend of columns can be found in the other 6-member groups with the same prefix (6 is the number of unique edge labels, $6 = |L|$), {1:1, 1:2, ..., 1:6}[1] and {2:1, 2:2, ..., 2:6} and so on. Hence, we can make an assumption that the label path that is composed of label paths with high cardinalities should also have high cardinality.

To define this hypothesis we introduce another notation: the **base label set** B, $B \subseteq \mathcal{L}, L \subseteq B$, such that every label path in $\mathcal{L}$ can be decomposed into pieces which are all in B, correspondingly, a **splitting rule** defines how to divide a label path. For example, $\mathcal{L}^6$ on data set "Moreno health" 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 that starts from start vertex, at each split step always cut piece as long as possible, this rule also produce the least number of pieces, a label path "4:4:3:3:6" can be divided into "4:4", "3:3" and "6".

The idea is that we store a base label set with appropriate size which provides sufficient base labels to approximate the sequence and not consume too much memory. With the splitting rule, a label path is decomposed into base characters whose cardinalities are recorded. Further, the cardinalities can be used to construct an approximately monotonic sequence of $\mathcal{L}^k$.

5.2.2 Concepts

An ordering method can be described by three components:

- a finite **character set**, the base label set in the previous section. In this thesis, we focus on the approach that takes the edge label set as the base label set, i.e. $B = L$. The memory needed for storing the cardinalities is $|L| * 4 * 32$-bit as we record not only the characters but also their cardinalities.

[1] In this section we use "a:b" to represent the label path identifier in order to avoid the confusion of using "a,b"

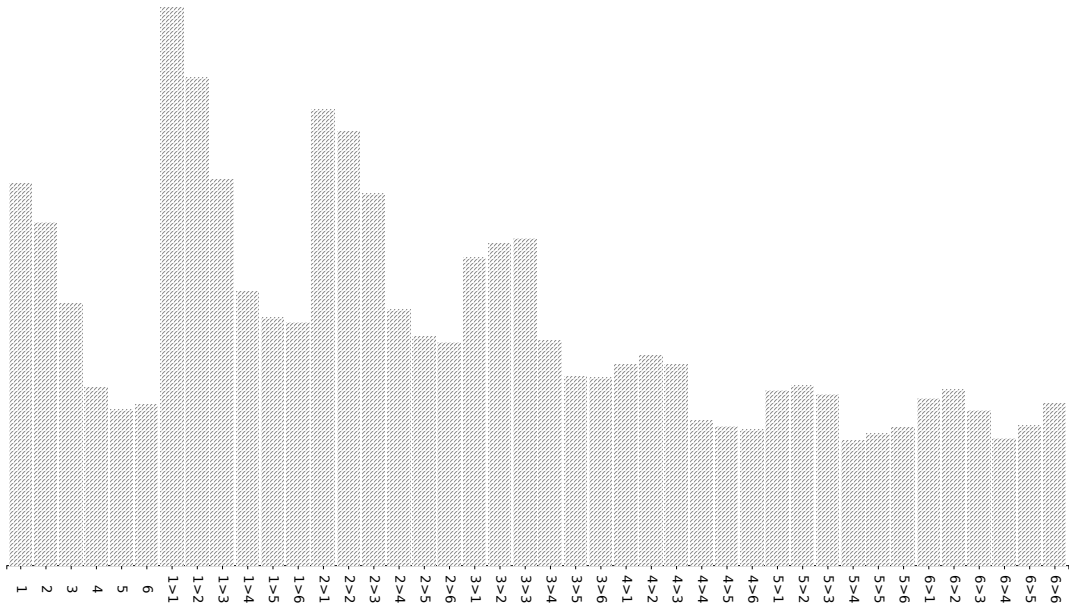


Figure 5.1: Visualization of data distribution of data set "moreno health" with k=2, in original order, ">" is the separator

- a **ranking rule** over the character set that gives a rank for each character. It is a bijection maps between edge label set L and integer set $[1, |L|]$.
- a **ordering rule** that associates with ranking rule to eventually determine the index of a label path (sequence of character) in $\mathcal{L}^k$. It is a bijection maps between label path set $\mathcal{L}$ and integer set $[0, |\mathcal{L}^k|)$. A constraint on it restricts its behavior, such that a ordering rule is efficient and valid only if it returns the exact positional number without generating the whole sequence of label paths, otherwise extra memory is needed to store $|\mathcal{L}|$ index values. That is exactly the same reason why ideal ordering is not feasible within the constraints of this research.

Note that all algorithms introduced in the following sections are only ordering rules, their outcome is not yet a valid label path, that is to say, a permutation of an edge label. It is actually a permutation of ranks, each of which represents an edge label. It needs to associate with the ranking rule to finally transform the result into a label path. A complete ordering method, therefore, is seen as the combination of a ranking rule and an ordering rule on a given dataset. We usually obscure the dataset and refer to an ordering method that is composed of A ranking rule and B ordering rule as B-A ordering.

Given an ordering method O, $O_{ranking}(\ell, \mathcal{L})$ denotes a *ranking function* which takes a label path ℓ and returns the position number i, i.e. the bijection maps from $\mathcal{L}$ to $[0, |\mathcal{L}|)$. Likewise, $O_{unranking}(i, \mathcal{L})$ denotes an *unranking function* which maps in the reversed direction.

Additionally, we define the property **backward compatible** of an ordering method. Given an ordering method O and two integer $k_1, k_2, k_1 < k_2$, we say O is backward compatible if and only if $O_{ranking}(\ell, \mathcal{L}^{k_1}) = O_{ranking}(\ell, \mathcal{L}^{k_2})\ \forall\ \ell \in \mathcal{L}^{k_1}$. It is important and necessary to specify k value when running an ordering method which is not backward compatible, since the positions of a label path in sequences ordered by two instances of the ordering method can be different.

There are two ranking rules in our project:

- **alphabetical ranking:** Ranking based on alphabetical order of edge labels. This is the

naive ranking rule which associates with no graph specific information.

- **cardinality ranking:** Ranking based on the cardinality of edge labels, which locates edge label with low cardinality in front of that with larger. More precisely, it can be expressed by the following formula:

$$l_1 <^{card} l_2 \iff f(l_1) < f(l_2)$$

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.

In this chapter, under the circumstance where no confusion will be caused, these two ranking rules are interchangeable in the interest of simplicity. The symbol $rank(l)$ represents $(alph(l), card(l))$, and the comparison between two edge labels l_1 and l_2 has the same result as that between their rankings:

$$l_1 > l_2 \iff rank(l_1) > rank(l_2)$$

5.2.3 Numerical Ordering (Original)

This ordering rule is exactly the same as an $|L|$-based positional numerical system. An integer itself has numerical value and also represent different values depending on the position it occupies in the permutation.

For example, to compare two sequence $\ell_1 =< l_1^1, l_2^1, ..., l_m^1 >$, $\ell_2 =< l_1^2, l_2^2, ..., l_n^2 >$, if they have different lengths, the one with shorter length has smaller ranking. Otherwise we check the pairs $(l_1^1, l_1^2), (l_2^1, l_2^2), ...$ one by one until we find a pair (l_i^1, l_i^2) that has different values, $\ell_1 < \ell_2$ if $l_i^1 < l_i^2$.

Formally, we have

$$\ell_1 < \ell_2 \iff \begin{cases} |\ell_1| < |\ell_2| & |\ell_1| \neq |\ell_2| \quad (5.1) \\ \wedge_{j=1}^{i-1}(l_j^1 = l_j^2) \wedge (l_i^1 < l_i^2) & |\ell_1| = |\ell_2| \quad (5.2) \end{cases}$$

Base on this definition, we present two algorithms (Algorithm 4 and Algorithm 5) for ranking and unranking permutations of integers in this order. Time complexity of Algorithm 4 and Algorithm 5 are both $\mathcal{O}(k)$.

Algorithm 4 Ranking in numerical order

```
1: procedure RANKING_IN_NUMERICAL_ORDER(path, |L|)      ▷ path expression, edge label size
2:     r ← 0
3:     for i ∈ {0, ..., |path| − 1} do
4:         j ← rank_of_character(path[i])
5:         r = r + (j + 1) ∗ pow(|L|, |path| − i − 1)
6:     end for
7:     return r − 1                  ▷ returns the index of path in a numerically ordered sequence
8: end procedure
```

The *ranking* is relatively easy to understand, it is the sum of number each edge label l_i represents based on its position and its ranking in $|L|$ in label path $\ell = l_1 l_2 ... l_{|\ell|}$.

$$rank(\ell) = \sum_{i=1}^{|\ell|} rank(l_i) * |L|^{|\ell|-i} \quad (5.3)$$

Algorithm 5 Unranking in numerical order

```
1: procedure UNRANKING_IN_NUMERICAL_ORDER(index, L)          ▷ index, edge label set
2:     if |L| = 0 then
3:         return null
4:     else
5:         path ← []
6:         if |L| = 1 then
7:             for i ∈ {0, ..., index} do
8:                 path.add(L[0], 0)
9:             end for
10:        else
11:            index ← index + 1
12:            while index > 0 do
13:                n ← (index − 1) mod |L|
14:                path.add(n, 0)
15:                index ← ⌊(index − n)/|L|⌋
16:            end while
17:        end if
18:        return path
19:    end if
20: end procedure
```

Depending on the feature of a numerical system, Numerical Ordering is backward compatible. In each subset of length-i, the position of any permutation is fixed, regardless of the maximum length of permutation.

5.2.4 Lexicographical Ordering

This ordering is the same as the ordering rule used in dictionaries, it is similar to numerical ordering with the following difference:

Instead of comparing length of two sequences first, we append $k-|\ell|$ **blank** (a special symbol that for all $l \in L, rank(blank) > rank(l)$) to every ℓ to form a length-k sequence, then apply Formula 5.2.

A recursive unranking algorithm (Algorithm 6) can be constructed by the Formula 5.4. Its time complexity is also $\mathcal{O}(k)$.

$$|\mathcal{L}^k| = \sum_{i=1}^{k} |L|^i = |L| * \sum_{i=0}^{k-1} |L|^i = |L| * (1 + \sum_{i=1}^{k-1} |L|^i) = |L| * (1 + |\mathcal{L}^{k-1}|) \tag{5.4}$$

As it locates permutation with the same prefix together in the same subset, when k increases or decreases, the size of a subset also rises or drops, the index of sequences in later subsets will also change. Hence, Lexicographical Ordering is **not** backward compatible.

5.2.5 Sum-based Ordering

The general idea of this ordering method is to decompose a given label path into edge labels, and then use the sum value of ranks of all edge label entities in it to represent the cardinality of the label path. It is an ordering method that is based on the sum of edge label ranks.

Algorithm 6 Unranking in Lexicographical order

```
1: procedure UNRANKING_IN_LEXICOGRAPHICAL_ORDER(index, L, k) ▷ index, edge label set, k
2:     if k = 0 then
3:         path ← {}
4:         path.add(L[index], 0)
5:         return path
6:     else
7:         k ← k − 1
8:         perchar ← Σ_{i=0}^{k} |L|^i
9:         remainer ← index mod perchar
10:        index ← ⌊index/perchar⌋
11:        if remainer = 0 then
12:            path ← {}
13:            path.add(L[index], add)
14:            return path
15:        end if
16:        subpath ← unranking_in_lexicographical_order(index, L, k)
17:        subpath.add(L[index], 0)
18:        return subpath
19:    end if
20: end procedure
```

Mathematically, we are looking at a k-length permutation of integers that range in $\{1, .., |L|\}$. The general idea is to divide the whole ordered sequence of permutations into partitions. This executes for several times with different depth, different scale, and different formula. The partitions generated after one dividing action are of different characteristics and marked by different properties than those of others. At each layer, the algorithm locates the integer into target partition after calculating: 1) size of partitions, 2) index of partition. Algorithms and formulas for calculating these two properties vary for each layer.

The first and the most shallow dividing is to group permutation by its length, namely, $1, 2, ..., k$, the one with shorter length locates in front of that with a longer length, the total number of n-length permutations is given by following formula:

$$sum_n = |L|^n$$

Secondly, all length-m label paths can be grouped by the **summed ranks** property. Those with smaller summed rank have the smaller ranking. This handles the assumption that the label path itself will have high cardinality if its decomposed elements, i.e. edge labels, have high cardinalities.

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

A problem needs to solve is to determine how many label paths are in the group with certain m and sr_m. This question is the same as how many ways to distribute sr_m indistinguishable balls over m distinguishable bins of finite capacity $|L|$ with at least one ball in each bin. Combined with knowledge of *Combinatorics* and *Inclusion−exclusion principle*. Formula 5.5 gives the solution.

$$dist(sr_m, m, L) = \sum_{j \geq 0} (-1)^j \binom{m}{j} \binom{sr_m - j * |L| - 1}{m - 1} \tag{5.5}$$

Next, we explore combinations inside a group marked by certain length m and summed rank sr_m, which is all **integer partitions** of sr_m into exactly m parts, where each part is less than $|L|$. Let integer v, b represent sr_m and $|L|$ respectively, a general formula of integer partition $ip(v, b, m)$ is as follows:

$$ip(v, m, b) = \bigcup_{i=0}^{\lfloor v/b \rfloor} ip(v - i * b, m - 1, b - 1), \underbrace{b, \cdots, b}_{\text{i } b\text{s}} \tag{5.6}$$

Based on the Formula 5.6, we present the Algorithm 7. Line 2 gives the relation between v and b, m, which indicates v is bounded by $b * m$. By looking at line 11 and line 16, we can identify that the fan-out of this recursive algorithm is $log(m)$, i.e. $log(|L|)$. Additionally, from the stop condition in line 2 and recursive call in line 17, the depth is k. Hence, its time complexity is $\mathcal{O}(log(|L|)^k)$.

Algorithm 7 Integer partition

```
1:  procedure IP(v, m, b)                          ▷ the integer, bin number, bin capacity
2:      if v < 1 ∨ v > b * m ∨ m ≤ 0 ∨ b = 0 then
3:          return null
4:      end if
5:      if m = 1 then
6:          if v > b then
7:              return null
8:          end if
9:          return [[v]]
10:     else
11:         nob ← ⌊v/b⌋                            ▷ Maximum number of bs in v
12:         if nob = m then
13:             return [[b, ··· , b]]
                                  m bs
14:         end if
15:         r ← []
16:         for i ∈ {nob, ..., 0} do
17:             for prtt ∈ IP(v − i * b, m − 1, b − i) do      ▷ recursive call
18:                 r.add(prtt)
19:             end for
20:         end for
21:         return r
22:     end if
23: end procedure
```

The algorithm we construct outputs all combinations in our most desired order: combinations with more *large* edge labels has the higher ranking. For example, a solution to the instance v=4, b=4, m=2 is [{2,2},{1,3}], the reason why {1,3} comes after {2,2} is we believe edge label with higher cardinality, that is, 3, has more impact on producing high cardinality of the whole label path, such that {1,3} outweighs {2,2}. After handling the order of combination, there are two steps before we eventually obtain the target permutation.

The first one is to determine how many permutations we skip when we skip a partition. This can be transformed into a simple mathematical question: How many permutations can be generated by a certain combination, in which there might be duplicates. The lengths of all permutations are the same as the length of the combination. Let C denote the combination, d_i denote the number of times an integer i occurs in C, the number of permutations is given by following formula:

$$nop(C) = \frac{|C|!}{\prod_{i \in \{0,\dots,|L|-1\}} d_i!} \tag{5.7}$$

Finally, we reach the combination the target permutation belongs to. Algorithm 8 finds it in the sequence of permutations generated by the combination. Its time complexity is $\mathcal{O}(|C|^2)$, i.e. $\mathcal{O}(k^2)$

Algorithm 8 Unranking permutation of combination with duplicate

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

In the end, Algorithm 9 illustrates the complete version of unranking permutation in sum-based order. It wraps Formula 5.5, 5.7 and Algorithm 7, 8. Note that the variable *index* is 0-based. However, Algorithm 7 outputs integer partition in 1-based. We use an extra variable p' to do the transform.

The algorithm has a triple *for* loop structure. However, with the *continue* statement in each loop and the *return* statement in the innermost loop, it is actually a loop-free function. There are two expensive operations in this algorithm, namely: integer partition in line 15 and unranking permutation in line 23. Both of them execute only once through the algorithm. Hence the overall time complexity is also $\mathcal{O}(log(|L|)^k)$.

Because of the fact that the rank of each edge label does not change as k changes, sum-based Ordering is also backward compatible.

Algorithm 9 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|)                    ▷ P is an array's array
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
```

5.3 Examples

Here we illustrate these ordering methods with examples on an artificial dataset which has 3 unique edge labels and its label paths set with k up to 2.

Edge Label	1	2	3
Degree	20	100	80
Rank	1	3	2

Table 5.1: Edge degree (length-1 distribution)

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 5.2: Summed ranks

Based on the edge degree in Table 5.1 and summed ranks in Table 5.2, label paths arranged in these orderings are displayed in Table 5.3. 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 will be referred to as *num-alph*, *num-card*, *lex-alph*, *lex-card* and *sum-based*.

Index / O	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 5.3: Ordered label path set in different ordering method O

5.4 Summary

Since in this chapter we have introduced several technical terms, here we give a summary.

- **ranks:** A integer for each edge label.
- **ranking rule:** The rule to assign the rank to edge label.
- **ordering rule:** The rule to locate label path within the label path set.
- **ranking function:** Algorithm that is used in the scope of ordering rule, takes a label path, returns the corresponding index.
- **unranking function:** Algorithm which is also used in the scope of ordering rule, takes a integer, returns the corresponding label path.
- **ordering method:** A instance of framework consists of character set, ranking rule and ordering rule, finally determines the bijection between label path set $\mathcal{L}$ and integer set $\{0, ..., |\mathcal{L}| - 1\}$

We introduce three ordering rules and two ranking rules in this chapter. Note that sum-based ordering only makes sense when associated with cardinality ranking. We run experiments on these 5 combinations in Section 7 to see their performances.

There is an unranking algorithm for each of the three ordering approaches, but only one ranking algorithm that gives a label path its index in Numerical Ordering. We always use its index in *num-alph* ordering *num-alph*(ℓ) to represent a label path ℓ. Given a label path identifier, knowing its index in other ordering method is useless for us in the project. We only care about the reversed way. The details about associating a histogram with an ordering method without consuming extra memory are described in Section 6.4.

Chapter 6

Implementation

As we already have introduced the construction algorithms for histograms, algorithms for ordering approaches as well as the logic behind them, in this section we will discuss the implementation in more details, and try to cover aspects and details that are worth mentioning.

6.1 Basic graph structure

We planned to use an open source Java library as our tool in inspecting graph, however, the library we chose does not provide **labeled** feature which is a necessity for us in this project, we chose to build a basic graph structure[1] ourselves from the ground.

This Graph class maintains a mapping between integers as vertex names and the Vertex objects, each Vertex has two maps, one records the mapping between the labels and lists of outgoing Edges, and one records that of incoming Edges, each Edge has a source Vertex object and a target Vertex object and an integer as a label.

6.2 Baseline of path selectivity

The baseline calculation is to get the exact number of results that fulfill the path query for each possible label path with length up to k. We use data structure we built in section 6.1 and do a Breadth First Search.

6.2.1 Java in-memory implementation - A Breadth First Search

Given a graph g, we have a path L of length n, which can be represented as $\langle l_1, l_2, ..., l_n \rangle$, to get the baseline, for each vertex v_1 in g, we first set up a counter c with initial value 1, we start from v_1 and search in the outgoing edge set of v_1 for any edge with label l_1, return the set of all the target vertices of these edges, now we set c to 2, and for each vertex in the set, which we will refer to as v_2, we search in the outgoing edge set of v_2 for any edge with label l_2, we keep doing this until c reaches n and we go through the edge e_n, we return the total number of v_{n+1}, which will be seen as the selectivity of path L on v_1. This process is performed on all vertices in the graph and the sum of selectivities is the selectivity of path L on g.

Based on the Graph structure we've built, the recursive method in Java is as follows:

[1] https://bitbucket.org/realyasswl/path_selectivity

```
int pathSelectivity(List<Integer> path, Vertex v, List<Integer> stored) {
    Integer result = 0;
    //  get the current label
    Integer label0 = path.get(0);
    List<Vertex> list = getVerticesByLabel(v, label0);
    if (path.size() == 1) {
        if (stored == null) {
            result = list.size();
        } else {
            for (Vertex t : list) {
                if (stored.indexOf(t.getId()) < 0) {
                    stored.add(t.getId());
                    result++;
                }
            }
        }
    } else {
        List<Integer> subPath = path.subList(1, path.size());
        for (Vertex v1 : list) {
            result += pathSelectivity(subPath, v1, stored);
        }
    }
    return result;
}
```

In the code above, the parameter *path* is the sequence of edge labels and the Vertex v is the starting point, this method is used in a *for* loop which travels the whole vertices set that the Graph maintains, it can also be used to find selectivity of a label path that starts at a specified vertex.

Note that we have an additional parameter *stored* representing a list of vertices in the method signature. Given the source vertex, it stores the vertices that have been visited as the target vertex of the label path. We introduce it because we notice in the testing process, given a specific label path, there exist graphs that have different paths that start and end at the same vertices pair, follow the given path identifier, and go through different vertices along the road. By setting the parameter to *null*, the method counts all target vertices, regardless of whether they are the same, which is the expected behavior according to the definition in Section 2.2. Additionally, we keep interface of the feature that excludes duplicate for further research and development.

As the example shows in Figure 6.1, the selectivity that ignores duplicate and records duplicate for label path $(1, 2)$ is 1, 2, respectively.

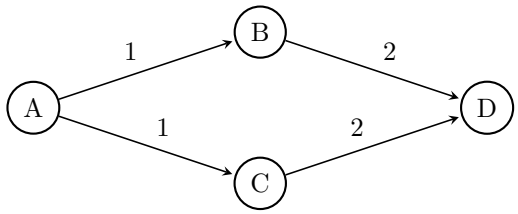


Figure 6.1: Multiple fit example

6.3 K-path index histogram

Once we have the baseline for all paths with length up to k, we can build our k-path index histogram. Given a parameter b, which indicates the number of buckets we can use, we divide the paths in ascending order into b groups and build a bucket for each group according to the construction algorithms described in Section 4.2, 4.3 and 4.4.

6.4 Decoupling

In construction phase and estimation phase, serial histogram (all histogram schemas in our project are serial) always sees $\mathcal{L}$ as an ordered sequence. In the interest of simplicity, we don't want histograms to maintain the order, which will lead to difficulty in both implementation and maintenance, with considerable code redundancy. The ordering and histogram schema are unrelated and independent with each other in principle, they should be kept that way in practice.

Guided by the *Object Orientation Programming* paradigm and the *dependency injection* technique, decoupling histogram and ordering is implemented with a **Mapper** Object, which is interpreted in *Java* as an *Interface*. It has only one method signature:

```
public int map(int index, int k);
```

The Class that implements Mapper Interface provides a bijection function, maps between two identical integer sets $\{0, ..., \mathcal{L}^k - 1\}$. The *index* parameter is the index of a label path in a sequence that is arranged in the specific ordering method, starts from 0. The k is the maximum length of label path, as in k-path index histogram. To make everything work, it also needs to associate with an integer array that represents the cardinalities of length-1 label path, i.e. the edge label distribution. This is included as a parameter of the constructor of an Implementation Class since it will be referred to for many times.

With access to all necessary data, the method runs the unranking function of corresponding ordering method to get the target label path identifier, which is then passed to the ranking function of *Numerical-Alphabet* ordering, finally, returns the index of the target label path in original order.

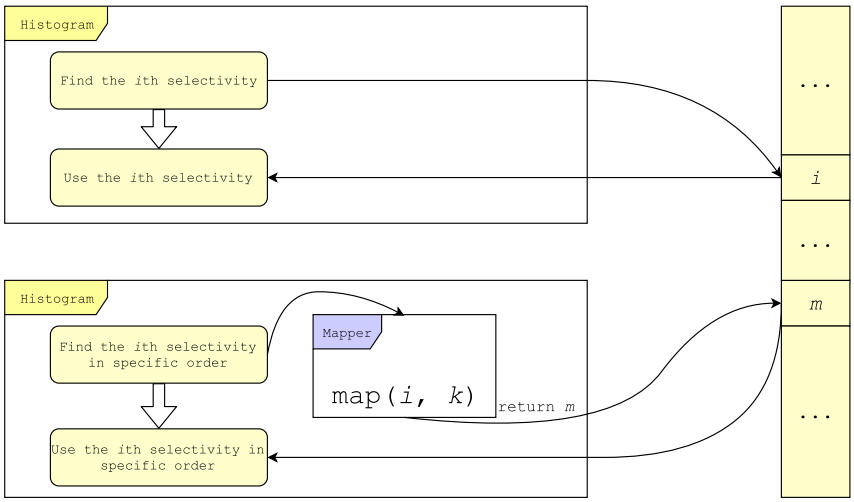


Figure 6.2: Flow chart of histogram with and without Mapper Object

Associated with a Mapper Object of specific ordering, a histogram can focus on the construction and estimation without knowing the exact order that it is handling. It sees through the Mapper as if it didn't exist. Furthermore, a histogram can switch between ordering methods seamlessly without modifying its code. The Figure 6.2 illustrates how histogram read cardinality array with and without Mapper.

Chapter 7

Experiments

7.1 Objectives

The objectives of our experiments include:

- Given a certain β, compare the performances of different histogram schemas in terms of estimation accuracy.
- Verify the impact of sum-based ordering on boosting estimation accuracy.
- Observe and explain the behavior of histograms and ordering methods on datasets with different characteristics, extract and summarize the findings to a reference of best practice.

7.2 Dataset

We run the experiments on the following datasets. Table 7.1 shows their basic information.

- Moreno health [8] from KONECT
- Subgraph of YAGO dataset [14]
- Subgraph of DBpedia dataset [4]
- Generated by SNAP.py [18], with *Erdos – Renyi* model [5], which construct a random graph by connection node randomly. Each edge is included in the graph with probability p independent from every other edge. This dataset will be referred to as *SNAP-ER*. Graphs generated by this model will be referred to as ER graphs.
- Generated by SNAP.py [18], with *Forest Fire* model [12], which resembles of how the forest fire spreads by igniting trees close by. This dataset will be referred to as *SNAP-FF*. Graphs generated by this model will be referred to as FF graphs.

Dataset	#Edge Label	#Vertices	#Edges	Real world data
Moreno health	6	2539	12969	yes
YAGO subgraph	6	52813	74631	yes
DBpedia subgraph	8	37374	209068	yes
SNAP-ER	6	12333	147996	no
SNAP-FF	8	50000	132673	no

Table 7.1: Dataset information

7.2.1 Moreno Health

We get it from KONECT as the dataset we start with. Here I quote from its page for description: "This directed network was created from a survey that took place in 1994/1995. Each student was asked to list his 5 best female and his 5 male friends. A node represents a student and an edge between two students shows that the left student chose the right student as a friend. Higher edge weights indicate more interactions and an edge weight shows that there is no common activity at all." [8]

This dataset is a directed labeled graph which consists of 12971 vertices and 2539 edges, the number of distinct labels is 6. It is a dataset with the proper size for our project, big enough to give meaningful result and considerable complexity, not too big to process with our limited time and hardware resource.

It is important to us that it is a real world dataset because it can help with show the how much impact does our approach have in practice and how robust it is to unpredictable exceptions in real world. Furthermore, one of its characteristics which is even more precious is that it is a **homogeneous** graph, i.e. all nodes in it are homogeneous, have the same properties and can be classified into the same category, theoretically and consequently, any two nodes have the potential to be connected.

7.2.2 YAGO subgraph and DBpedia subgraph

Using graph theory in analyzing Wikipedia has been an active field of research, both YAGO and DBpedia are Linked Open Data that are extracted from Wikipedia, the latter is larger in terms of volume.

The original full datasets have millions of edges, which is too large to process in our research. Our implementation can Additionally, depending on the shallow investigation we take on these datasets, we know the two datasets are not homogeneous. the nodes can be categorized into thousands of types or even more. The problem with the heterogeneous graph is that it produces 0 cardinality for many label paths even when k is small, the portion of 0-cardinality label path grows rapidly as k increases. For example, *wasBornIn* and *hasChild* are two edge labels in YAGO. Judging by the meaning of the text, the former connects a *Human* and a *Location* or *City*, while the latter connects two *Human*. By common logic we can assert that the cardinality of label path $\langle wasBornIn, hasChild \rangle$ is 0, as well as any label path that contains such a piece.

Therefore, extracting approximately homogeneous subgraphs from the original datasets provides datasets with not only proper size, also better quality in terms of homogenization. With the help of these two pre-processed datasets which already provide the selectivities of all length-1 and length-2 label paths, we develop a **filtering algorithm** to produce a subset of label set, and extract all edge records whose edge label are in this subset to form a subgraph, such that none label path has 0 cardinality among $\mathcal{L}^2$. Additionally, this algorithm specifies the edge degree to be in a range, such that neither universal labels nor unique labels will be included in the subset. Universal labels means the labels that connect almost anything, for example, *linksTo*, and unique labels are those that appear under strict conditions. Reflected in the edge degree, the degree of universal labels can reach millions or more while that of unique labels is only no more than one thousand.

7.2.3 SNAP-ER and SNAP-FF

Snap.py provides the following parameters through its interface: number of nodes, average degree per node, number of edge labels, edge label distribution type, graph model type. As mentioned before, the graph model defines how the edges are created, from the description of both models,

we can see SNAP-ER as a correlation free dataset, and SNAP-FF a dataset with correlation to some extent. Additionally, since distribution type has a considerable impact on graph quality, we use *normal variate* distribution to generate the result, other than this, there are two other distributions supported by snap.py, namely: *uniform* and *exponential variate*. We choose normal variate to generate a biased dataset while keeping it from being extreme.

7.3 System Specification

All experiments are conducted on a local desktop computer. It consists of an Intel(R) Core(TM) i5 CPU 750 @ 2.67GHz 4-core processor, 4 GB DDR3-1333 memory. The operating system is 16.04 Long Term Support Ubuntu with kernel version 4.4.0-83. The Java Runtime Environment information is OpenJDK 1.8.0_131, 64-Bit Server VM.

7.4 Evaluation

7.4.1 Estimation Running Time

Once histogram is constructed, the estimation of a given label path can be obtained in two steps: 1) locate the bucket that contains the label path, 2) calculate the estimation within the bucket. As specified in Section 4.1.1, operation in step 2 is cheap in terms of calculation. Furthermore, as explained in Section 6.4, and 4.1.1, since all histogram schemas in this research are *serial*, locating bucket for a given label path can be solved efficiently by performing a binary search through the buckets.

Theoretically, the running time in histogram dimension is cheap, we didn't run experiment specially for each schema. Instead, we run extensive experiments to verify the running time of estimation associated with different ordering methods. More specifically, we set $k = 6$, five V-Optimal histograms are built, each of which associated with corresponding ordering method. The total number of label path is 55996. There are seven test cases $T_1, ..., T_7$, in T_i we assign $\frac{55996}{2^i}$ to β. All seven test cases are executed for 100 times. We calculate the average value of running time and show it in Table 7.2. The result shows that generally the four naive ordering method, namely, num-alph, num-card, lex-alph and lex-card have similar running time, sum-based is approximately 20% slower.

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 7.2: Average estimation running time in V-Optimal histogram with different ordering methods (in millisecond)

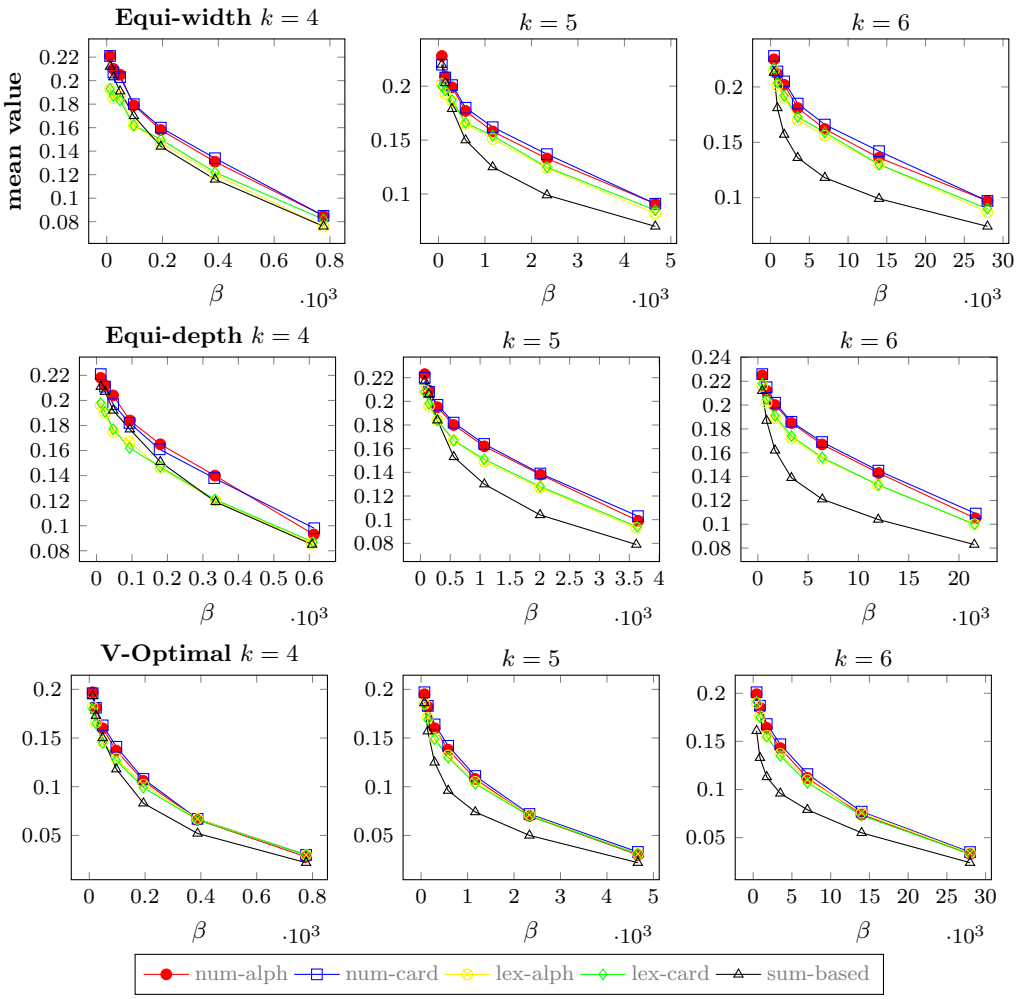


Figure 7.1: Moreno Health mean value of error rate of estimation, group by histogram type

7.4.2 Mean Error rate of Moreno Health

Figure 7.1 shows the mean value of error rate of all three histogram schemas with 5 different ordering approaches with k equals to 4, 5 and 6. The x axis is the number of buckets actually used, the y axis is the error rate. The result is grouped by histogram schema. Several interesting things are observed:

1) The results of lexicographical ordering associated with alphabet ranking and cardinality ranking are almost the same throughout the whole process. The results of the two numerical orderings are also very close to each other in all nine sub-figures.

2) By looking at the columns, we notice that in the first column, the quality of sum-based ordering is slightly better than other ordering methods in the V-Optimal histogram, in other two histograms, the qualities of different orderings are very close to each other. As k increases to 5, sum-based starts to outperform others in Equi-width and Equi-depth histogram, while keeps its leading position in V-Optimal. When k reaches 6, the distances between sum-based and other ordering approaches in all three histograms become larger.

3) By looking at the rows, we notice when k equals to 5 and 6, in V-Optimal histogram, sum-based

loses its advantage to other orderings in other two histograms to some extent. Furthermore, a less obvious fact which draws our attention is that Equi-depth histogram with sum-based ordering produces similar estimation in terms of error rate as V-Optimal histogram does with original ordering. This means that the performance of a basic histogram with original ordering can be boosted in two directions: switch the ordering method and use advanced histogram. While giving similar result, the former is cheaper in terms of running time.

4) Lexicographical ordering generally outperforms numerical ordering, even though slightly.

Figure 7.2 shows the same result from a perspective in which results are grouped by ordering methods. In the charts, we can observe that optimized-quality histogram gives the best result among all orderings. Equi-depth has the same quality as Equi-width with sum-based ordering and slightly outperforms Equi-width in the other four. The shorter curve of Equi-depth reflects its *saving-bucket* feature, which is discussed in Section 4.5.

7.4.3 Mean Error rate of SNAP-FF and SNAP-ER

As illustrated in Figure 7.3, 7.4, 7.5 and 7.6, it is not surprising that sum-based ordering outperforms other ordering methods in every dimension in both SNAP-FF and SNAP-ER dataset.

In the SNAP-ER dataset, the process of a traveling through the graph along a label path can be seen as a series of independent events. At any node, the probability of taking an edge label l in next step is decided by the degree of this edge label. This process perfectly matches our assumption. As for the result of SNAP-FF, the probability is not only decided by the degree of edge label, but also by the path it already took. Although it does not fit our assumption as closely as SNAP-ER does from the aspect of edge label distribution, it is compensated by the correlation between edges.

Other than the obvious fact that V-Optimal histogram gives the best estimation, the behaviors of numerical ordering and lexicographical ordering also draw our attention. Figure 7.3 shows the two numerical ordering methods generally outperform the two lexicographical ordering methods in all three histograms. In Figure 7.5, lexicographical ordering methods outperform numerical ordering methods in Equi-width histogram and Equi-depth histogram. The two models determine the characteristics of the two graphs. As k increases, cardinality of a label path with many high-degree edge labels in an ER graph grows much faster than that in an FF graph. Such that for a label path, the degrees of edge labels in it have more impact on its cardinality than its length in an ER graph. This partily explains the performance difference between two types of ordering methods.

Also note that in Figure 7.6, the curve of Equi-depth is much shorter than that of other two, it is also shorter than the Equi-depth curve in Figure 7.4. This is caused by two reasons. The first one is the *saving-bucket* feature of Equi-depth histogram. The second is that by choosing edge label totally randomly with normal variate distribution in SNAP-ER, and also keep in mind that cycles are allowed, the cardinality of the label paths with long length and many high-degree labels can be huge, can easily reach hundreds of thousands times of that of shorter label path. Consequently, the *depth* is also large, a bucket stores more label paths than it does in other datasets. This is also reflected by the exceptional quality of sum-based ordering in V-Optimal histogram in SNAP-FF dataset.

Another notable fact in these four figures is the abnormal performance of num-card ordering in Equi-width histogram, a possible explanation for this is that as we choose the normal-variate distribution for these two datasets, and the numbers of edge labels in them are both even numbers, for $i > 0$, the cardinalities of $(2 * i)$th and $(2 * i - 1)$th label path in num-card ordering can be close to each other.

Edge Label	0	1	2	3	4	5	6	7
Degree	31154	30980	21368	21056	30812	21338	21276	31084

Table 7.3: DBpedia Subgraph edge degree

7.4.4 Mean Error rate of DBpedia Subgraph

The result illustrated in Figure 7.7 is less exciting. Although sum-based ordering generally gives best quality when β is small, it does not show the dominating power as it does in other datasets. When β is relatively large, num-card ordering method outperforms others.

From Figure 7.8 we observe that Equi-width gives better quality than Equi-depth with the two numerical ordering. As with other three ordering methods, the two generally have the same quality. As always, V-Optimal outperforms them all.

Although sum-based ordering does not give the outstanding performance anymore, it is still one of the ordering methods that give estimation with the lowest error. There are several possible reasons for this: 1) In a real world dataset, anything can happen. Sum-based ordering is introduced with a simple and concentrated idea, which certainly has trouble in perfectly fitting for all datasets, especially for real world dataset with unpredictable correlations and exceptions. 2) As shown in Table 7.3, the edge degree of DBpedia Subgraph is generally flat, which makes the impact of high-degree edge labels less significant, also the actual cardinality of a label path is affected more by the small probability events, i.e. node picks a low-degree label than a high-degree one. Consequently, sum-based is less accurate and effective than it is when handling datasets more distinguishable edge degrees.

7.4.5 Mean Error rate of YAGO Subgraph

The result of estimation with the histogram on YAGO subgraph is really bad, regardless of ordering method. The reason for this is very clear: When k is big, cardinalities of most of the label paths are 0. Based on a closer investigation we have the following speculation:

1) Although this subgraph is produced by the filtering algorithm in Section 7.2.2 and matches the standard, the edge labels can be categorized into two classes, and the interaction between labels in the same class is much more than that in different classes.

2) Edge label has a hierarchical characteristic which prevents the label path from going backward, such that longer label path produces less cardinality. For example, *hasChild* and *hasAcademicAdvisor*.

7.4.6 Analysis

From the discussion in last several pages, we can confidently conclude that, among all ordering methods, sum-based generally gives estimation with lowest error rate. Num-card is another degree-aware ordering method that occasionally gives good estimation. From the perspective of a histogram, V-Optimal certainly is the most effective one in terms of reducing the error rate. Equi-depth and Equi-width slightly outperforms each other alternately.

Next, we would like to explore deeper by describing the behavior of sum-based ordering and comparing it with other ordering methods. We refer to $|L|$-size set formed by label paths that have the same prefix but different last edge as a **group**. For example, in Moreno Health dataset, {1>1, 1>2, ..., 1>6} and {2>3>1, 2>3>2, ..., 2>3>6} are two groups with prefix 1 and 2>3 respectively.

In numerical ordering, to transfer from alphabetical ranking to cardinality-base ranking, operations on two scales are performed. One takes place within the group they belonged to, to form an approximately monotonic trend within each group. The other is moving the groups themselves. As all groups have monotonic trends with the same direction, i.e. either increase or decrease, it is likely that the last few label paths in one group have greater or lower cardinalities than the first few in the next group. Such that it creates gaps between the borders of two adjacent groups. When the capacity of a bucket is small, the reordering does have a impact, paths within buckets are more ordered, which becomes less effective when the bucket is large enough to cover many groups.

Lexicographical ordering, in the other hand, breaks the groups completely by matching the prefix first. It is effective with the graphs in which degrees of edge labels have more influence in determining the cardinality of a label path than its length. A possible candidate is the graph that has ER model with normal variate or exponential variate.

Sum-based ordering breaks the $|L|$-size groups to form groups with a larger size, i.e. the integer partitions, while also keeps the length information. Label paths in the same group have the same summed ranks, which approximately represent their actual cardinality. Such that as k increases, sum-based outperforms numerical ordering even significantly.

Here we present the best practice of dataset from perspective of histogram and ordering method such that the corresponding histogram and ordering method gives the best performance.

First of all, a clear impression from the discussion above suggests that the graph for sum-based ordering to give the best performance should have the following characteristics:

- homogeneous graph, any two nodes in it are potentially connectable.
- have as little correlation between edges as possible
- edge label differ greatly from each other in terms of degree

Lexicographical ordering works better than numerical ordering on a graph in which degrees of edge labels have more impact on producing cardinality of a label path than its length.

As for histogram, V-Optimal should be the No.1 choice for all scenarios. As an advanced histogram schema, it outperforms others with incomparable advantages. However, its construction time is indeed expensive according to Section 4.4. Building a V-Optimal histogram for 40,000 label paths takes about 20 minutes. If this is an issue for some case, the user can either construct the histogram with a more powerful machine in advance or try Equi-depth histogram, which is easy to build and gives a relatively good estimation.

Last but not least, we definitely don't recommend histogram approach for estimating in heterogeneous graphs, neither for that in homogeneous graphs which have hierarchical characteristics in edge labels. As the connection only exists in isolated islands or single direction, it is certain that the cardinalities of most of the long label path are 0. Take YAGO subgraph as an example, when $k = 6$, the total number of label paths is 55986 while the number of label paths that have non-zero cardinality is 3918, which is even smaller than the number of buckets. Instead of using a histogram, we can simply store the exact value for all those that have non-zero cardinality. A possible research direction for estimating path selectivity in heterogeneous graphs is described in Section 8.2.

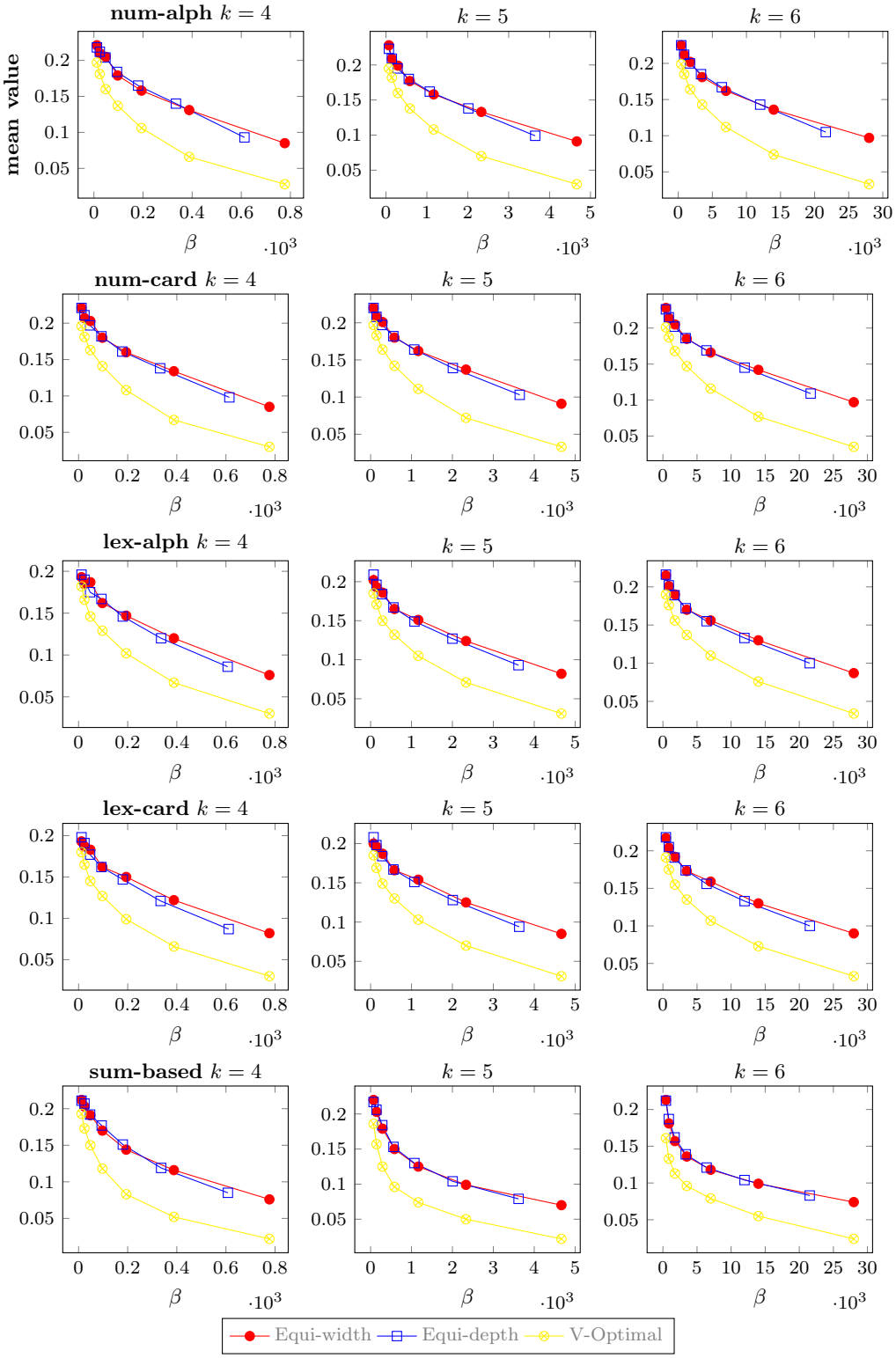


Figure 7.2: Moreno Health mean value of error rate of estimation, group by ordering method

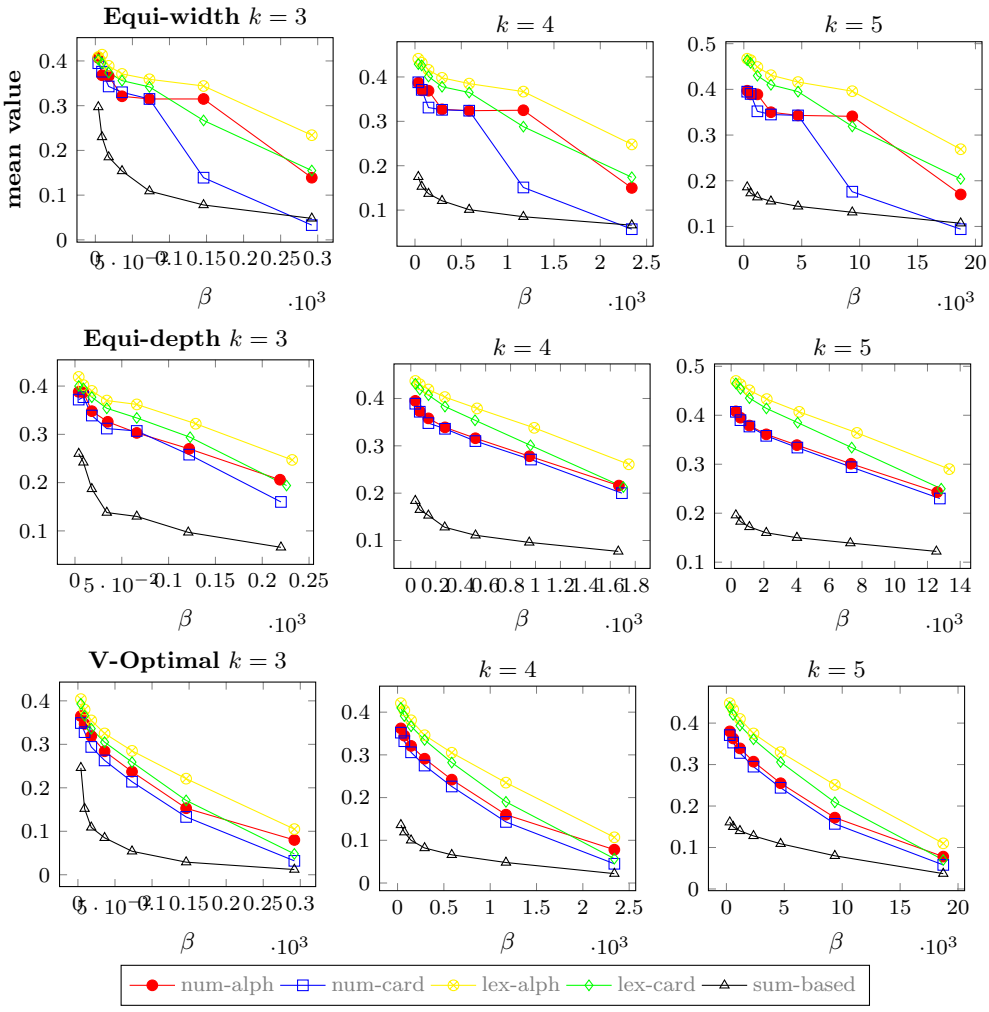


Figure 7.3: FF mean value of error rate of estimation, group by histogram type

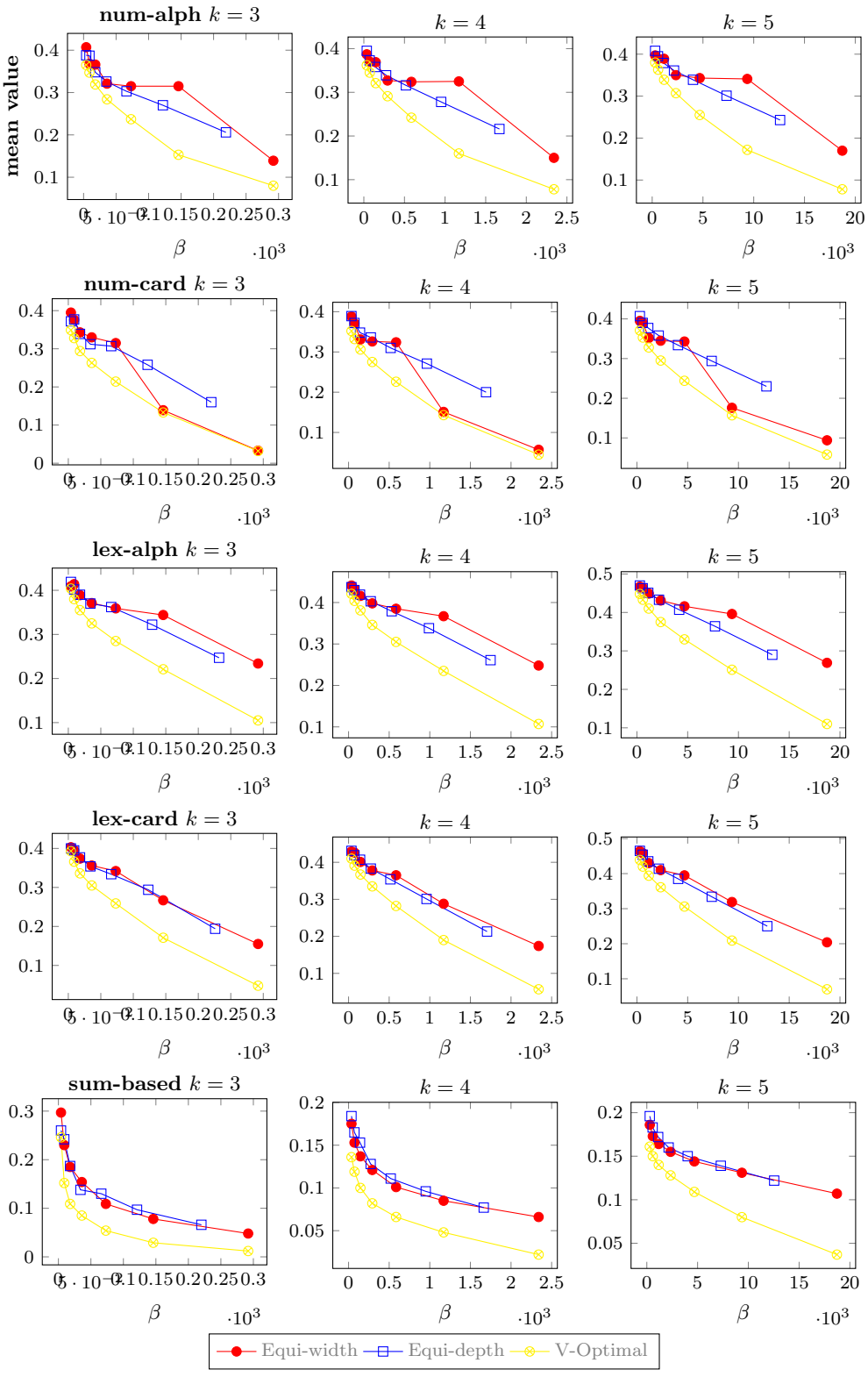


Figure 7.4: FF mean value of error rate of estimation, group by ordering method

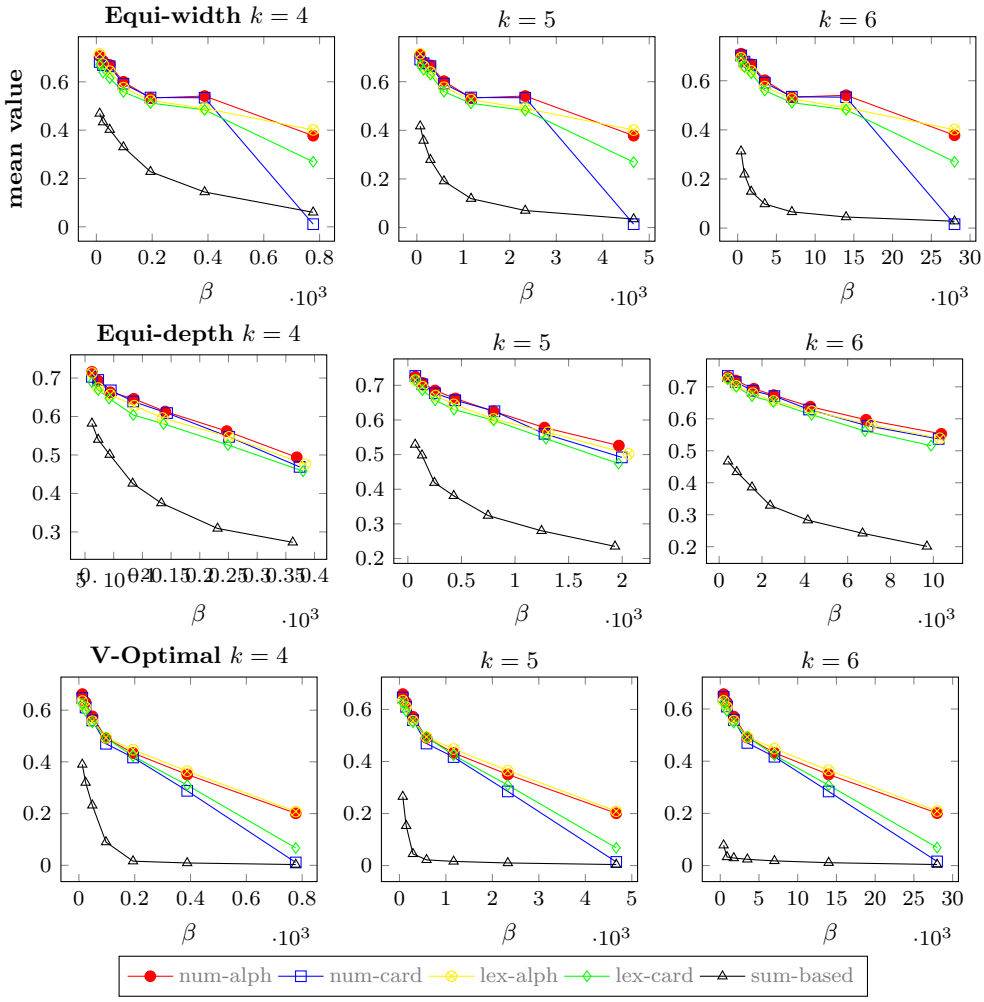


Figure 7.5: ER mean value of error rate of estimation, group by histogram type

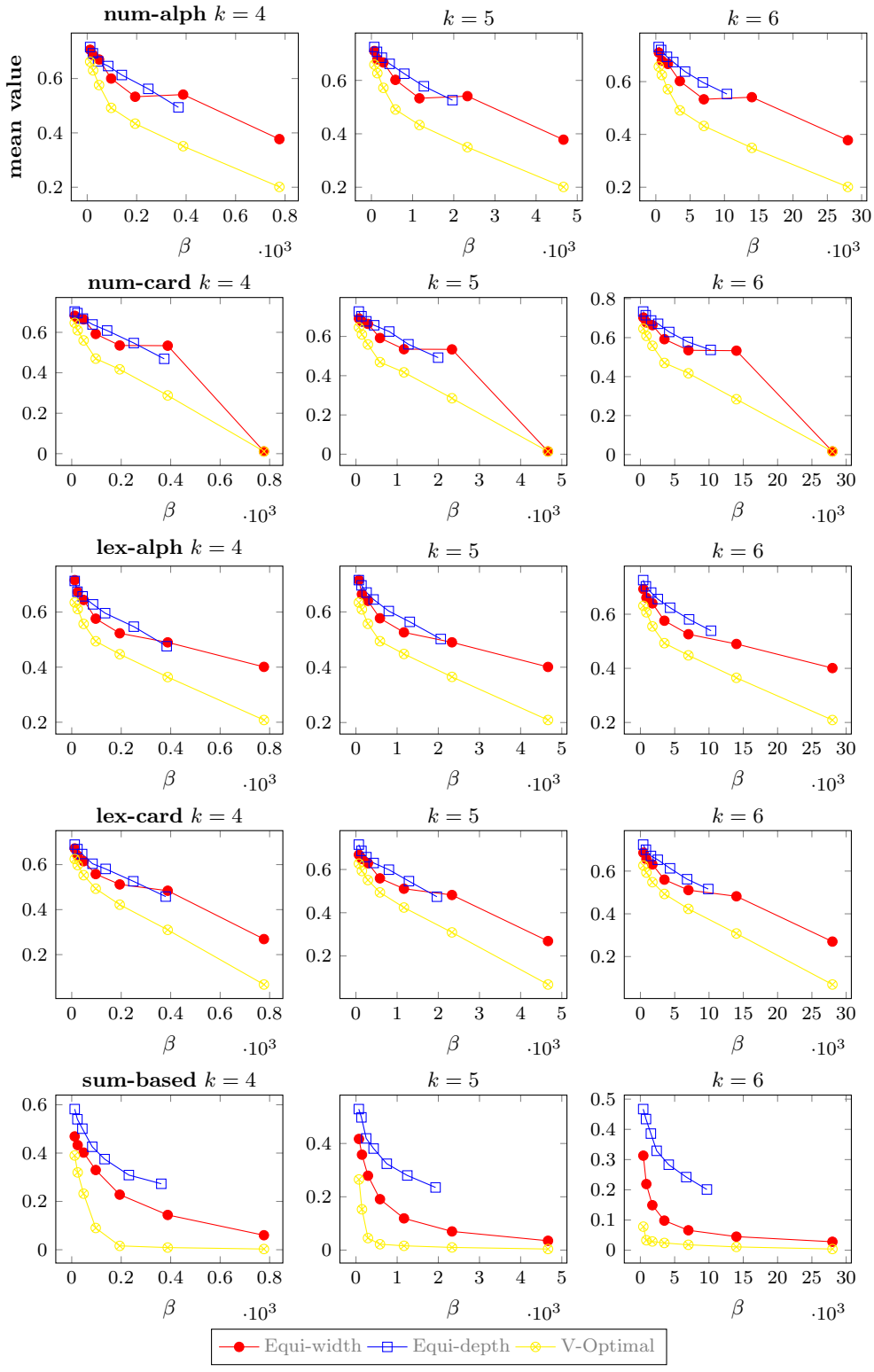


Figure 7.6: ER mean value of error rate of estimation, group by ordering method

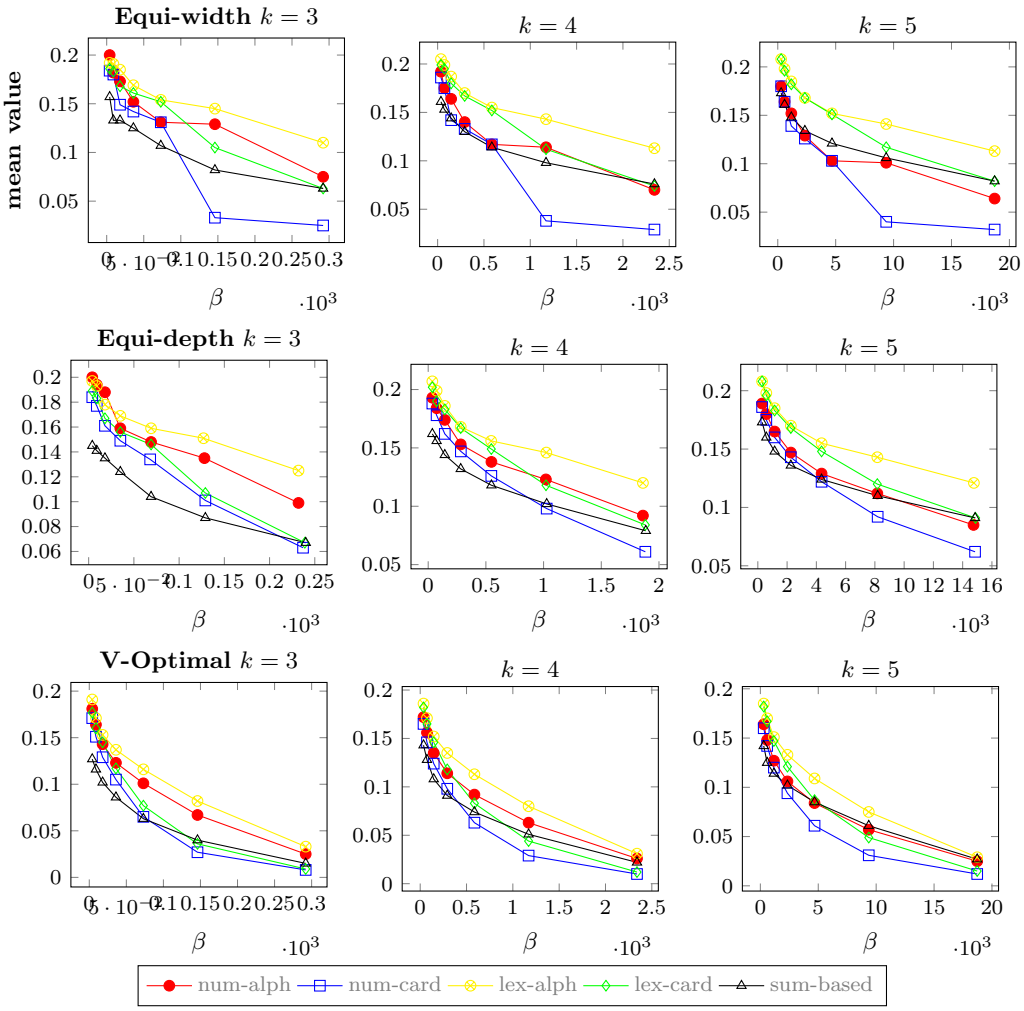


Figure 7.7: DBpedia subgraph mean value of error rate of estimation, group by histogram type

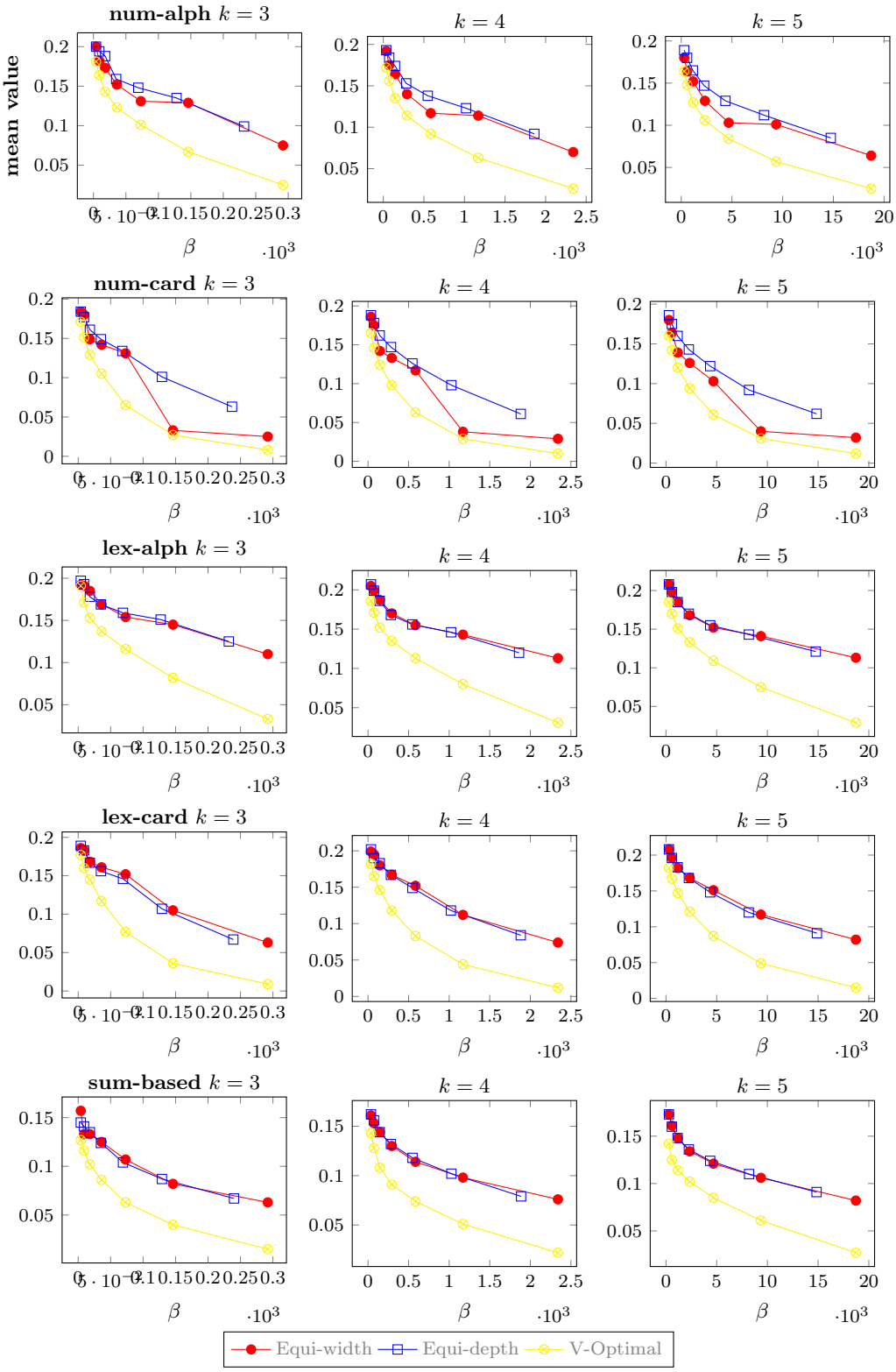


Figure 7.8: DBpedia subgraph mean value of error rate of estimation, group by ordering method

Chapter 8

Conclusions

8.1 Overview

In this thesis, we study the problem of path selectivity estimation in a graph database. This problem is required to be solved with three restrictions: low memory consumption, fast execution time and high estimation accuracy.

After giving definitions to all necessary notations and identifying the research question in Chapter 2, in Chapter 3 we examine the existing approaches in estimating path selectivity in a variety of areas, namely sampling, k-path histogram, path tree, Markov histogram, and others. We choose k-path histogram as the most promising method.

A current issue with k-path histogram is that its accuracy needs to be boosted. In order to identify this problem, in Chapter 4, we first study three most popular histograms: Equi-width, Equi-depth and V-Optimal. A framework is used in describing these histograms formally.

As the empirical study goes on, we notice the main reason for high error rate in estimation with a histogram is its high intra-bucket variance. Chapter 5 starts from here, describes the motivation of introducing ordering concept into our research.

We realize that ordering of a distribution has a huge impact on the estimation accuracy. An intuitive ordering method which arranges data distribution by the cardinality of each label path is proposed and then proven to be not efficient as it uses extra memory space. To overcome this shortcoming, we present novel framework for describing an ordering over a distribution. Several naive ordering methods are described by using this framework. Furthermore, we present data-aware ordering method which constructs an approximately monotonic sequence by utilizing small amount of extra memory.

Experiments conducted in Chapter 7 cover all three restrictions we mentioned in the beginning. The performance of different histograms and different ordering methods on datasets with different characteristics are presented and discussed in detail. Additionally, we identify the features a graph should have such that the accuracy of estimation can be mostly boosted. This is done from the perspective of a histogram and that of an ordering method.

8.2 Limitations and Future Work

There are three noticeable limitations in this work:

- Long construction time of V-Optimal histogram when the label path set $\mathcal{L}$ is large. Although it is not a focus of our research and can be covered by using more powerful hardware, people can still hope for a better performance. Fortunately, advanced histogram schema is always an active research topic.
- Time complexity of sum-based ordering method is high. In the thesis, we are handling datasets with small $|L|$, i.e. the number of unique edge label, and histograms with small k, the result from Table 7.2 indicates the performance in terms of running time is not a big issue. However, this algorithm is not optimized and we expect improvements on it from future work.
- Inability of handling heterogeneous graphs. A possible research direction is described in Section 8.2.4.

During the research, besides those that have been presented in previous chapters, we have many other ideas as the result of discussions, each of which can be a promising research direction. The ideas include but not limited to the following:

8.2.1 Expand the framework

In our study, all ordering methods utilize only the edge label set L, i.e. $\mathcal{L}^1$. The idea is to store more than just that in extra memory to construct the approximately monotonic sequence. For example, use $\mathcal{L}^2$. The challenge is to find an efficient and valid algorithm to unrank a given label path, without generating the whole label path sequence in the new order.

Edge label set L only stores basic information of edge degree. With the access to the distribution of length-2 label paths, we have the knowledge of **correlation** in the graph. Hence, the potential of this idea is huge.

8.2.2 Sum-based biased frequency approximation

This idea is inspired by the same hypothesis as the *sum-based* ordering. Currently, we are making *uniform distribution assumption*, which assumes the frequency within a histogram bucket is uniformly distributed. *Sum-based biased frequency approximation* assumes the cardinality of a label path ℓ is determined by the degree of edge labels which are decomposed from ℓ. The algorithm calculates a factor for each label path within a bucket, which then is used to scale up or down the mean value of cardinalities, to finally produce the estimation.

8.2.3 Data-aware approximation/calculation

This idea is similar to the second. It performs on a larger scale, i.e. the whole data distribution. Instead of using buckets, it stores necessary information of a graph and constructs a complex mathematical formula to calculate the estimation.

Very roughly, an instance of this approach can be constructed as following: Firstly, the program calculates the cardinalities for the label path set $\mathcal{L}^k$ with some large k, extracts properties such as how fast the cardinality scales up or down as k grows, does the pattern vary when k is small and when k is large, and so on. Secondly, the program stores cardinalities for all label path in $\mathcal{L}^k$ with a small k, and constructs a formula which takes the cardinalities and the properties as parameters and produces estimation for given label path. Further, there are other parameters to be tuned for different dataset.

8.2.4 Handling heterogeneous graphs

The problem with heterogeneous graphs is that we don't know which two nodes are potentially connectable. A possible solution to this is we first summarize and extract the patterns of the existence of connections, then classify nodes into corresponding categories or mark nodes with properties, finally use categories or properties to determine whether there could be a connection between two nodes.

Bibliography

[1] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB*, pages 591–600. Morgan Kaufmann, 2001. 7

[2] Arnab Bhattacharya. *Fundamentals of Database Indexing and Searching*. Chapman & Hall/CRC, 1 edition, 2014. 3

[3] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012. 12

[4] DBpedia. Dbpedia, 2017. 35

[5] Paul Erdos and A. Renyi. On random graphs i. *Publ. Math. (Debrecen)*, 6:290, 1959. 35

[6] George H. L. Fletcher, Jeroen Peters, and Alexandra Poulovassilis. Efficient regular path query evaluation using path indexes. In Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amlie Marian, Letizia Tanca, Ioana Manolescu, and Kostas Stefanidis, editors, *EDBT*, pages 636–639. OpenProceedings.org, 2016. 5, 7

[7] Yannis E. Ioannidis. The history of histograms (abridged). In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *VLDB*, pages 19–30. Morgan Kaufmann, 2003. 7, 12

[8] KONECT. Adolescent health, 2016. 35, 36

[9] Zoe Lacroix, Hyma Murthy, Felix Naumann, and Louiqa Raschid. Links and Paths through Life Sciences data sources. In *Proc. of the 1st International Workshop on Data Integration in the Life Sciences (DILS)*, 2004. 1

[10] Ulf Leser. A query language for biological networks. In *ECCB/JBI*, page 39, 2005. 1

[11] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636, New York, NY, USA, 2006. ACM Press. 9

[12] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proc. of KDD'05*, 2005. 35

[13] Mauro San Martn, Claudio Gutierrez, and Peter T. Wood. Snql: A social networks query and transformation language. In Pablo Barcel and Val Tannen, editors, *AMW*, volume 749 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011. 1

[14] Max Planck Institute for Informatics. Yago, 2017. 35

[15] Neo Technology. Neo4j, 2017. 1

[16] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database system concepts*. McGraw-Hill, New York, 6 edition, 2010. 1

[17] Ahmet Soylu, Felix Modritscher, and Patrick De Causmaecker. Ubiquitous web navigation through harvesting embedded semantic data: A mobile scenario. *Integrated Computer-Aided Engineering*, 19(1):93–109, 2012. 1

[18] Stanford SNAP Group. Snap.py, 2017. 35

[19] Silke Tril and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 845–856, New York, NY, USA, 2007. ACM. 8

[20] Silke Tril and Ulf Leser. Estimating result size and execution times for graph queries. In Mirjana Ivanovic, Bernhard Thalheim, Barbara Catania, and Zoran Budimac, editors, *ADBIS (Local Proceedings)*, volume 639 of *CEUR Workshop Proceedings*, pages 11–20. CEUR-WS.org, 2010. 8

[21] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Using histograms to estimate answer sizes for xml queries. *Inf. Syst.*, 28(1-2):33–59, 2003. 8