Selectivity Estimation of Twig Queries on Cyclic Graphs

Yun Peng, Byron Choi, Jianliang Xu

Department of Computer Science, Hong Kong Baptist University, Hong Kong, China

{ypeng, bchoi, xujl}@comp.hkbu.edu.hk

Abstract

Recent applications including the Semantic Web, Web ontology and XML have sparked a renewed interest on graph-structured databases. Among others, twig queries have been a popular tool for retrieving subgraphs from graph-structured databases. To optimize twig queries, selectivity estimation has been a crucial and classical step. However, the majority of existing works on selectivity estimation focuses on relational and tree data. In this paper, we investigate selectivity estimation of twig queries *on* possibly cyclic *graph data. To facilitate selectivity estimation on cyclic graphs, we propose a matrix representation of graphs derived from prime labeling — a scheme for reachability queries on directed acyclic graphs. With this representation, we exploit the* consecutive ones property (C1P) *of matrices. As a consequence, a node is mapped to a point in a two-dimensional space whereas a query is mapped to multiple points. We adopt histograms for scalable selectivity estimation. We perform an extensive experimental evaluation on the proposed technique and show that our technique controls the estimation error under 1.3% on* XMARK *and* DBLP*, which is more accurate than previous techniques. On* TREEBANK*, we produce* RMSE *and* NRMSE *6.8 times smaller than previous techniques.*

1 Introduction

Graph-structured databases have a wide range of emerging applications, *e.g.*, the Semantic Web, eXtensible Markup Language (XML), biological databases and network topologies. Up-to-date, there has already been voluminous real-world (possibly cyclic) graph-structured data [3]. To retrieve subgraphs from a large graph-structured database efficiently, various query optimization techniques have been proposed. Among others, selectivity estimation of queries has been a crucial support for query optimization technique in databases. In particular, selectivity estimation has been built into the query optimizer of all commercial relational databases. In a nutshell, given a query, we want to determine the number of results of the query, without invoking potentially costly query evaluation. However, the majority of previous research on selectivity estimation, with a few exceptions (see Section 2), focuses on relational and tree-structured data. In this paper, we propose *histogram-based selectivity estimation of twig queries for possibly cyclic graphs.*

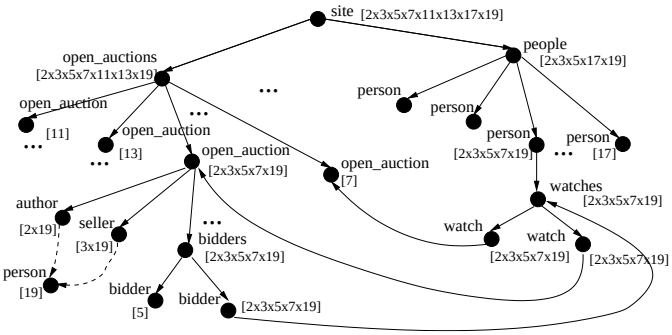


Figure 1. An example graph of auction information (XMARK) with its prime labeling

Twig queries have been a popular and classical tool for retrieving subgraphs from a graph-structured database. To facilitate our technical discussions, let us consider a twig query over a simplified XMARK (cyclic) graph [18]. The graph (Fig. 1) encodes auction information, where people watch over auctions and bidders bid items. Consider a sample query `//person[//open_auction//person]`, which selects the `person`s who watch some auctions that are still open and related to some other `person`s. Note that the twig query is recursive, where the query selects `person`s that have some `person` descendants. In XMARK with a scaling factor 1.0, there are 25,500, 12,000, and 13,192 `person`s, `open_auction`s and `open_auction//person`s, respectively. Based on selectivity information, a query optimizer should produce a plan that evaluates `//open_auction//person` prior to `//person`, to minimize intermediate result size.

Recently, there have been studies on selectivity estimation of XML data, *a.k.a.* tree data, and twig queries. In particular, Wu *et al.* [24] propose to adopt histograms for

selectivity estimation of XML queries. An advantage of this technique is that histograms are by far the most popular technique for query result estimation. The proposed technique relies on an interval representation of nodes, which presumes tree data. That is, each node of a tree is associated with one interval. The challenge in adopting the interval representation to cyclic graphs (and even directed acyclic graphs) is that multiple intervals may be associated with a node [2], as there may be muliple paths between any two nodes. Thus, the storage requirement of this technique on cyclic graphs can be prohibitive. In addition, it does not appear straightfoward to extend the existing estimation framework [24] to support multiple interval representation either.

Regarding cyclic graphs, there has been a work, namely XSKETCH [16], that exploits (local) minimal bisimulation of a graph for selectivity estimation of path queries. When compared to the histogram approach, bisimulation is not yet available in any commercial database and there has not been a *de facto* external representation of bisimulation graphs. Another drawback of local bisimulation is that the estimation accuracy relies on a strong statistical assumption (uniform distribution) of data.

In this paper, we propose a novel selectivity estimation technique for twig queries on cyclic graphs. The novelties of the technique are twofold. First, different from [16], we undertake a histogram approach to conduct selectivity estimation; we use auxiliary histograms to tackle possibly skewed data and do not make any assumption on the data distribution. To facilitate summarization of data, we propose a prime number labeling scheme (or simply *prime labeling*) to represent (cyclic) graph data, which was originally proposed for tree data [23]. (We defer the discussion on the drawbacks of other alternative representations to Section 2.) With prime labeling, the checking of the descendant-ancestor relationship among nodes of graphs and (later) estimation methods become very simple. Specifically, previous prime labeling scheme essentially associates each node with an exclusive prime number and labels each node with the product of its children's labels and its own prime number. Reachability between nodes is simply mapped to divisibility test of labels. Unlike previous works, our prime labeling requires fewer prime numbers for labeling and is therefore smaller in size.

A known issue of prime labeling is that it often results in very large integers. The second novelty lies in a new binary matrix representation of prime labeling, which further reduces the labeling size. In this way, we bridge selectivity estimation to the work on matrices. In particular, we transform a cyclic graph into a matrix with the *Consecutive Ones Property (C1P)*. Subsequently, a node of a cyclic graph can be represented as an interval of column IDs, (*start*, *end*).[1] Querying is then done by logical operations on the matrix. Nodes are essentially summarized in a two-dimensional histogram. In matrix transformations, new columns are often introduced. We store mappings between equivalent column IDs in a compressed form. Given a query, we translate it into multiple equivalent queries (intervals) with the compressed mappings. Such interval representations of data and queries make histograms a feasible solution for summarization.

The contributions of this paper are as follows.

- To the best of our knowledge, this is the first work on selectivity estimation of twig queries on cyclic graphs. Previous works focus on either twig queries or cyclic graphs but not both.

- We propose a prime labeling scheme to represent cyclic graphs and a binary matrix representation of prime labeling (Section 5). We transform the matrix in order to map a node of a graph to an interval and in runtime, a query to possibly multiple intervals. A two-dimensional histogram is used to summarize the matrix (Section 6). We propose an estimation algorithm with the histograms (Section 7).

- We perform a performance evaluation (Section 8) that verifies our technique controls the estimation error under 1.3% for XMARK and DBLP datasets . In comparison, the previous work XSKETCH/TREESKETCH [16, 17] reports that it controls the error under 5%. On TREEBANK dataset, our implementation produces RMSE and NRMSE that are at least 6.8 times smaller than XSEED's [25].

2 Related Work

There have been some recent works on selectivity estimation for path or twig queries on trees or cyclic graphs. The techniques can be roughly classified into two categories: graph-based approach and relational-based approach.

Graph-based approach. While graph-structured data model has its root at network data model, it was revisited in Tsimmis project, in which Object Exchange Model (OEM) is proposed. DATAGUIDE [14] is proposed to summarize the paths of OEM. Graphs are considered as NFA and their DATAGUIDEs are DFA of the graphs. DATAGUIDE has been extended to support approximate query processing [8]. Straight-Line Grammar (STL) [6] is a special form of context-free grammar, for summarizing a data graph. To reduce the size of the grammar, [6] proposes to use a wildcard to simplify some non-terminals in a production.

[1] Our interval represents the column IDs of a C1P matrix. In contrast, the multiple intervals of nodes in [2] represents the preorder and postorder numbers of a traversal on the spanning tree of a DAG and the connectivity due to non-tree edges.

Another graph-based approach [16, 17] (XSKETCH and later TREESKETCH) is derived from bisimulation of graphs. XSKETCH supports only path queries on cyclic graphs. TREESKETCH, on the other hand, supports twig queries on acyclic graphs only. In comparison, we support twig queires on cyclic data. [16, 17] propose to adopt bisimulation as the synopses of a data graph. To further reduce the size of bisimulation, a notion of local bisimulation [10] has been applied. To recover the path information from a local bisimulation graph, graph stability is exploited and uniform distribution of nodes is assumed. Unlike their techniques, we do not assume the data exhibits uniform distribution but use auxiliary histograms to summarize skewed data. A recent survey shows that some popular graph (XML) benchmarks contain highly skewed data [13]. Our overall technique adopts histogram, which is by far the most popular selectivity estimation technique.

Correlated subpath tree (CST) [4] stores the count of small twigs (*branches*) in data trees. It has been shown by recent experiments that [16, 17] outperform [4]. XSEED [25] initially derives a compact path summary (kernel) from data trees and adaptively tunes memory budgets of summaries based on query workload. Its experiment showed XSEED outperforms TREESKETCH [17] when 1000 queries are considered. In our experiments, we compare our techniques with XSEED. STATIX [7] proposes to count subtrees in XML, not cyclic graphs, with schema information. In contrast, we do not assume schemas.

Relational-based approach. Histograms from relational databases have been adapted to support selectivity estimation of queries on graphs. [24] proposes an interval representation of nodes of a tree. The start and end position of the interval is used as the x and y coordinates of a point in a two-dimensional plane. A two-dimensional histogram and auxiliary histograms are used to summarize the points. Bloom histograms [21], path trees and Markov tables [1,12] have been proposed for path selectivity estimation for tree data. However, it is not clear how these techniques support cyclic graphs, which contain infinitely many paths, for selectivity estimation.

Alternative representation of graphs. In this work, we adopt prime labeling [22, 23] as the representation of cyclic graphs, due to its simplicity. In addition to the interval representation discussed earlier, there have been alternative representations. Transitive closure of the graph G consists of an entry (u, v) if u can reach v in G. However, its storage is prohibitive $O(|G|^2)$. Adjacency matrix has been a classical representation of graphs. However, determining the ancestor-descendant relationship in an adjacency matrix is relatively complex, which requries taking self-products of the matrix. There has been a host of ad-hoc indexes for reachability queries on graphs, *e.g.*, 2-hop labeling [5]. However, the structures of ad-hoc indexes are often complex and their summarization does not seem to be straightforward.

3 Definitions and Preliminaries

We begin our technical discussions with the definitions and notations used.

3.1 Data Model

In this paper, we study directed node-labeled rooted data graphs, or simply *graphs* in the subsequent discussions. A graph can be denoted as $G = (V, E, r, \Sigma, \lambda, oid)$, where V is a set of nodes and E: $V \times V$ is a set of edges, $r \in V$ is a root node, Σ is a set of tags and λ: $V \rightarrow \Sigma$ is a function that returns the tag of a node and oid is a function that returns a unique identifier of a node. For simplicity, we may denote a graph as (V, E) when other components are irrelevant.

3.2 Twig Queries

Among the queries on graphs, XPATH has been studied more extensively recently than others and it has been an indispensable part of eXtensible Markup Language (XML) — the *de facto* standard for electronic data exchange. Hence, we consider a fragment of structural XPath — *twig queries*. The syntax is given in BNF below:

$$\begin{aligned} p &::= \epsilon \mid A \mid * \mid // \mid p/p \mid p[q], \\ q &::= p \mid q \wedge q \mid q \vee q, \end{aligned}$$

where ϵ, A, $*$ and $/$ denote the *self-axis*, a tag, a wildcard and the *child*-axis, respectively; $//$ stands for */descendant-or-self::node()/*; and q in $p[q]$ is called a *filter*, in which $\wedge$ and $\vee$ denote conjunction and disjunction, respectively. For $//$, we abbreviate $p_1/\ //$ as $p_1//$ and $//\ /p_2$ as $//p_2$. For simplicity, our technical discussion focuses on // axes, while the extension to / axes can be addressed by introducing an index on the depth of nodes. We use $r[|p|]$ to denote the evaluation of the query p from the node r.

Problem statement. Let R be the set of nodes of the evaluation, where $R = r[|p|]$. In this paper, given p and r, we want to determine $|R|$ efficiently and accurately.

3.3 Consecutive Ones Property

Next, we provide the definition of the *Consecutive Ones Property* (C1P), which is useful to summarize the ones (non-zeros) in a matrix. In this paper, we represent a cyclic graph with a *binary matrix*, denoted by M. The u-th row is denoted as $M[u]$. The entry at the u-th row and the v-th column, denoted as $M[u][v]$, can be either '0' or '1'.

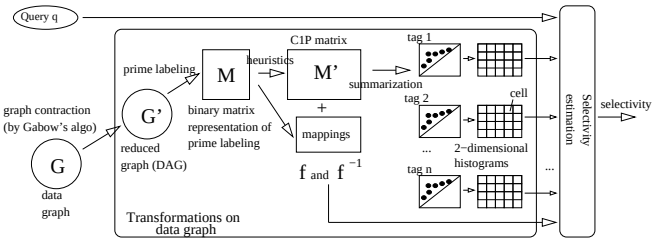


Figure 2. An overview of our proposed technique

Definition 3.1: A matrix M has the *weak Consecutive Ones Property (C1P)* if its columns can be permuted such that in each row, the ones are adjacent. A matrix M has *strong C1P* if the ones of each row are adjacent. ∎

For simplicity, we call a matrix with strong C1P a *CIP matrix*. Since the ones in a row of a C1P matrix are adjacent, we can represent the ones in the row with a start and end column number which can be considered as the x- and y-coordinates of a data point. In subsequent discussions, we may use the term intervals and data points interchangeably.

4 Overview

In this section, we provide an overview of the design of our proposed technique to the selectivity estimation problem.

Consider a cyclic data graph G, as shown in Fig. 2. G is first reduced into a DAG (G'), on which prime labeling is applied. We design a prime labeling scheme that not only facilitates simple query processing but also selectivity estimation of twig queries. Furthermore, we propose a new binary matrix representation M of prime labeling. Its advantages are twofold. First, prime labeling may require large prime numbers for labeling large graphs since at least each leaf node needs a unique prime number. Second, we can adopt existing work from matrices, for summarization. In particular, we transform the binary matrix M into a C1P matrix M' and represent graph nodes with two-dimensional data points. As a consequence, a well-known estimation technique, two-dimensional histogram, can be used to summarize the data points of each kind of tag, where the histogram's cell size ρ can be tuned for space or estimation accuracy.

Regarding estimating the selectivity of a twig query q, we encounter a unique challenge that data points derived from the C1P matrix M' are often highly skewed. When developing the estimation algorithm, we observe that there are sometimes few queries with large errors, which lead to a poor overall accuracy. Hence, we propose additional auxiliary histograms for summarizing skewed data points and do not opt to adopt [24] for selectivity estimation.

In what follows, we discuss the prime labelling and binary respresentation of cyclic graphs in Section 5, matrix transformations in Section 6, and selectivity estimation of twig queries in Section 7.

5 Representation of Cyclic Graphs

Most labeling techniques on descendant-or-self axis focus on tree data. It is not clear how these labelings can be modified to support cyclic graphs. For example, path-based labelings do not work in cyclic graphs as there are infinitely many paths and the interval labeling [2] has a high space requirement for cyclic graphs. In this section, we present a new representation of cyclic graphs based on prime labeling [23], to efficiently estimate the descendant-or-self axis on cyclic graphs.

5.1 The Original Prime Labeling

Prime labeling was originally proposed for indexing trees. The main idea of prime labeling is that each node is labeled with a product of prime numbers such that the ancestor-descendant relationship between nodes could be determined by using the division of the prime labels. A node n_1 is an ancestor of another node n_2 if and only if the label of n_1 is *divisible* by that of n_2. In [23], a unique prime number is assigned to each leaf node. The prime label of an internal node is the product of the prime labels of its children. Such labeling works on trees only. To extend prime labeling to DAGs, [22] requires a unique prime number per node.

5.2 Prime Labeling for Cyclic Graphs

To support cyclic graphs, we follow the standard preprocessing to reduce each SCC into a supernode and apply prime labeling on the reduced graph. We propose two modifications on prime labeling: First, the previous work [22] on prime labeling uses excessive prime numbers (one prime number per node). We propose to use fewer number of (unique) prime numbers needed for labeling and hence reduce the overall size of prime labeling. More specifically, we require a new unique prime number for labeling a node n in one of the following scenarios: (i) n is a leaf node; or (ii) all the children of n have more than one parents. Regarding the second scenario, if a new prime number is *not* used for labeling n, then it is possible to have a node n' whose label is divisible by n's label but n' is not an ancestor of n, since n' can be an ancestor of other parents of n's children. Second, prime labeling needs to support possibly multiple strongly connected components (SCCs) in cyclic graphs. By definition, each node in an SCC can reach any other node in

the SCC. Therefore, the nodes in an SCC can be associated with the same prime label.

Next, we present the definition of prime labeling for cyclic graphs. Let `get_next()` be a special function which returns a prime number that has not been returned before. Assume that a cyclic graph G has been preprocessed by Tarjan's algorithm [20], where each SCC is reduced to a supernode. Denote the reduced graph to be $G'(V', E')$. Each node n is associated with a prime label ℓ as defined below.

Definition 5.1: The *prime label* ℓ of a node n of the reduced graph $G'(V', E')$ can be defined as follows:

1. If n is a leaf node, then $n.\ell$ = `get_next()`.
2. If n is a non-leaf node and all the children of n have multiple parents, then $n.\ell$ = `get_next()` $\times \prod_{c \in C} c.\ell$, where C is the set of n's children.
3. Otherwise, $n.\ell = \prod_{c \in C} c.\ell$.

∎

The prime labels of the nodes of a reduced graph G' are assigned in a reverse-topological order, *i.e.*, a bottom-up traversal. The pseudo-code of the prime labeling construction (`prime-construct` in Fig. 3) can be readily dervied from Definition 5.1. It assigns prime labels to the reduced graph (G'). In Line 01, we apply Tarjan's algorithm to reduce a cyclic graph G into a DAG G', where an SCC is reduced to a supernode. We initalize the prime label of each node to be 1 in Line 02. Then, we assign the prime labels of nodes in a reverse-topological order (bottom-up traversal). There are two possible cases. (1) If the node n is a leaf node (Lines 04-05), we assign a new prime number to the node. (2) If the node n is not a leaf node, the prime labels of the node is set to the product of the prime labels of its children in Lines 07-08. However, if all the children of the node have multiple parents, we assign an additional new prime number to the prime label of n (Lines 09-10), as argued earlier.

While `prime-construct` and [22, 23] assign prime numbers differently, querying with our prime labeling remains simple. Assume that we have a set of A-nodes S_A and B-nodes S_B. A naive way to determine the number of B-descendants in S_B of the nodes in S_A takes $O(|S_a| \times |S_b|)$. With prime labeling, this can be done by first computing the product of the prime labels of S_A, denoted by M_A, and then check the divisibility between M_A and the prime label of each node in S_B. This requires $O(|S_a| + |S_b|)$ only.

Example 5.1: Reconsider the XMARK graph shown in Fig. 1. The prime label of each node is shown in the square bracket. We show a strongly connected component whose nodes have the same label $2 \times 3 \times 5 \times 7 \times 19$, as they can reach one another, by definition. The `person` with label 19 is both a `seller` and an `author`. We use a new prime label for the `author` (2) and `seller` (3). Since 2×19 and 3×19 are not divisible, `author` and `seller` are not a descendant of each other. Furthermore, to check the number of `person`s that are a descendant of some `open_auction`s, we can simply check the divisibility between the `person`'s label, *e.g.*, 17, to the label of `open_auction`s, *i.e.*, $2 \times 3 \times 5 \times 7 \times 11 \times 13 \times 19$. ∎

Input: A data graph G
Output: A data graph with prime labeling

```
01 G' = tarjan(G)
02 initialize the prime label of nodes in G' to 1
03 for each n in G'.V in reverse topological order
04   if n is a leaf node                /* Definition 5.1 */
05     n.ℓ = get_next()
06   else
07     for each c in n.children
08       n.ℓ = n.ℓ × c.ℓ
09     if ∀ n' ∈ n.children. n' has multiple parents
10       n.ℓ = get_next() × n.ℓ
```

Figure 3. Prime labeling construction `prime-construct`

5.3 Matrix Representation of Cyclic Graphs

Given voluminous graph data, such as biology pathways, social networks and XML, prime labeling may result in very large integers. To address this issue, we propose a binary matrix representation of prime labeling and map integer divisions simply to logical operators of vectors.

Definition 5.2: Suppose that the prime label ℓ of a node n of a graph G is $p_{i_1} \times p_{i_2} \times ... \times p_{i_m}$, where p_{i_j} is the i_j-th prime number. ℓ is then presented by a vector $\vec{\ell}$ where $\vec{\ell}[i_j] = 1$ if and only if p_{i_j} is a factor of ℓ; and $\vec{\ell}[i_j] = 0$ otherwise. The size of the vector is the total number of prime numbers used in labeling G. ∎

A graph is represented as a set of binary vectors which form a matrix. Here, we always discuss binary vectors and matrices. For simplicity, we may omit the term "binary".

With this representation, divisions and multiplications of prime labels can be mapped into logical operators on the vector representation of the prime labels.

Definition 5.3: Given two nodes n_1 and n_2, $n_1.\ell$ is divisible by $n_2.\ell$ if and only if $\neg(n_1.\vec{\ell}) \wedge n_2.\vec{\ell} = \vec{0}$. ∎

Definition 5.3 can be alternatively understood that the vector $\neg(n_1.\vec{\ell})$ and $n_2.\vec{\ell}$ are *orthogonal*, where the product of the two vectors is 0.

Definition 5.4: Given a set of nodes V and n_2, $\prod_{n \in V} n.\ell$ is divisible by $n_2.\ell$ if and only if $\neg(\bigwedge_{n \in V} n.\vec{\ell}) \wedge n_2.\vec{\ell} = \vec{0}$. ∎

To end this section, we remark that `prime-construct` (presented in Fig. 3) can be used with minor modifications to compute the binary matrix representation *directly* from cyclic graphs.

6 Matrix transformations

In this section, we present the transformation of the binary matrix of prime labeling into a C1P matrix for simple summarization. On one hand, a C1P matrix can be readily summarized by a set of intervals, as discussed in Section 1. On the other hand, converting a matrix into a C1P matrix is intractable [19]. Worst still, there is no polynomial time approximation scheme for determining a C1P submatrix in a given matrix. Therefore, we propose (i) a heuristic algorithm for converting a matrix into a C1P matrix and (ii) two practical optimizations, namely, horizontal decomposition on the matrix and extraction of the largest common subset of non-zeros in the decomposed submatrices, to reduce the size of the matrix passed to the heuristic algorithm.

6.1 Transforming to C1P Matrix

The heuristic algorithm uses a C1P detection algorithm proposed by Hsu [9] as a component, which determines if a matrix is C1P or not and has been known to have simple implementations. The overall heuristic algorithm `heuristic_c1p` is presented in Fig. 4.

We assume that the rows of the input matrix M are assumed to be sorted by the number of non-zeros in descending order. The heuristic algorithm is to first process the rows that have more overlapping non-zeros with the first row. The idea is that there may be a higher chance for such rows to share more columns containing non-zeros. Subsequently, we may obtain a smaller C1P matrix.

The details of `heuristic_c1p` are as follows. We first compute the overlappings between the rows in M with the first row — the row with the most number of non-zeros (Lines 01-03). Then, we sort the rows by the amount of overlappings (Line 04) and construct a new C1P submatrix R (Line 05). We may merge a row $M[i]$ into R if one of the three conditions is satisfied (Lines 07-09): (i) Hsu's algorithm (denoted as `c1p_detect`) reports that $M[i]$ can be merged to R to form a C1P matrix. (ii) $M[i]$ does not overlap with R. (iii) $M[i]$ is contained in some rows in R. We remark that Conditions (ii) and (iii) do not arise in [9]. However, such a row can be readily merged into the C1P matrix R.

In Line 11, `column_partition` is the COLUMN-PARTITION algorithm in [9] extended to handle Conditions

Procedure `heuristic_c1p`
Input: A matrix representation of a cyclic graph $M[\,][\,]$, where the rows of M are sorted by # of non-zeros (descending)
Output: The C1P matrix from M

01 $r = M[0]$ /* 1st row */
02 **for each** i **in** [1...m-1], where m is the number of rows of M
03 $M[i]$.`overlap` = $|\{\ j \mid M[i][j] \wedge r[j],\ j \in [1...n]\}|$
04 sort M by the `overlap` attribute of the rows
05 $R = \{r\}$
06 **for each** i **in** [1...m-1]
07 **if** (i) `c1p_detect`($R \cup \{M[i]\}$) **or** /* [9]*/
08 (ii) $M[i] \wedge \bigwedge R = \emptyset$ **or** /* non-overlapping row*/
09 (iii) $\exists\ j$ s.t. $M[i] \wedge R[j] = M[i]$ /* $M[i]$ in $R[j]$*/
10 **then**
11 R = `column_partition`(R, $M[i]$) /* Section 6.1 */
12 **return** $R \oplus$ `heuristic_c1p`(M - R)

Figure 4. Heuristic C1P transformation

(ii) and (iii). In a nutshell, assuming that R and r form a C1P matrix, `column_partition`(R, r) reorganizes the columns of R and r in partitions such that a C1P matrix can be trivially generated from the partitions. Due to space constraints, we opt to present COLUMN-PARTITION as a black box.

Finally, we recursively call `heuristic_c1p` to process the remaining rows, until the whole matrix is transformed into a C1P matrix (Line 12). A subtle note is that the C1P submatrix R constructed from each call of `heuristic_c1p` is often not mergable to each other. Otherwise, these submatrices may be returned in a single call of `heuristic_c1p`. Hence, we append (denote as $\oplus$) the submatrix, returned from recursive calls, to R.

The operator $\oplus$ is a special append operator. Suppose R_1 and R_2 is a n_1 by m_1 matrix and n_2 by m_2 matrix, respectively. $R_1 \oplus R_2$ returns a (n_1 + n_2) by (m_1 + m_2) matrix R', where R_1 and R_2 are placed at the top-left and bottom-right corner of R', respectively. Fig. 5 illustrates $\oplus$ and the heuristics `heuristic_c1p` with a sketch of the run of `heuristic_c1p`. (A real example of a C1P matrix produced by `heuristic_c1p` is presented in Fig. 7.) `heuristic_c1p` generates a C1P matrix recursively.

Mappings between columns and positions. In general, a column of a matrix may be duplicated in multiple submatrices returned by `heuristic_c1p`. To avoid confusions, we refer the columns of the C1P matrix to *positions*. Two mappings are needed to record the relationship between the column and its positions. In particular, we store the mappings in two binary relations f and f^{-1}, where $f(v_i)$ returns the positions of v_i in V and $f^{-1}(p)$ returns v_i where $p \in f(v_i)$.

Analysis. The runtime of Hsu's algorithm (`c1p_detect`

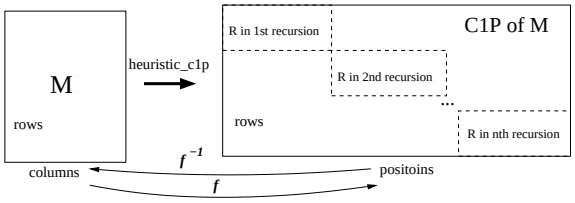


Figure 5. Schematics of heuristics `heuristic_c1p`

and `column_partition`) is $O(m+n+r)$, where m, n and r are the number of rows, the number of columns and the number of non-zeros. The loops (Lines 02 and 06) iterate through the rows of a matrix. There are at most m recursive calls. Thus, the time complexity of `heuristic_c1p` is $O(m^2 \times (m+n+r))$.

6.2 Optimizing Matrix Transformation

This subsection presents two optimizations for matrix transformation that are specific to our approach.

We make two observations on the matrix representation of a cyclic graph, constructed as in Section 5. First, the prime labels are assigned essentially bottom-up, where the rows (the nodes) near the root have relatively more non-zeros. That is, the number of non-zeros of the rows (the nodes) near the root is often very different from those near the leaf nodes. Second, since the nodes in an SCC can reach one another, the rows of an SCC are identical. Hence, we propose two matrix manipulations to reduce the size of the matrix passed to `heuristic_c1p`.

6.2.1 Horizontal Matrix Decomposition

The patterns in the rows with many non-zeros are often different from those with few non-zeros. We propose a simple decomposition to separate these rows of a matrix M and summarize them separately. First, note that the order of rows does not carry any information. We sort the rows by the number of non-zeros, in descending order. Denote by $\bar{M}$ and σ, respectively, the mean and standard deviation of the number of non-zeros for all rows of M. Second, we scan the sorted matrix. Let R be the rows scanned thus far and r be the next row in the scan. If the number of non-zeros of r is beyond $\bar{R} - c\sigma$, where c is a constant, *e.g.*, 3, this indicates the remaining rows in the matrix are significantly different from those in R. Hence, we decompose the matrix at r and then continue the scan.

6.2.2 Common Pattern Extraction

A pattern that appears in *all* rows of a matrix contains little information. Therefore, we extract the largest common pattern of a matrix in a scan of the matrix, which can be incorporated with the decomposition discussed above. In the scan, we maintain the current common pattern P of the scanned rows. Assume r is the next row in the scan. The next largest common pattern is simply defined as $P \wedge r$.

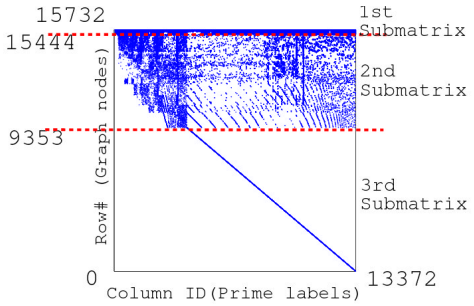


Figure 6. Matrix representation of XMARK

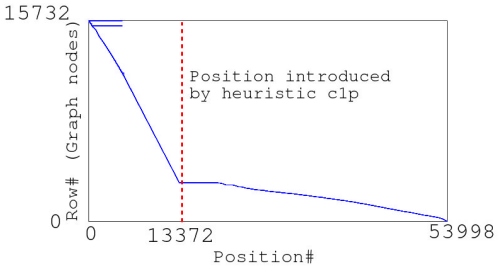


Figure 7. C1P matrix representation of XMARK

Example 6.1: In Fig. 6, we show the matrix representation of XMARK (with the scaling factor 0.01) after sorting the rows by the number of non-zeros (prime numbers). A dot '.' and a blank space ' ' represent a non-zero and zero, respectively. The figure shows that there are three distinguishable submatrices with different non-zero densities. The dotted lines show the decomposition when $\bar{R} - 3\sigma$ is used. Most of the non-zeros of the matrix occur in the topmost submatrix. After we locate the common pattern in the submatrix, we extract it out from the submatrix. We find that the topmost submatrix has a large common pattern, containing 10,684 non-zeros. Finally, we apply `heuristic_c1p` on the decomposed matrices to obtain a C1P matrix shown in Fig. 7. The number of positions needed for 13,372 columns is 53,998. ∎

7 Selectivity Estimation

This section presents the details of using two-dimensional histograms to summarize the C1P matrix derived in Section 6 and perform selectivity estimation. We first discuss our data structures associated with the histograms (Section 7.1) and the overall estimation algorithm (Section 7.2) and then highlight its technical details (Sec-

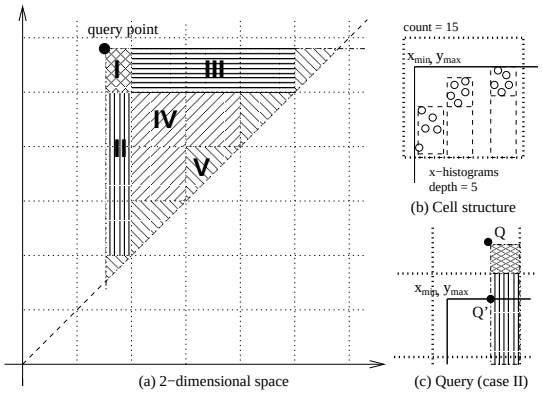


Figure 8. (a) Two-dimensional histogram; (b) Cell structure; and (c) Intermediate query

tions 7.3).

7.1 Two-dimensional Histograms

As discussed in Section 3.3, a C1P matrix can be represented by a set of (two-dimensional) data points (x, y)'s, where x and y are the start and end columns of 1's. We use a two-dimensional histogram for summarizing the data points of each kind of tag. The two-dimensional space is covered by a grid, which consists of non-overlapping cells. An example is shown in Fig. 8(a). The cell size is controlled by a parameter ρ. A data point (x, y), where $x \leq y$, resides in the upper diagonal area of the grid. To reduce space, we only consider the cells with data point(s).

Through our experiments, we found that the data points of our benchmark datasets are not evenly distributed on the two-dimensional space. First, the data points are skewed towards the diagonal line. Such phenonmenon has also been found in the data points of the interval approach [24], though the definition of their interval is different from ours. Second, the estimation rules in [24] assume data points are uniformly distributed along the diagonal line, which essentially integrate the area of possible regions containing some answer. In contrast, we associate three auxiliary data summaries to each cell to tackle skewed data points. An example of the cell structure is shown in Fig. 8(b). The auxiliary summaries are discussed below.

1. We introduce a tight bounding rectangle, defined by (x_{min}, y_{max}), of the data points of a cell, where x_{min} is the smallest x-coordinate and y_{max} is the largest y-coordinate of the data points in the cell.

2. We build equi-depth histograms based on the x-coordinate, where the depth of the histograms can be specified by a parameter φ. In addition, we keep the largest and smallest x and y values for each bin of the equi-depth x-histogram.

3. For the cells on the diagonal line, we further keep their data points, for partial query evaluation.

Procedure `top_down`
Input: A twig query p, a set of query points Q, a data graph G
Output: the count of p in G

01 **case of** p:
02 (i) $//A/p'$ /* A is a tag */
03 Q' = `estimate_intermediate`($//A$, Q) /*Sec. 7.3.1*/
04 **return** `top_down`(p', `equiv`(Q'), G)
05 (ii) $//A$
06 **return** `estimate_count`($//A$, Q) /* Sec. 7.3.2 */
07 (iii) $//A[q]/p'$
08 C_q = `bottom_up`(q, G, G)
09 Q' = `estimate_intermediate`($//A$, Q)
10 $Q' = \{g \mid g \in Q' \wedge$ `at_right_bottom`$(g, C_q)\}$
11 **return** `top_down`(p', `equiv`(Q'), G)
12 (iv) $//A[q]$
13 C_q = `bottom_up`(q, G, G)
14 **return** `estimate_count_with_Qf`($//A$, Q, C_q)

Figure 9. The overall estimation algorithm `top_down`

7.2 The Overall Estimation Algorithm

The estimation exploits a property of data points, which can be readily derived from Definition 5.4. A node v is a descendant of another node u if and only if the interval of v is *contained* in that of u. This is equivalent to say that v is a descendant of u if and only if the data point of v is at the bottom-right region of the data point of u. Fig. 8(a) shows the region containing the descendants of a query point. While the region is divided into five cases as in [24], our detailed esitmation exploits the auxilary structures to handle skewed data points each of the region.

With the above, we are now ready to present the overall estimation algorithm `top_down`, shown in Fig. 9. In a nutshell, `top_down` estimates the path of the twig query top down and invokes `bottom_up` to estimate the filters (branches) in the query. Some important technical details of `top_down` are given in Section 7.3.

The input of `top_down` is a twig query p, a set of query points Q and a data graph G. Initially, Q contains the root node, at where the evaluation starts. `top_down` proceeds according to the structural form of the query as follows (We omitted $\wedge$ and $\vee$ for presentation simplicity):

(i) If the query is $//A/p'$ (Lines 02-04), where $//A$ is an intermediate query, we compute *the next queries* (*i.e.*, points)

Procedure `bottom_up`
Input: A filter query q, a set of query points P, a data graph G
Output: the points that have some data points that satisfy q
01 **case of** q:
02 (i) $//A$
03 **return** `estimate_intermediate_reverse`($//A$, P, G)
04 (ii) $/p'//A$
05 P' = `estimate_intermediate_reverse`($//A$, P, G)
06 **return** `bottom_up`(p', `equiv`(P'), G)
07 (iii) $//A[q']$
08 P' = `bottom_up`(q', P, G)
09 $P = \{p \mid p \in P \wedge \exists\, p' \in P'\ p'$ is a descendant of $p\}$
10 **return** `estimate_intermediate_reverse`($//A$, P, G)
11 (iv) $/q'//A[q'']$
12 P' = `bottom_up`(q'', P, G)
13 $P = \{p \mid p \in P \wedge \exists\, p' \in P'\ p'$ is a descendant of $p\}$
14 P'' = `estimate_intermediate_reverse`($//A$, P, G)
15 **return** `bottom_up`($//q'$, `equiv`(P''), G)

Figure 10. Auxiliary procedure for handling filters `bottom_up`

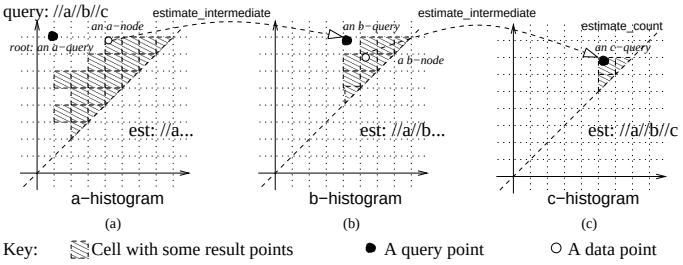


Figure 11. A schematics of the overall estimation algorithm

Q' from Q with `estimate_intermediate` (to be detailed in Section 7.3.1). Then, we proceed to estimate p'. Since columns may be represented by multiple positions, we need to process all the equivalent query points of Q' determined by `equiv` (to be detailed in Section 7.3.1).

(ii) If the query is the last step (Lines 05-06), we generate *the selectivity count* with `estimate_count` (to be detailed in Section 7.3.2).

(iii) Suppose the query contains a filter q ($//A[q]/p'$) (Lines 07-11). We determine the points C_q whose bottom-right region contains *some* points satisfying q (Line 08). In a nutshell, `bottom_up` is mostly symmetric to `top_down` and returns a set of points that satisfy the filter q. (Its details will be presented in the next subsection.) Then, we estimate the next query points Q' (Line 09) as in Case (i) but we only keep the query points that have some points in C_q in their bottom-right region (Line 10). Next, we estimate p' recursively, as in Case (i).

(iv) If the filter occurs in the last step ($//A[q]$), we need to generate the selectivity count, similar to Case (ii). However, we invoke `bottom_up` to find the query points C_q, similar to Case (iii). The difference between Line 06 and Line 14 is that when we generate the count, we only include the points that have some points in C_q in their bottom-right region.

Example 7.1: A partial run of `top_down` on an example query `//a//b//c` is shown in Fig. 11. The estimation starts with the root node (Fig. 11(a)). The root node becomes a query point that searches for a-descendant nodes, with `estimate_intermediate`. The shaded cells illustrate the cells that contain some a-nodes. The selected a-nodes become the next query points — b-queries. In Fig. 11(b), we show one b-query. Similarly, we determine the cells that contain some b-descendant nodes and subsequently c-queries. In Fig. 11(c), we show one c-query. Since `//c` is the last step, `estimate_count` is invoked to count the number of c-descendant nodes in the shaded cells. ∎

7.2.1 Details of `bottom-up`

This subsection provides the details of the handling of filters in the overall estimation algorithm `top_down` (Fig. 9). The filters are handled by Algorithm `bottom_up` shown in Fig. 10. The input of `bottom_up` is a filter q specified in the form of twig query, a set of intermediate query points P and a data graph G.

We first discuss `estimate_intermediate_reverse`, which is used in `bottom_up`. We skip its pseudo-code because it is straightforward. `estimate_intermediate_reverse`(q, P, G) returns a set of points that satisfy q in a graph G and has a point $p \in P$ in its bottom-right region.

Next, we focus on the structural recursion in `bottom_up`.

(i) If the filter query is $//A$, *i.e.*, the last step (Lines 02-03), we simply return the set of points (of cells) that contain some A-nodes which have some $p \in P$ in their bottom-right region.

(ii) If the filter query is not the last step ($//p'//A$), we first determine the points that satisfy $//A$ (Line 05). Then, we recursively determine the points that satisfy $//p'$ based on the result of $//A$ (Line 06).

(iii) If the filter query contains yet another filter query q', we invoke `bottom_up` recursively to first determine the points P' that satisfy q' first (Line 08). We keep only the points in P that have some points in P' in their bottom-right region (Line 09). Then, we determine the points of $//A$ as in Case (i) (Line 10).

(iv) This case (Lines 11-15) is similar to Case (iii). The difference is that after we determine the points P'' for $//A[q']$, we use P'' to determine the points for $//p'$ recursively.

Example 7.2: Recall from Algorithm `top_down` that `bottom_up` is invoked as `bottom_up`(q, G, G) (at Lines 08 and 13) and consider an example where q = `//a//b`. Note that `bottom_up` processes q bottom-up. Initially, we encounter Case (ii). We determine the nodes that satisfy `//b` and has a descendant in G. Hence, we obtain a set of all B-nodes in G as P' (Line 05). Next, we evaluate `//a` with all B-nodes (Case (i)). We obtain a set of A-nodes that have some B-descendant nodes. These nodes will be returned to `top_down` to filter query points. ∎

7.3 Estimation Details with Histograms

This subsection provides the technical details of the overall algorithm, specifically, `estimate_intermediate`, `estimate_count` and `equiv`.

7.3.1 Generation of Intermediate Queries

Given a query point, `estimate_intermediate` generates a set of next query points from five distinguishable cases in the bottom-right region of the query point. The five cases are visualized in Fig. 8(a).

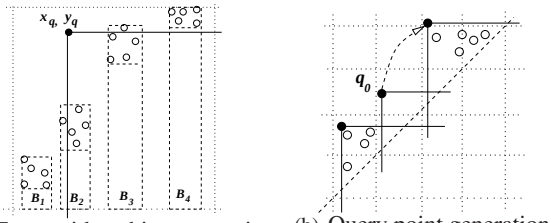
To illustrate how `estimate_intermediate` works, we present the estimation details with an example query `a//b//...`. Suppose the query point in Fig. 8(a) is an a-query and the histogram summarizes b-nodes. With reference to Fig. 8(a), we present the generation of b-queries below:

- *Cases I, II and III.* Suppose an a-query point is (x_q, y_q) and (x_{min}, y_{max}) represents the bounding rectangle of a cell of these cases. The b-query point is (`max`(x_q, x_{min}), `min`(y_q, y_{max})).
- *Cases IV and V.* The b-query point is simply (x_{min}, y_{max}) of a cell of these cases.

7.3.2 Generation of Result Count

`estimate_count` generates a count from the histogram. Similarly, assume that the query dot in Fig. 8(a) is an a-query (generated from `estimate_intermediate`) and the histogram summarizes b-nodes. We present the generation of the count of b-nodes of the query $a//b$ as follows:

- *Cases I.* The count of the result points of a query point (x_q, y_q) in the cell is estimated to be the the sum of the count of the bins that contain data points with an x-coordinate larger than x_q *and* a y-coordinate smaller than y_q. We illustrate this with Fig. 12(a). The estimated count is 10 (from B_2 and B_3).



(a) Est. with x-histograms in grid (b) Query point generation

Figure 12. Estimation with x-histograms and query point generation

- *Case II.* The count is estimated to be the sum of the count of the bins with some data point whose x-coordinate is larger than x_q.
- *Case III.* This is similar to the above case except that we check the y-coordinates and y_q.
- *Case IV.* We simply return the count of the cell.
- *Case V.I.* If the query point is not in the diagonal cell, we simply return the count of the cell.
- *Case V.II.* If the query point is in the diagonal cell, we check the bins with some data point whose x-coordinate is larger than x_q as follows: (i) If the bin's largest y is smaller than y_q, we simply include the count of the bin. (ii) If the bin's smallest y is larger than y_q, we skip the bin. (iii) Otherwise, we *evaluate* the query with the bin. Evaluation is invoked because as (x_q, y_q) approaches the diagonal line, there are fewer data points in the bottom-right region, where the error introduced can be relatively large. Suppose Fig. 12(a) is a diagonal cell. The estimated count is 5 (from B_2) + 4 (from B_3) = 9.

`estimate_count_with_Qf` is similar to `estimate_count` except that it considers a set of points C_q, which satisfy a filter q. In addition to checking the bins and data points with (x_q, y_q), `estimate_count_with_Qf` includes the bins and points in estimation only if there are some nodes in C_q in their bottom-right region.

7.3.3 Generation of Equivalent Query Points

A query point (x_q, y_q) in general has many equivalent query points in the two dimensional space, since a column may be mapped to multiple positions (at the end of Section 6.1) with `equiv`. We now discuss the details of `equiv`. The equivalent query points are generated in two steps. First, we determine the set of column IDs that involve the query point:

$$C = \{c \mid i \in [x_q, y_q], f^{-1}(i) = c\} \quad (1)$$

Second, we compute the intervals that can be constructed by the column IDs:

$$Q = \{(x', y') \mid \forall j \in [x', y']. \exists j = f(c), c \in C\}. \quad (2)$$

To optimize the generation of query points, *i.e.*, Q, we propose to skip generating query points that have empty results. The main idea is illustrated with Fig. 12(b). First we assume that the positions of a column ID are sorted in ascending order offline. We sort C obtained from Equation (1). We scan through the positions of C in parallel. When we obtain a query q_0: (x_0, y_0) that has empty result, we probe the histogram to obtain the grid that contains the data point with the next x_{min}, where $\nexists x.\ x_0 < x < x_{min}$ and (x, y) is a data point. Finally, we skip all positions of columns in C that are smaller than x_{min}.

Compression of mappings between columns and positions. The mappings f and f^{-1} between column IDs and positions can be potentially large. In query point generation, the mappings are scanned, as just discussed. We compress the mappings such that the scan can be efficiently supported in the compressed domain. In essence, instead of storing the equivalent positions of column IDs, we store the difference (delta) between each pair of adjacent positions. We replace repetitive deltas with an ID and their occurrence. For example, the positions (2,3,4,6,8,10,12,14) are compressed to (2,#1×2 ,#2×5), where #1=1 and #2=2. The positions can be trivially regenerated in a scan through the compressed deltas.

Offline `equiv` computation. The next optimization on f and f^{-1} is to precompute `equiv` for all data points and use histograms to summarize the equivalent points, as opposed to computing `equiv` on-the-fly. It is possible because f and f^{-1} depend only on the data graph, not query workloads. In this case, a node is represented by multiple intervals.

8 Experimental Evaluation

In this section, we present an extensive experimental evaluation that verifies the accuracy of our proposed technique and the effectiveness of proposed optimizations. We performed an experimental comparison with XSEED [25] on tree data. Since the implementation of XSKETCH/TREESKETCH [16, 17] is not supported by recent operating systems, we perform an indirect comparison with them.

Experimental settings. We ran our experiments on a server with a Dual 4-core 2.93GHz CPU and 30GB memory running SOLARIS OS (CENTOS release 5.4). Our implementation was written in `Java JDK` 1.6. We implemented equi-depth histograms for grid cells. The default value of the depth of a bin is 10% of the points in a grid cell. We tested equi-width histograms as well but they exhibited a similar performance to equi-depth histograms in our preliminary experiments.

Table 1. XMARK **Characteristics**

XMARK s.f.	0.1	0.4	0.7	1.0	DBLP	TREEBANK
Avg. bindings	3.1k	14.1k	22.9k	35.4k	338k	3k

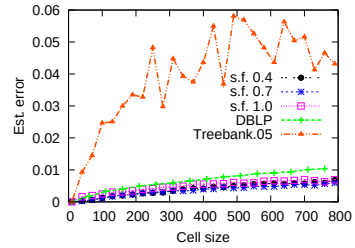
Benchmark datasets. We used XMARK [18], DBLP [15] and TREEBANK [11] to obtain a set of large graphs for evaluation. The scaling factor (s.f.) of XMARK was ranged from 0.4 to 1.0. We set the default s.f. value at 1.0. The DBLP used contains 3.3 million nodes. We note that XSEED supports TREEBANK by extracting up to 5-percentile vertices and hence we followed such an extraction. In addition, since XSEED supports trees only, we ignore the IDREFs in XMARK for the experimental comparisons with XSEED

The definition of metrics. In our experiments, we used the error metrics used in [16] and [25]. The definitions of these metrics can be described as follows. Let n be the number of positive queries, a be the real result count of a query and e be the estimation value. The estimation error is defined to be $(\sum_{i=1}^{n} \frac{|a_i - e_i|}{e_i})/n$. Similar to [16], we applied a sanity bound s to avoid high percentages of low-count queries. We set s to 10-percentile as in [16]. Two alternative error definitions, root mean square deviation (RMSE) and normalized RMSE (NRMSE), were also adopted [25]. RMSE is defined as $\sqrt{(\sum_{i=1}^{n}(e_i - a_i)^2)/n}$ and NRMSE is defined as RMSE/$\bar{a}$, where $\bar{a}$ is $(\sum_{i=1}^{n} a_i)/n$.

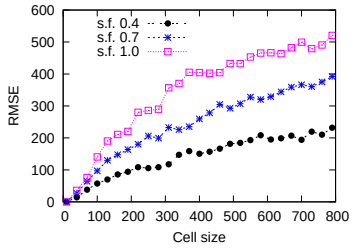
Query workload. We implemented a query generator based on the description in Polyzotis *et al.* [17]. However, since our proposed technique does not involve the synopses of XSKETCH/TREESKETCH, our query generator generates twig queries by sampling the data graph, as opposed to the synopses. On the XMARK, DBLP and TREEBANK datasets, we generated 1,000 positive twig queries, where the query results are larger than 0. The twig queries have one branch on average. The length of the main path ranges from 2 to 5. The number of branches ranges from 1 to 3. This workload is similar to the CP workload reported in [25]. Some characteristics of query workloads are shown in Table 1.

8.1 Experiments on overall performance

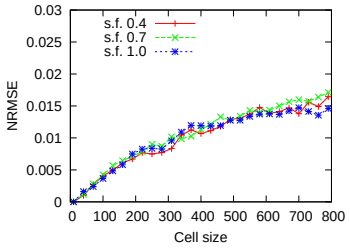
Scalability tests. The estimation errors of the queries on various XMARK graphs are shown in Figs. 13(a)-(e). The x-axis of Figs. 13(a)-(c) is the cell size. From Fig. 13(a), we note that the estimation error increases as the cell size increases (from 0% to 0.7%), as fewer details are captured by larger cells. Our technique is less accurate in DBLP and TREEBANK but the error is still lower than 1.3% and 6%, respectively. Fig. 13(b) shows that RMSE of our implementation increases with the data graph size. However, the normalized RMSE of our implementation is roughly a constant as the data graph size increases, shown in Fig. 13(c). This



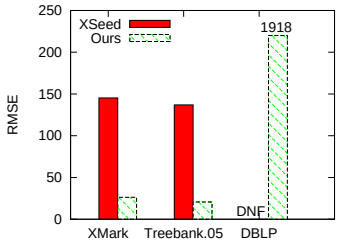
(a) Estimation error of our method



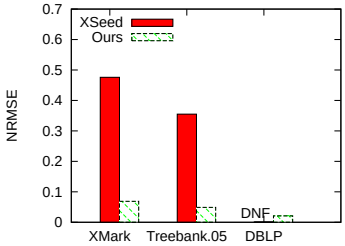
(b) RMSE of our method (XMARK)



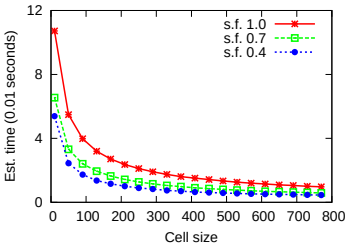
(c) NRMSE of our method (XMARK)



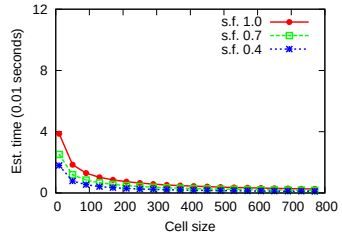
(d) RMSE comparison



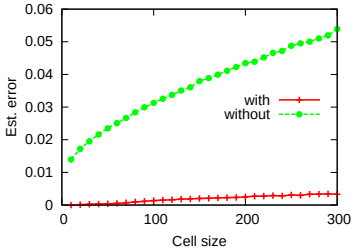
(e) NRMSE comparison



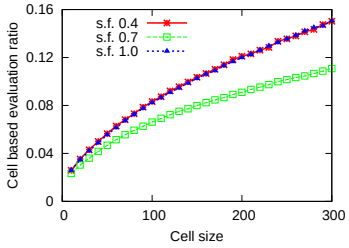
(f) Est. time with eval. on diagonal grid cells



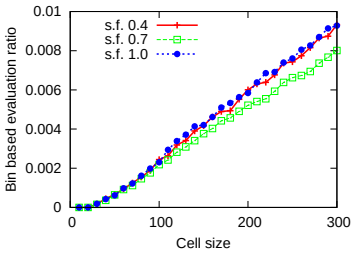
(g) Est. time w/o eval. on diagonal cells



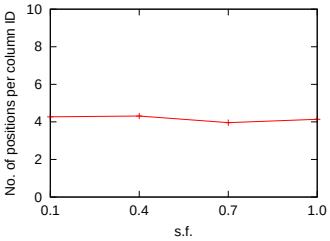
(h) w. vs w/o eval. on diagonal cells



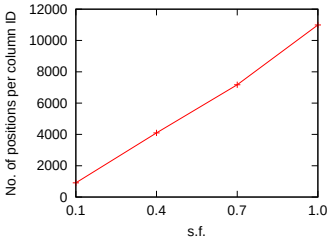
(i) Cell-based eval. to est. ratio



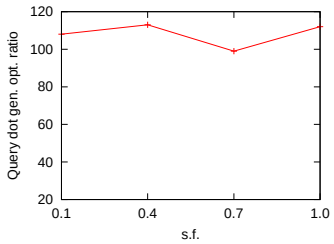
(j) Bin-based eval. to est. ratio



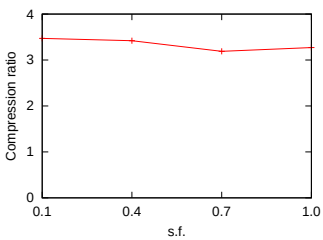
(k) C1P matrix size w. optimization



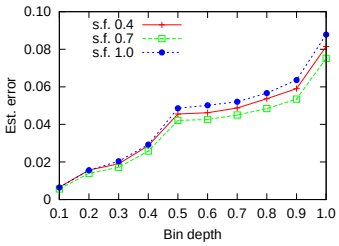
(l) C1P matrix size w/o. optimization



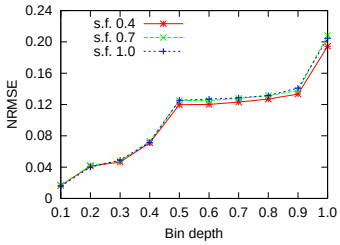
(m) Optimization on query dot generation



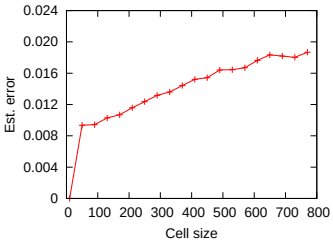
(n) Compression ratio of f and f^{-1}



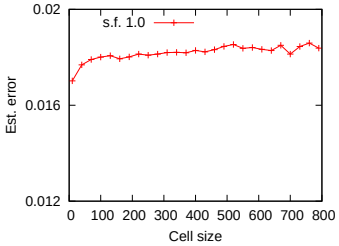
(o) Estimation error as a function of φ



(p) NRMSE as a function of φ



(q) Path queries on XMARK graph



(r) Twig queries on XMARK tree

Figure 13. Performance results on synthetic dataset (XMARK) and real datasets (DBLP and TREEBANK)

is because the absolute result counts are larger in XMARK datasets with larger s.f.'s.

Next, we set the cell size to 800 and use XMARK s.f. 1.0, unless otherwise specified.

We ran the query workload with our implementation and XSEED's. The results given in the two error metrics are presented in Figs. 13(d)-(e). Regarding XMARK, our method's RMSE and NRMSE are more accurate than XSEED's by a factor of 7.1 and 6.9, respectively. On TREEBANK, our method is roughly 6.8 times more accurate than XSEED, in terms of both RMSE and NRMSE. XSEED's evaluation does not finish on DBLP, which is needed for computing XSEED's errors. Our implementation gives a large RMSE on DBLP since the average number of bindings of DBLP is relatively large (Table 1).

Experiment on estimation time. Figs. 13(f)-(g) show the estimation time with and without evaluation on diagonal cells. In any case, the estimation time is less than 0.12 seconds. On average, the estimation with evaluation on diagonal cells is (on average) 2.8 times slower than the estimation without evaluation. However, due to evaluation on diagonal cells, the estimation error reduces from 5% to 0.3% when the cell size is 300 (Fig. 13(h)). In Figs. 13(f) and (g), we did not include the time for `equiv` as it is the same for both methods. For example, the time for `equiv` on XMARK s.f. 1.0 was approximately 0.01 seconds among various cell sizes.

8.2 Experiments on optimizations

Optimizations on diagonal cells. In Fig. 13(h), the error reduction due to evaluation on diagonal cells is large. In this experiment, we analyze how much evaluation is there in the overall estimation. We computed the ratio between the number of points involved in evaluation and those in estimation. Fig. 13(j) shows the ratio is smaller than 1%. The reasons are that most of the points involved in estimation are not in diagonal cells and the evaluation is not performed on the entire diagonal cell, due to x-histograms.

Experiment on evaluation vs estimation ratio. To support the argument that the x-histograms in diagonal cells effectively reduce the amount of evaluation, we tested the ratio between the number of points involved in evaluation and that in estimation when the x-histograms in cells are *not* even used. The result is shown in Fig. 13(i). Without the x-histograms, the ratio reaches 16%, when the cell size is 300. In comparison, Fig. 13(j) shows that the ratio does not reach 1% when x-histograms in cells are used.

Matrix transformations. We tested the size of the C1P matrix obtained from the transformation proposed in Section 6.1 with and without the optimizations in Sections 6.2.1-6.2.2. With the optimizations, Fig. 13(k) shows that the ratio of the increase of the matrix size is approximately 4.1 as s.f. increases. While the size of the matrix increases by a *constant* factor, the C1P matrix is much simpler (recall Figs. 6-7). In contrast, without optimizations, the ratio increases linearly with the s.f. as shown in Fig. 13(l).

Query point generation. We determined the ratio between the number of query points in estimation with and without the query point optimization proposed in Section 7.3.3. The result is shown in Fig. 13(m). It shows that we reduce the number of query points generated by a factor over 100. That is, given a query, there are many equivalent query points that do not contribute the result counts.

Compression of f and f^{-1}. Next, we tested the compression performance presented in Section 7.3.3. The size of f and f^{-1} is important to estimation time as each generation of equivalent query points requires a scan on the compressed f and f^{-1}. The result is plotted in Fig. 13(n). The figure shows that the compression ratio is roughly 3.1 for various s.f.'s.

The depth of the bin in x-histograms. In previous experiment, we set the depth to be 10% of the number of dots of a cell. To show the effect of the depth of x-histograms on the estimation accuracy, we performed an experiment by varying the bin's depth from 10% to 100%. The cell size is 800. The results are shown in Figs. 13(o)-(p). As expected, when the depth increases, the estimation error and NRMSE gradually increase. Due to space constraints, we skip the results on RMSE as we observed a similar trend.

8.3 Indirect comparison with XSketch and TreeSketch

Although the implementation of XSKETCH/TREESKETCH [16, 17] has been available, it was developed on a legacy `gcc`, which is no longer supported. Some `gcc` libraries used have no longer been available. Therefore, we could only compare the numbers reported from [16, 17]. We compared the estimation error of XSKETCH/TREESKETCH and ours on XMARK dataset s.f. 1.0.

As discussed, XSKETCH supports path queries only for cyclic graphs. XSKETCH generates queries based on the popularity of tags in their synopses, which is absent in our method. Hence, we generated path queries based on the popularity of tags in data graphs. As shown in Fig. 13(q), our estimation error has not reached 2% when the cell size is smaller than 800. When the cell size is smaller than 200, our estimation error has been controlled under 1%. In comparison, the estimation error of XSKETCH reported in [16] is well-controlled under 10%.

Next, we compared the results reported from TREESKETCH [17]. TREESKETCH estimates the selectivity of twig queries but on acyclic graphs only. As in

TREESKETCH, we did not consider IDREF in XMARK. The twig queries were generated as described in the beginning of this section. Fig. 13(r) shows that our technique controls the estimation error around 1.6%. In comparison, TREESKETCH controls the estimation errors of twig queries on XMARK tree under 5%.

9 Conclusion

In this paper, we propose a histogram-based selectivity estimation of twig queries on cyclic graphs. To the best of our knowledge, previous works only focus on either twig queries or cyclic graphs but not both. Specifically, we propose a new matrix representation of cyclic graphs by our prime labeling scheme. Next, we derive a heuristic transformation of the matrix to a C1P matrix for summarization. As a result, a data node is represented by an interval and subsequently a two-dimensional data point. A query is then represented by multiple points in runtime. Two-dimensional histograms are used to summarize data points and auxiliary structures are introduced to tackle skewed data points. We present a selectivity estimation algorithm on the histograms. Our experiments with XMARK and DBLP show that the estimation error is well-controlled under 1.3%, which is more accurate than XSKETCH/TREESKETCH and XSEED. On TREEBANK, we produce RMSE and NRMSE 6.8 times smaller than XSEED's.

As for future works, (i) we are incorporating this technique with queries with filters on data values; and (ii) we are investigating graph partitioning to optimize the computation of the binary matrix, which is currently maintained in the main memory.

References

[1] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the selectivity of xml path expressions for internet scale applications. In *VLDB*, pages 591–600, 2001.

[2] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, pages 253–262, 1989.

[3] V. Batagelj and A. Mrvar. Pajek datasets. http://vlado.fmf.uni-lj.si/pub/networks/data/.

[4] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng, and D. Srivastava. Counting twig matches in a tree. In *ICDE*, pages 595–604, 2001.

[5] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[6] D. K. Fisher and S. Maneth. Structural selectivity estimation for xml documents. In *ICDE*, pages 626–635, 2007.

[7] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Statix: making xml count. In *SIGMOD*, pages 181–191, 2002.

[8] R. Goldman and J. Widom. Approximate dataguides. In *In Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, volume 97, pages 436–445, 1999.

[9] W.-L. Hsu. A simple test for the consecutive ones property. *J. Algorithms*, 43(1):1–16, 2002.

[10] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, page 129, 2002.

[11] Language and Information in Computation at Penn. Penn treebank project. Available at http://www.cis.upenn.edu/ treebank/.

[12] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Parr. Xpathlearner: an on-line self-tuning markov histogram for xml path selectivity estimation. In *VLDB*, pages 442–453, 2002.

[13] Z. Lin, B. He, and B. Choi. A quantitative summary of xml structures. In *ER*, pages 228–240, 2006.

[14] J. McHugh and J. Widom. Query optimization for xml. In *VLDB*, pages 315–326, 1999.

[15] G. Miklau. UW XML repository. Available at http://www.cs.washington.edu/research/xmldatasets/.

[16] N. Polyzotis and M. Garofalakis. Xsketch synopses for xml data graphs. *ACM Trans. Database Syst.*, 31(3):1014–1063, 2006.

[17] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Approximate xml query answers. In *SIGMOD*, pages 263–274, 2004.

[18] A. Schmidt, F. Waas, M. Kersten, M. J. Carey I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *VLDB*, pages 974–985, 2002.

[19] J. Tan and L. Zhang. The consecutive ones submatrix problem for sparse matrices. *Algorithmica*, 48(3):287–299, 2007.

[20] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[21] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom histogram: Path selectivity estimation for xml data with updates. In *VLDB*, pages 240–251, 2004.

[22] G. Wu, K. Zhang, C. Liu, and J.-Z. Li. Adapting prime number labeling scheme for directed acyclic graphs. In *DASFAA*, pages 787–796, 2006.

[23] X. Wu, M. L. Lee, and W. Hsu. A prime number labeling scheme for dynamic ordered xml trees. In *ICDE*, page 66, 2004.

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

[25] N. Zhang, M. Ozsu, A. Aboulnaga, and I. Ilyas. Xseed: Accurate and fast cardinality estimation for xpath queries. In *ICDE*, pages 168–197, 2006.