Path-centric Cardinality Estimation for Subgraph Matching

Zhengdong Wang
Shanghai Jiao Tong University
lnwzd2009@sjtu.edu.cn

Qiang Yin*
Shanghai Jiao Tong University
q.yin@sjtu.edu.cn

Longbin Lai
Alibaba Group
longbin.lailb@alibaba-inc.com

ABSTRACT

This paper presents PathCE, a path-centric cardinality estimation framework for subgraph matching. PathCE improves estimation accuracy by utilizing statistics from short graph queries. At its core is a novel data structure called the *path-centric summary graph* (PSG), which captures short path query statistics from a data graph G and represents them in a new graph $\mathcal{G}$. Given a graph query Q and a PSG graph $\mathcal{G}$ for G, PathCE decomposes Q into a simpler query $\mathcal{Q}$, where each edge in $\mathcal{Q}$ corresponds to a sub-path query in Q with statistics included in $\mathcal{G}$. PathCE estimates the cardinality using $\mathcal{Q}$ and $\mathcal{G}$, requiring significantly fewer estimation iterations while ensuring that the estimate remains an upper bound on the true cardinality of $Q(G)$. It also includes PSGBuilder, a parallelly scalable algorithm that constructs PSG's for any given graph in linear time, efficiently scaling with the number of processors. Empirical results on real-world and synthetic datasets show that PathCE outperforms state-of-the-art baselines in accuracy, estimation latency, and summary construction efficiency.

PVLDB Reference Format:
Zhengdong Wang, Qiang Yin, and Longbin Lai. Path-centric Cardinality Estimation for Subgraph Matching. PVLDB, 18(9): 3063 - 3076, 2025.
doi:10.14778/3746405.3746428

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/wzzzzd/pathce.

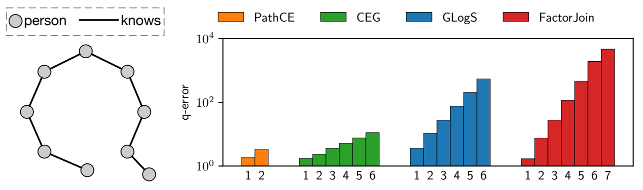
1 INTRODUCTION

Given a graph query Q and a data graph G, *subgraph matching* is to find all matches of Q in G. It serves as a fundamental component for modern graph query languages such as SPARQL [16], Cypher [13], and the most recently released ISO/IEC standard GQL [11]. Cardinality estimation for subgraph matching is to estimate the number of matches of Q in G, denoted as $|Q(G)|$, without explicitly computing them. It plays a pivotal role in cost-based query optimizers, which rely on it to estimate the cost of candidate query plans [24].

While cardinality estimation has been extensively studied in relational databases, techniques specifically tailored for graph data remain relatively underdeveloped [6, 9, 34]. Although graph queries can be expressed as joins, directly applying relational methods to graph data poses significant challenges. Relational DBMSs typically rely on table-level statistics, which correspond to edge-level statistics in graphs. However, such statistics are often too coarse to accurately capture the complex structural correlations inherent in graph queries, leading to substantial estimation errors [6, 23]. In addition, collecting detailed statistics such as multi-way join counts or maximal degrees across multiple tables is often impractical. In contrast, graph DBMSs efficiently support neighborhood access, enabling the collection of statistics for small graph patterns such as paths, which capture richer structural correlations beyond edge-level statistics alone [34]. This opens up new opportunities for developing graph-specific cardinality estimation techniques.

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746428



(a) Query Q **(b)** Estimation accuracy of subqueries across iterations

Figure 1: Cardinality estimation using graph query Q on LDBC. For each estimator, (b) shows the q-error incurred during each iteration.

Summary-based cardinality estimators are often favored in practice because they rely solely on the data rather than on specific queries [6, 15, 34]. In subgraph matching, these estimators typically precompute statistics from small queries of G and employ an iterative estimation approach to estimate $|Q(G)|$. Each iteration generates estimates for a subquery of Q, referred to as an *estimation iteration*. However, existing summary-based estimators designed for graph or relational data, such as CEG [6], GLogS [23], and FactorJoin [47], face significant accuracy challenges when handling complex graph queries, which are common in real-life workloads [2, 30], as illustrated by the example below.

Example 1: Consider cardinality estimation using a graph query Q on the LDBC dataset [2], which is developed by the Linked Data Benchmark Council to benchmark graph DBMSs. As shown in Fig. 1(a), Q has 9 vertices labeled person, and 8 edges labeled knows.

(1) The number of estimation iterations varies among estimators. For example, FactorJoin generates estimates for the subquery Q_i in the i-th iteration, where Q_i is a subquery of Q consisting of i+2 vertices labeled person and i+1 edges labeled knows. Thus, FactorJoin takes 7 iterations to produce the estimate of Q, while PathCE, CEG and GLogS require 2, 6 and 6 iterations, respectively.

(2) For each estimator, we break down its estimation process and analyze the subquery estimates across different iterations. We find that the number of iterations strongly affects estimation accuracy. We employ the classic accuracy metric q-error [31] to measure deviations of estimated cardinality from true cardinality. Figure 1(b) shows how q-error evolves across iterations for each estimator. □

Observation and challenges. We observe that q-error increases progressively in later iterations due to *error accumulation*, as later subquery estimates rely on earlier (potentially inaccurate) results. This effect is particularly evident in estimators requiring many iterations. For example, in FactorJoin, the q-error grows from 1.69 to 4657.23 across its 7 iterations for the query in Fig. 1(a). In contrast, PathCE needs 2 iterations, with q-error rising from 1.90 to 3.39. These findings suggest that *minimizing the number of estimation iterations is crucial for reducing cumulative estimation error and improving accuracy*. However, this also raises several challenges.

- (**C1**) Is there an effective way to reduce the number of estimation iterations and improve accuracy? Prior research indicates that utilizing statistics from small graph queries, *e.g.,* 2-path queries and triangle queries, can improve accuracy [6, 23].
- (**C2**) Is there a scalable way to construct sufficient statistics while substantially reducing estimation iterations? It is clear that the estimation accuracy can be improved using statistics of more generic queries, but concerns about the efficiency of computing these statistics remain [23, 34, 39]. For example, although GLogS [23] can maintain any query statistics theoretically, the cost is prohibitive. Thus, GLogS still relies on small queries, requiring extensive estimation iterations for processing the aforementioned graph query.
- (**C3**) Another critical argument from previous studies [4, 10, 47] highlights the value of upper bound estimates in query optimization, which has also been demonstrated in our experiments (Sec. 8), where FactorJoin, despite having lower estimation accuracy, often surpasses CEG and GLogS in generating more effective execution plans. This leads us to consider: Can we ensure estimators constantly provide an upper bound on the true cardinality?

In this paper, we propose PathCE, a *path-centric* summary-based cardinality estimation framework, to answer these questions.

A path-centric approach. Given a graph query Q, PathCE first decomposes Q into a collection of path queries, forming a new query $\mathcal{Q}$, where each edge of $\mathcal{Q}$ represents a sub-path in Q. Estimation is conducted using $\mathcal{Q}$ and the path query statistics, which addresses **(C1)** by significantly reducing the number of iterations due to $\mathcal{Q}$'s simpler structure, thereby enhancing estimation accuracy. To tackle **(C2)**, PathCE introduces a scalable algorithm to efficiently construct path query statistics. This naturally balances accuracy and efficiency: offering higher accuracy compared to edge-only statistics [47], while being more efficient than using statistics of generic graph queries [23]. Finally, inspired by recent advancements in upper bound estimation [1, 4], PathCE addresses **(C3)** by integrating both count and maximal-degree information into the path query statistics to ensure an upper bound estimate.

Below, we summarize our contributions and paper organization.

Contribution & organization. We develop a path-centric cardinality estimation framework PathCE to improve estimation accuracy, while maintaining the efficiency of computing the statistics.

(1) A novel summary structure (Sec. 3). We introduce *path-centric summary graphs* (PSG) as a novel data structure for maintaining the statistics of path queries. A PSG for a graph G is itself a graph $\mathcal{G}$, where (i) each vertex in $\mathcal{G}$ represents a subset of vertices in G with an identical label, and (ii) each edge in $\mathcal{G}$ represents a path query between the vertex subsets at its endpoints. The size of $\mathcal{G}$ depends only on types of vertices/edges in G rather than the size of G, making PSG a scalable summary structure for large graphs.

(2) A path-centric framework (Sec. 4). Given a query Q and a summary graph $\mathcal{G}$, PathCE estimates $|Q(G)|$ in two steps. PathCE first generates an estimation scheme $(\mathcal{Q}, \mathcal{S})$ for Q, consisting of a new query $\mathcal{Q}$ and an estimation order $\mathcal{S}$. The query $\mathcal{Q}$ has a simpler structure than Q, and $\mathcal{S}$ is a vertex sequence of $\mathcal{Q}$. PathCE next conducts estimation using $\mathcal{Q}$ and $\mathcal{G}$. This significantly reduces estimation iterations since $\mathcal{Q}$ is much simpler. In addition, the estimation is guided by $\mathcal{S}$, avoiding the exhaustive search for better estimates.

(3) Cardinality estimation with PSG (Sec. 5 & Sec. 6). We develop a systematic approach to generate an effective estimation scheme $(\mathcal{Q}, \mathcal{S})$ for a given query Q (Sec. 5). To achieve upper bound estimation, we introduce a *graph-based* approach that guarantees every estimation iteration produces an upper bound estimate (Sec. 6).

(4) Scalable PSG *construction* (Sec. 7). We develop a *parallelly scalable* [22] algorithm PSGBuilder to construct PSG's. The algorithm PSGBuilder runs in linear time *w.r.t.* the size of G and guarantees to reduce the running time when working with more processors.

(5) Experimental study (Sec. 8). We prototype PathCE and compare it with state-of-the-art summary-based, sampling-based and ML-based baselines. We find the following. **(a)** Among the summary-based estimators, PathCE produces the most accurate estimates for cyclic queries on both real-world and synthetic datasets. For acyclic queries, PathCE achieves comparable accuracy to CEG, *e.g.,*PathCE achieves an average q-error below 20 on IMDB. The performance of PathCE is quite stable: it always produces upper bound estimates, its q-error grows slowly with query size, and remains consistent *w.r.t.* query topology (Exp-1). **(b)** PathCE is one of the fastest estimators in terms of estimation latency. It completes all test queries within 0.1 seconds, with low variance (Exp-2). **(c)** The PSGBuilder algorithm of PathCE is highly efficient. It constructs a PSG graph for the LDBC dataset in 734 seconds and scales well with both graph size and number of thread (Exp-3 & Exp-4). **(d)** PathCE consistently beats all baselines in end-to-end performance, with lower latencies in both query planning and query execution (Exp-5).

Related work. We categorize the related work as follows.

Summary-based estimators. Summary-based estimators have been widely adopted by both relational and graph databases.

(1) Summary-based estimators for *relational databases* utilize summaries, also referred to as *catalogs* or *sketches* in some literatures, such as histograms [20], HyperLogLog [12], and CountMin [8], in combination with statistical assumptions on underlying data to make estimates, following the paradigm pioneered by the System R optimizer [37]. Closer to this work are BoundSketch [4], Simplicity [18], FactorJoin [47], and SafeBound [10], which utilize maximal-degree statistics to produce upper bound estimates.

PathCE differs from prior work in the following. (a) Unlike the System R style estimator that relies on statistics assumptions, PathCE relies on no statistics assumption and is essentially derived from the degree-constraint bound [1], which guarantees upper bound estimates. using maximal-degree statistics. (b) PathCE collects statistics of short path queries, in contrast, most prior work relies on single edge (or base table), *e.g.,* [4, 10, 18, 47]. (c) PathCE produces upper bound estimates by a graph-base approach, as opposed to the bound formula generator of BoundSketch [4], or probabilistic graphical models adopted by FactorJoin [47].

(2) Summary-based estimators have also been developed for *graph databases* [6, 9, 23, 33, 39, 46]. C-SET [33] introduces the *characteristic set*, which represents the distinct outgoing labels for each vertex in an RDF graph. SumRDF [39] proposes a *typed summary* graph that groups vertices with similar labels and incident edge distributions. GLogS [23] presents a graph-structured summary retaining small subgraph counts, while CEG [6] uses a *Markov table* to store short path query cardinalities. A-LHD [46] collects single-edge statistics and adopts a label probability propagation model to improve accuracy. Color [9] is a recent summary-based estimator that exploits graph coloring techniques in cardinality estimation. Like System R style estimators, they face similar challenges due to reliance on assumptions.

PathCE differs from prior work in several ways. (a) PathCE is assumption-free and guarantees upper bound estimates by leveraging maximal-degree statistics. (b) PathCE uses short path queries, unlike C-SET, SumRDF and Color, which only consider single edge queries. (c) While both CEG and GLogS collect statistics on 2-path queries, they do not consider maximal-degree information. (d) CEG and GLogS also utilize triangle counts to improve accuracy, but collecting these counts is cost-prohibitive, requiring techniques like graph sparsification [35, 41]. In contrast, PathCE constructs summaries efficiently without sacrificing estimation accuracy. (e) While PathCE employs the heuristic GBSA [47] for vertex partitioning, Color utilizes graph coloring to achieve the same goal.

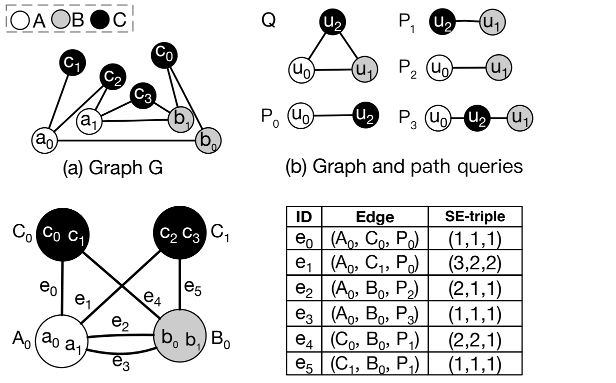
Sampling-based and ML-based estimators. Sampling-based estimators are alternatives to summary-based estimators, and they have played an important role in query optimization for decades [26, 43, 51]. Generally, they are able to handle queries with correlations and non-uniformity better than summary-based estimators, but with problems of high estimation latency and poor scalability in terms of the query size [10]. Recently, ML-based *query-driven* [17, 21, 32, 50] and *data-driven* [36, 44, 48, 52] estimators have been actively developed and they have achieved satisfactory accuracy in open benchmarks [14, 40]. Latest work on ML-based estimators propose pre-trained models and fine-tuning [7, 28, 49] to avoid costly model retraining over changing data and workloads. As a summary-based estimator, PathCE generally offers (a) better estimation latency compared to sampling-based one, (b) scalable summary construction without extensive model training, and (c) interpretable results, unlike the black-box nature of ML-based estimators [45].

2 PRELIMINARIES

We begin with basic notations. Let Γ be a countable set of labels.

Graphs and queries. We follow the typical setting of subgraph matching [34] and focus on *undirected* labeled graphs. Our approach can be easily extended to directed graphs.

- A *graph* $G = (V, E, L)$ is a triple, where V is a finite vertex set, $E \subseteq V \times V \times \Gamma$ is a finite set of labeled edges, and L is a *labeling function* that maps each vertex $v \in V$ (and each edge $e \in E$) to a label from Γ. For an edge $e = (v_1, v_2, \ell)$, we have $\ell = L(e)$, with v_1 and v_2 referred to as the endpoints of e. For a vertex v, let $\mathsf{Nbr}(v)$ denote the neighbors of v in G.


○A ●B ●C
c_1 c_2 c_3 c_0 a_1 b_1 a_0 b_0
(a) Graph G
Q u_2 u_0 u_1
P_1 u_2 u_1
P_2 u_0 u_1
P_0 u_0 u_2
P_3 u_0 u_2 u_1
(b) Graph and path queries
C_0 c_0 c_1 c_2 c_3 C_1
e_0 e_1 e_4 e_5 e_2 e_3
A_0 a_0 a_1 b_0 b_1 B_0
(c) A summary graph

ID	Edge	SE-triple
e_0	(A_0, C_0, P_0)	(1,1,1)
e_1	(A_0, C_1, P_0)	(3,2,2)
e_2	(A_0, B_0, P_2)	(2,1,1)
e_3	(A_0, B_0, P_3)	(1,1,1)
e_4	(C_0, B_0, P_1)	(2,2,1)
e_5	(C_1, B_0, P_1)	(1,1,1)

(d) Summary edges and SE-triples

Figure 2: A data graph G, example graph and path queries, along with a path-centric summary graph $\mathcal{G}$ of G.

- A graph query Q is often represented by its query graph $Q = (V_Q, E_Q, L_Q)$, where V_Q, E_Q and L_Q are its vertex set, edge set and labeling function. A query Q' is a *subquery* of Q if its query graph is a subgraph of Q's query graph. Q is *cyclic* if its query graph contains a cycle as a subgraph; otherwise Q is *cyclic*.
- A *path query* is a special graph query whose query graph forms a labeled path. We use the symbol P for path queries. A path query with k edges is called a *k-path query* and can be represented as $u_0 e_1 u_1 \ldots u_{k-1} e_k u_k$, where each edge e_i connects endpoints u_{i-1} and u_i for $i = 1, \ldots, k$. We refer to u_0 and u_k as the *head* and *tail* of P, denoted by h_P and t_P, respectively.

Subgraph matching. Given a graph query $Q = (V_Q, E_Q, L_Q)$ and a data graph $G = (V, E, L)$, a *subgraph homomorphic match* of Q in G is a mapping $\rho : V_Q \longrightarrow V$ such that (i) for every $u \in V_Q$, $L_Q(u) = L(\rho(u))$; and (ii) for every edge (u_1, u_2, ℓ) in E_Q, there exists an edge $(\rho(u_1), \rho(u_2), \ell')$ in E with $\ell = \ell'$. For simplicity, we will refer to ρ as a *match* of Q in G. Note that there may be multiple matches of Q in G. We denote by $Q(G)$ the set of all matches of Q in G. Similarly, $P(G)$ is defined for a path query P. Given a query graph Q and a data graph G, the *subgraph matching problem* is to compute $Q(G)$.

Cardinality estimation problem. Given a query graph Q and a data graph G, the cardinality of $Q(G)$, denoted by $|Q(G)|$, refers to the number of matches of Q in G. The *cardinality estimation problem for subgraph matching* is to estimate $|Q(G)|$ without explicitly computing the set $Q(G)$. Such estimation is typically performed by leveraging statistics from smaller subqueries [6, 23, 47]. As we will see shortly, we collect statistics of short path queries only and utilize them to support cardinality estimation for arbitrary graph queries.

Example 2: Consider a data graph G as shown in Fig. 2(a), with three vertex labels A, B, and C, represented by different colors. The edge labels are assumed to be identical and thus are omitted.

(1) We have $|Q(G)| = 1$ for the graph query Q shown in Fig. 2(b), as $Q(G)$ contains exactly one match ρ_0, *i.e.,* $Q(G) = \{\rho_0\}$, where $\rho_0 = \{(u_0, a_1), (u_1, b_1), (u_2, c_3)\}$.

Table 1: Notations

$G = (V, E, L)$, Π_V, $\mathbb{P}$	graph, vertex partition, path query set
$\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$, $\mathcal{Q}$	PSG and PSG query
$Q = (V_Q, E_Q, L_Q)$, $Q(G)$	graph query and its match set on G
$Q[u_1, u_2]$, $Q[u]$	graph query with designated endpoints

(2) For the path query P_1 shown in Fig. 2(b), we have $|P_1(G)| = 3$, with ρ_1, ρ_2, and ρ_3 as its matches, where $\rho_1 = \{(u_2, c_0), (u_1, b_0)\}$, $\rho_2 = \{(u_2, c_0), (u_1, b_1)\}$, and $\rho_3 = \{(u_2, c_3), (u_1, b_1)\}$. Furthermore, the head and tail vertices of P_1 are u_2 and u_1, respectively. □

The notations of this paper are summarized in Table 1.

3 PATH-CENTRIC SUMMARY GRAPHS

In this section, we introduce *path-centric summary graphs* (PSG), a data structure that maintains statistics of path queries as a graph.

We first introduce a general framework for defining the statistics of a graph query Q. Note that these statistics are always defined with respect to an underlying data graph. To simplify our presentation, we assume a fixed data graph $G = (V, E, L)$ throughout this section.

Definition 3.1: Let Q be a graph query, and let u_1 and u_2 be two designated nodes in Q. Let V_1 and V_2 be two subsets of V. The summary of $Q[u_1, u_2]$ with respect to the vertex subset pair (V_1, V_2) is defined as a triple (c, d_1, d_2), where:

- $c = \left|\{\rho \in Q(G) \mid \rho(u_1) \in V_1,\ \rho(u_2) \in V_2\}\right|$;
- $d_1 = \max_{v_1 \in V_1}\left|\{\rho \in Q(G) \mid \rho(u_1) = v_1,\ \rho(u_2) \in V_2\}\right|$;
- $d_2 = \max_{v_2 \in V_2}\left|\{\rho \in Q(G) \mid \rho(u_1) \in V_1,\ \rho(u_2) = v_2\}\right|$. □

Intuitively, c denotes the number of matches in $Q(G)$ where u_1 and u_2 are mapped to vertices in V_1 and V_2, respectively. In these matches, d_1 (resp. d_2) represents the maximum number of occurrences of any vertex in V_1 (resp. V_2) as the match for u_1 (resp. u_2). We refer to d_1 and d_2 as the *maximal-degree statistics* of $Q[u_1, u_2]$.

Similarly, we define the summary for a single designated node.

Definition 3.2: Let Q be a graph query, u_1 be a designated node in Q, and $V_1 \subseteq V$ be a vertex subset. The summary of $Q[u_1]$ with respect to V_1 is also a triple (c, d_1, d_2), where:

- $c = \left|\{\rho \in Q(G) \mid \rho(u_1) \in V_1\}\right|$;
- $d_1 = \max_{v_1 \in V_1}\left|\{\rho \in Q(G) \mid \rho(u_1) = v_1\}\right|$;
- $d_2 = \infty$, *i.e.*, d_2 is undefined as Q has only one designated node. □

Intuitively, we treat $Q[u_1, u_2]$ as an *edge-like* query by designating u_1 and u_2 as its two endpoints. Similarly, $Q[u_1]$ can be viewed as a *vertex-like* query, with u_1 as its sole endpoint.

Example 3: Continue from Example 2 and consider the path query P_1 depicted in Fig. 2(b). Let $B_0 = \{b_0, b_1\}$ and $C_0 = \{c_0, c_1\}$ be two subsets of V. We can observe the following.

(1) The summary of $P_1[u_2, u_1]$ *w.r.t.* (C_0, B_0) is $(c, d_1, d_2) = (2, 2, 1)$. Indeed, among the three matches in $P_1(G)$, only ρ_1 and ρ_2 satisfy the additional constraint that u_2 is matched to a vertex in C_0 and u_1 is matched to a vertex in B_0 (see the definitions of ρ_1 and ρ_2 in Example 2); $d_1 = 2$ since both ρ_1 and ρ_2 map u_2 to the vertex c_0, which has the maximal number of occurrences; similarly, $d_2 = 1$ since both b_0 and b_1 occur once as the match of u_1 in these matches.

(2) The summary of $P_1[u_1]$ *w.r.t.* B_0 is $(3, 2, \infty)$. This is because all three matches of $P_1(G)$, *i.e.*, ρ_1, ρ_2, and ρ_3, map u_1 to a vertex in B_0, and the vertex b_1 in B_0 appears twice, which has the maximal number of occurrences among these matches. □

To construct a PSG for a data graph G, we utilize a *summary vertex partition* Π_V to partition V into a collection of disjoint subsets. The vertices in each subset have an identical label and each subset is referred to as a *summary vertex*. Specifically, let V^ℓ be the set of vertices in V with label ℓ. Given a predefined number M, Π_V divides each V^ℓ into M summary vertices $V_1^\ell, \ldots, V_M^\ell$. Thus, Π_V can be represented as $\Pi_V = \{V_i^\ell \mid \ell \text{ is a vertex label in } G,\ i = 1, \ldots, M\}$.

Collecting statistics for all possible graph queries, even just for path queries, is computationally infeasible [6, 23]. Thus we focus on a small set $\mathbb{P}$ of k-path queries (with small k), representing their statistics with a graph-based data structure called the *path-centric summary graph* (PSG). These path query statistics will then be used to support cardinality estimation for arbitrary graph queries.

Definition 3.3: Given a summary vertex partition Π_V of G, and a set $\mathbb{P}$ of path queries, a *path-centric summary graph* (PSG) of G *w.r.t.* Π_V and $\mathbb{P}$ is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ satisfying the followings.

(1) The *vertex set* $\mathcal{V}$ consists of all the summary vertices defined by Π_V. Observe that all vertices in a summary vertex V' share an identical label ℓ. Thus, we let $\mathcal{L}(V') = \ell$.

(2). The *edge set* $\mathcal{E}$ consists of all edges $e = (V_1, V_2, P)$ such that (i) P is a path query from the set $\mathbb{P}$, which serves as the label of e, and (ii) the labels $\mathcal{L}(V_1)$ and $\mathcal{L}(V_2)$ match the labels of h_P and t_P, respectively. Each edge in $\mathcal{E}$ is referred to as a *summary edge*.

(3) Each summary edge $e = (V_1, V_2, P)$ carries a triple (c, d_1, d_2) as its property, which is precisely the summary of the path query $P[h_P, t_P]$ with respect to (V_1, V_2), where the head vertex h_P and tail vertex t_P are taken as the designated nodes of P.

We refer to the property triple associated with a summary edge as an SE-*triple*. The collection of all SE-triples associated with summary edges of the form (V_1, V_2, P) in $\mathcal{G}$ is called the *statistics of* P. □

In a nutshell, a PSG adopts a path-centric approach, representing the statistics of *path queries* within a graph structure. The size of a PSG is independent of the size of the data graph G and is bounded by $O(NM^2)$, where N is the number of path queries in $\mathbb{P}$, and M is the number of summary vertices per label. This property makes PSG a scalable summary data structure for large graphs.

Example 4: Continue from Example 2 and Example 3, and let $\Pi_V = \{A_0, B_0, C_0, C_1\}$ be a vertex partition with 4 summary vertices, where $A_0 = \{a_0, a_1\}$, $B_0 = \{b_0, b_1\}$, $C_0 = \{c_0, c_1\}$, and $C_1 = \{c_2, c_3\}$ (see Fig. 2(c)). Let $\mathbb{P} = \{P_0, P_1, P_2, P_3\}$ denote the set of all path queries shown in Fig. 2(b). The summary graph $\mathcal{G}$ with respect to Π_V and $\mathbb{P}$ is depicted in Fig. 2(c), and Fig 2(d) lists its 5 summary edges, along with their labels and SE-triples. Observe the following.

(1) The SE-triple of the summary edge e_4 is $(2, 2, 1)$, as it corresponds to the summary of $P_1[u_2, u_1]$ with respect to the summary vertex pair (C_0, B_0). Similarly, one can verify the other SE-triples.

(2) There are two path queries in $\mathbb{P}$, namely P_2 and P_3, that connect a vertex labeled A to a vertex labeled B. Consequently, two summary edges, e_2 and e_3, exist in $\mathcal{G}$ between the summary vertices A_0 and B_0. The SE-triple associated with e_2 represents the sum-

Algorithm 1: PathCE: Path-centric estimation workflow

```
Input: A graph query $Q$ and a PSG graph $\mathcal{G}$ of data graph $G$.
Output: An upper bound estimaiton of $|Q(G)|$.
1 $(\mathbf{Q}, \mathcal{S}) \leftarrow$ GenScheme($\mathbf{Q}, \mathcal{G}$);    // $\mathcal{S} = [u_1, \ldots, u_k]$
2 for $i = 1, \ldots, k-1$ do
3     $Q_i[\mathsf{Nbr}(u_i)] \leftarrow$ ExtraSubQ($\mathbf{Q}, u_i$);    //Extract a subquery
4     $\mathcal{G} \leftarrow \mathcal{G} \cup$ EstTriple($\mathcal{G}, Q_i[\mathsf{Nbr}(u_i)]$); //Estimate SE-triple
5     $\mathbf{Q} \leftarrow$ ReduceQuery($\mathbf{Q}, Q_i[\mathsf{Nbr}(u_i)]$); //Reduce PSG query
6 return AggCnt($\mathcal{G}, \mathbf{Q}, u_k$);
```

mary of $P_2[u_0, u_1]$ with respect to (A_0, B_0), while the SE-triple of e_3 represents the summary of $P_3[u_0, u_1]$ (see more in Fig. 2(d)). □

We often need to represent the statistics of longer path queries beyond $\mathbb{P}$, as well as general graph queries. Fortunately, our framework allows all such queries to be represented in PSG.

(1) For $Q[u_1, u_2]$ with u_1 and u_2 as its designated endpoints, we add a new summary edge $(V_1, V_2, Q[u_1, u_2])$ in $\mathcal{G}$, for every summary vertex pair (V_1, V_2) with $\mathcal{L}(V_1) = L(u_1)$ and $\mathcal{L}(V_2) = L(u_2)$. Its SE-triple is defined as the summary of $Q[u_1, u_2]$ *w.r.t.* (V_1, V_2).

(2) For $Q[u_1]$ with u_1 as its sole designated endpoint, we first introduce a *dummy summary vertex* U to the summary vertex set $\mathcal{V}$, if it does not already exist. Then, for each summary vertex V_1 with $\mathcal{L}(V_1) = L(u_1)$, we add a new summary edge $(V_1, U, Q[u_1])$, whose SE-triple is the summary of $Q[u_1]$ with respect to V_1.

Similarly, the set of SE-triples associated with these edges constitutes the statistics of $Q[u_1, u_2]$ and $Q[u_1]$, respectively.

Cardinality estimation with PSG**'s**. Given a query Q and a path-centric summary graph $\mathcal{G}$ of G, our goal is to estimate $|Q(G)|$ by making use of the path query statistics included in $\mathcal{G}$. Recall we only include the statistics of a small set of k-path queries when constructing $\mathcal{G}$. If $\mathcal{G}$ already contains statistics for Q, *e.g.,* if Q is a small path query in $\mathbb{P}$, we aggregate the statistics to estimate $|Q(G)|$. For longer paths or generic graph queries like Q, we estimate their statistics using the shorter path statistics contained in $\mathcal{G}$. Once we have the statistics for Q, we aggregate them to estimate $|Q(G)|$.

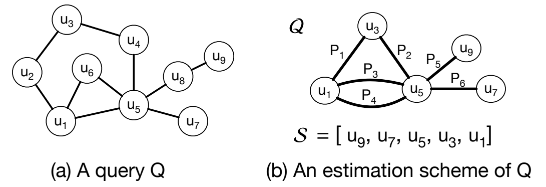
Remark. Our framework restricts the number of designated endpoints in a graph query Q to at most two (see Definitions 3.1, 3.2, and 3.3). While this design can be extended to support more endpoints, we make this choice deliberately for the following reasons.

(1) Limiting to two endpoints allows all summaries to be represented using a simple graph structure. Supporting more endpoints would require hypergraphs, significantly increasing design complexity.

(2) This restriction also reduces both storage and computation costs. A path query with two endpoints requires $O(M^2)$ space in PSG. Allowing three endpoints would increase this to $O(M^3)$. Moreover, more designated endpoints requires additional maximal-degree statistics, further increasing construction overhead (see Sec. 7). □

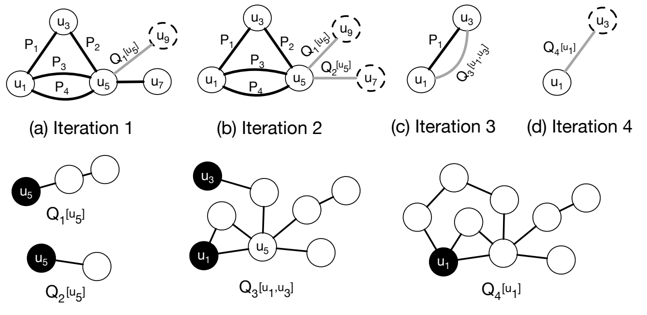
4 PATH-CENTRIC ESTIMATION FRAMEWORK

We present PathCE in this section, a path-centric cardinality estimation framework. We begin with an overview of PathCE (Sec.4.1), followed by the key challenges in developing PathCE (Sec.4.2).



u3
u4
u2
u6
u8
u9
u5
u1
u7
Q
P1
P2
P3
P4
P5
P6
$\mathcal{S}$ = [u9, u7, u5, u3, u1]
(a) A query Q
(b) An estimation scheme of Q

Figure 3: A graph query Q and its estimation scheme $(\mathbf{Q}, \mathcal{S})$.



(e) Subqueries processed in iterations; designated endpoints are marked in black.

Figure 4: Iterative cardinality estimation. A dotted circle indicates a dummy vertex, while a gray edge represents a subquery.

4.1 Cardinality Estimation Workflow

Taking a graph query Q and a summary graph $\mathcal{G}$ of G as input, PathCE outputs an estimate of $|Q(G)|$. Algorithm 1 outlines the overall workflow of PathCE, which consists of an *estimation scheme generation* step (line 1) and a *cardinality estimation* step (lines 2–6).

Estimation scheme generation. PathCE decomposes the input query Q and generates an estimation scheme for it. This scheme is represented as a pair $(\mathbf{Q}, \mathcal{S})$ (see Fig. 3(b) for an example), where (i) $\mathbf{Q}$ is also a query such that every edge of $\mathbf{Q}$ represents a sub-path query of Q whose statistics have been included in $\mathcal{G}$, and (ii) $\mathcal{S}$ specifies an *estimation order* $[u_1, \ldots, u_k]$ that will be used in the estimation step (see below), where $u_1, \ldots, u_k$ are the vertices of $\mathbf{Q}$. We will refer to $\mathbf{Q}$ as a PSG *query of* Q, to highlight that PathCE utilizes $\mathbf{Q}$ directly over the PSG graph $\mathcal{G}$ to estimate $|Q(G)|$.

Example 5: A graph query Q and its estimation scheme $(\mathbf{Q}, \mathcal{S})$ are shown in Figure 3. Every edge in $\mathbf{Q}$ represents a path of Q, *e.g.,* the edges labeled with P_1 and P_2 represent the two paths $u_1–u_2–u_3$ and $u_3–u_4–u_5$ of Q, respectively. The estimation order is $\mathcal{S} = [u_9, u_7, u_5, u_3, u_1]$. Note that $\mathbf{Q}$ has a simpler structure than Q, *e.g.,* $\mathbf{Q}$ has 5 vertices and 6 edge, whereas Q has 9 vertices and 10 edges. □

Iterative cardinality estimation. Guided by $(\mathbf{Q}, \mathcal{S})$, PathCE next estimates $|Q(G)|$ by making use of the path query statistics included in $\mathcal{G}$. PathCE performs the estimation in an iterative manner, as also adopted in prior studies [18, 47]. One key difference is that, PathCE uses $\mathbf{Q}$ to leverage path query statistics included in $\mathcal{G}$ and reduce the number of estimation iterations, while prior approaches usually estimate directly with Q and require more iterations.

Let $\mathcal{S} = [u_1, \ldots, u_k]$ be the estimation order. As outlined in Algorithm 1, in the i-th iteration PathCE estimates the statistics of a subquery Q_i of $\mathbf{Q}$ (lines 2-5). The subquery Q_i takes $\mathsf{Nbr}(u_i)$ as its designated endpoints, where $\mathsf{Nbr}(u_i)$ are u_i's neighbors in Q_i. In particular, each iteration involves the following three steps.

(1) Extract a subquery. Let $\mathsf{Nbr}(u_i)$ be the neighbors of u_i in the PSG query $\mathcal{Q}$. PathCE extracts a subquery $Q_i[\mathsf{Nbr}(u_i)]$ from Q, consisting of the edges corresponding to u_i's incident edges in $\mathcal{Q}$ (line 3). For example, in Fig. 3(b), u_9 has one neighbor u_5 and one incident edge labeled P_5, so $Q_1[u_5]$ includes the two edges in Q covered by P_5 (see Fig. 3(e); vertex labels omitted for simplicity).

(2) Estimate SE*-triples.* With $\mathsf{Nbr}(u_i)$ as the designated endpoints, PathCE next estimates the statistics of $Q_i[\mathsf{Nbr}(u_i)]$ and includes them to $\mathcal{G}$ (line 4). Recall that we require $\mathsf{Nbr}(u_i) \leq 2$ to ensure the statistics remain representable in $\mathcal{G}$. The statistics are computed by combining the relevant SE-triples included in $\mathcal{G}$ (see Sec. 6).

(3) Reduce PSG *query.* PathCE then simplifies $\mathcal{Q}$ by replacing the subquery $Q_i[\mathsf{Nbr}(u_i)]$ with a new edge (line 5). Specifically, PathCE eliminates u_i and its incident edges from $\mathcal{Q}$ and adds a new edge labeled $Q_i[\mathsf{Nbr}(u_i)]$ to connect u_i's neighbors. Here, u_i is referred to as an *eliminated node*. When $\mathsf{Nbr}(u_i)$ consists of a single vertex u', then a dangling edge labeled $Q_i[u']$ is attached to u'. An edge is *dangling* if one of its endpoints links to an eliminated node, *e.g.,* u_i. See Fig. 4(a) for an example, where an eliminated node is depicted with a doted cycle. This step simplifies $\mathcal{Q}$ since it decreases the vertex number by one. Also note that the statistics for the new edge labeled $Q_i[\mathsf{Nbr}(u_i)]$ are represented as SE-triples and have been computed in the previous step. Consequently, the estimation iteration continues with the revised $\mathcal{Q}$ and updated $\mathcal{G}$.

After $k-1$ iterations, $\mathcal{Q}$ consists of only the last vertex u_k, with t dangling edges attached to u_k, where $t \geq 1$. PathCE applies a function AggCnt to aggregate statistics of $Q[u_k]$. Specifically, for each summary vertex V_i with $\mathcal{L}(V_i) = L(u_k)$, AggCnt computes a count c_i to estimate $|\{\rho \in Q(G) \mid \rho(u_k) \in V_i\}|$. The final estimate $|Q(G)|$ is obtained by summing over all such c_i values (line 6). The computation of c_i is performed by exploring the statistics of the dangling edges attached to u_k in $\mathcal{Q}$ (see Sec. 6 for details).

Example 6: Figures 4(a)–4(d) illustrate how PathCE estimates $|Q(G)|$, using the scheme $(\mathcal{Q}, \mathcal{S})$ from Fig. 3. The subqueries involved in the estimation process are shown in Fig. 4(e).

(1) In the first iteration, PathCE processes $Q_1[u_5]$ with u_5 as the designated endpoint, as shown in Fig. 4(e). The resulting SE-triples are of the form (c, d, ∞). After removing $Q_1[u_5]$, a dangling edge labeled $Q_1[u_5]$ is attached to u_5, and u_9 is eliminated from $\mathcal{Q}$ (depicted as a dotted circle in Fig. 4(a)). The second iteration then processes $Q_2[u_5]$, resulting in the updated PSG query shown in Fig. 4(b).

(2) In the third iteration, PathCE processes the subquery $Q_3[u_1, u_3]$, with u_1 and u_3 as designated endpoints (see Fig. 4(e)). It combines the initial statistics for P_2, P_3, and P_4 from $\mathcal{G}$ with those of the dangling edges $Q_1[u_5]$ and $Q_2[u_5]$ to compute new SE-triples. Then, u_5 and its incident edges are removed from $\mathcal{Q}$ and replaced by a new edge labeled $Q_3[u_1, u_3]$ (see Fig. 4(c)).

(3) In the final iteration, PathCE eliminates u_3 and computes the SE-triples for the subquery $Q_4[u_1]$. Figure 4(d) shows the revised $\mathcal{Q}$ with a single vertex u_1 and a dangling edge labeled $Q_4[u_1]$.

In the end, $\mathcal{Q}$ reduces to u_1 with a single dangling edge. For each summary vertex V_i such that $\mathcal{L}(V_i) = L(u_1)$, PathCE computes an estimate c_i for $|\{\rho \in Q(G) \mid \rho(u_1) \in V_i\}|$. Since u_1 has a single dangling edge, we let $c_i = c$, where (c, d, ∞) is the SE-triple associated with the summary edge $(V_i, U, Q_4[u_1])$ in $\mathcal{G}$, computed in the fourth iteration. The sum of all c_i is then returned as $|Q(G)|$. □

4.2 Challenges and Approaches

We highlight the challenges in developing the PathCE framework.

Challenge 1: Effective estimation scheme generation. Generating an effective estimation scheme requires solving two problems. First, how to construct a PSG query $\mathcal{Q}$ from Q with as few vertices as possible, since fewer vertices mean fewer estimation iterations and lower error. Second, how to derive a valid estimation order $\mathcal{S}$ for $\mathcal{Q}$ so that the overall estimation process can proceed as a sequence of SE-triple computation. To see the difficulty, consider the case where $\mathcal{Q}$ is a 4-clique. Then any estimation order of $\mathcal{Q}$ starts with a subquery in which the eliminated node has 3 neighbors. This violates the requirements of the extracted subqueries.

To address this, we propose a divide-and-conquer algorithm Dcmp that computes a PSG query $\mathcal{Q}$ with the minimal number of vertices for a given query Q (Sec. 5.1), and a *minimal neighbors* strategy to generate a feasible estimation order, if possible (Sec. 5.2). If none exists, such as in the 4-clique case, we apply a *pruning strategy* to remove from Q some edges that have minimal impact on estimation accuracy, creating a new query Q' that allows a feasible elimination order from the PSG query of Q' .

Challenge 2: Cardinality upper bound estimation. Prior studies show that upper bound estimates help generate high-quality query plans [4, 10, 47]. Moreover, upper bound estimation usually does not rely on statistical assumptions and offers theoretical guarantees [1, 3]. The challenge is how to provide upper bound estimates within PathCE, ensuring that each estimate C satisfies $|Q(G)| \leq C$.

Inspired by [1, 4], we have included *maximal-degree* statistics in the SE-triples for path queries and PSG construction. These statistics are then used for estimation within each extracted subquery (Sec. 6). We employ a graph-based way to compute upper bound estimates for each such subquery. By an inductive analysis, we can show that $|Q(G)| \leq C$ (Theorem 3, Sec. 6).

Challenge 3: Scalable PSG construction. Given a graph G, a vertex partition Π_V and a path query set $\mathbb{P}$, it is non-trivial to construct a PSG of G *w.r.t.* Π_V and $\mathbb{P}$ since each SE-triple includes both count and maximal-degree statistics of a path query P. A naive approach is to compute the match set $P(G)$ first and then derive the SE-triples from $P(G)$. However, careful analysis indicates that this would incur significant computational and space overheads.

To address this challenge, we develop a parallelly scalable algorithm PSGBuilder to construct PSG's in linear time *w.r.t.* the input graph size $|G|$. PSGBuilder achieves this by utilizing a compact intermediate representation of SE-triples and conducting the computation in a bottom-up manner. Better still, the algorithm PSGBuilder is *parallelly scalable* [22], *i.e.,* it guarantees to reduce the running time when working with more processors. This implies that PSGBuilder is capable of constructing PSG's for large graphs.

Remark. PathCE is a path-centric cardinality estimator that differs from FactorJoin [47], CEG [6] and GLogS [23] in the followings.

(1) PathCE relies exclusively on statistics of path queries, organized in a PSG graph. When path queries are restricted to single-edge

Algorithm 2: GenScheme: Estimation scheme generation

```
Input: A graph query $Q$ and a PSG $\mathcal{G}$ w.r.t. graph $G$.
Output: An estimation scheme $(\mathcal{Q}, \mathcal{S})$ for $Q$.
1 $\mathcal{Q} \leftarrow$ Dcmp($Q, \mathcal{G}$);  $\mathcal{S} \leftarrow$ GenEstOrder($\mathcal{Q}$);
2 while $\mathcal{S} = \emptyset$ do
3     $Q \leftarrow$ Prune($Q, \mathcal{Q}$);
4     $\mathcal{Q} \leftarrow$ Dcmp($Q, \mathcal{G}$);  $\mathcal{S} \leftarrow$ GenEstOrder($\mathcal{Q}$);
5 return $(\mathcal{Q}, \mathcal{S})$;
```

patterns, the statistics used by PSG reduce to those adopted by FactorJoin. In contrast, CEG and GLogS consider statistics of general graph queries beyond path queries, *e.g.,* triangles.

(2) While PathCE and FactorJoin use maximal-degree statistics to ensure cardinality upper bounds, PathCE adopts an graph-based approach, whereas FactorJoin uses a probabilistic graphical model. Both CEG and GLogS rely on statistical assumptions for estimation and do not guarantee cardinality upper bounds.

(3) PathCE builds statistics in a parallelly scalable manner, making it suitable for large graph datasets. In contrast, FactorJoin and CEG do not include statistics construction algorithms. While GLogS includes its own statistics construction mechanism, it relies on graph sparsification to scale to large graphs.

5 ESTIMATION SCHEME GENERATION

In this section, we show how to generate an estimation scheme for a given query Q. We outline the generation workflow as follows.

Workflow. As outlined in Algorithm 2, PathCE first utilizes a procedure Dcmp to decompose Q into a PSG query $\mathcal{Q}$. The optimization goal of Dcmp is to minimize the number of vertices of $\mathcal{Q}$. Next, PathCE tries to compute an estimation order $\mathcal{S}$ from $\mathcal{Q}$ by GenEstOrder (line 1). If that fails, PathCE then tries to prune some edges from Q so that $\mathcal{Q}$ has a better chance to generate a feasible estimation order (line 3). After pruning, PathCE retries Dcmp to compute a new $\mathcal{Q}$ and GenEstOrder to find a new $\mathcal{S}$ (line 4). Once a feasible estimation order $\mathcal{S}$ is identified, PathCE returns $(\mathcal{Q}, \mathcal{S})$ as the estimation scheme for Q (line 5).

In the rest of this section, we discuss the ideas behind the procedures Dcmp, GenEstOrder and Prune in more details.

5.1 PSG Query Computation

We first develop an algorithm Dcmp to decompose a given query Q into a PSG query $\mathcal{Q}$ such that $\mathcal{Q}$'s vertex number is minimized. The PSG query $\mathcal{Q}$ of Q satisfies the following requirements.

- **(R0)** The vertex set of $\mathcal{Q}$ is a subset of Q.
- **(R1)** Each edge e in $\mathcal{Q}$ represents a subpath P of Q with statistics included in $\mathcal{G}$. In that case, we say that e *covers* the edges of P.
- **(R2)** Every edge of Q is covered by exactly one edge of $\mathcal{Q}$.
- **(R3)** $\mathcal{Q}$'s vertices can only occur as the endpoints of the subpaths that $\mathcal{Q}$ covers, which is to ensure the estimation correctness.

Note that from (R0)(R1)(R2)(R3), if a vertex u of Q has a degree ≥ 3, then u must be kept in $\mathcal{Q}$. We call each such vertex u a *pivot* of Q. With the pivots, Dcmp obtains a PSG query $\mathcal{Q}$ as outlined below.

Algorithm Dcmp. The algorithm Dcmp first identifies all the pivots of Q. From these pivots, it conducts a series of DFS traversal on Q to compute a set PSet of subpaths of Q. Specifically, starting from a pivot u, a DFS traversal terminates when it either hits a node with degree 1, *i.e.,* the DFS cannot continue, or it hits a vertex with degree ≥ 3, *i.e.,* the starting pivot u or another pivot. Each DFS traversal is included as a path in PSet. It is possible that statistics of a path query P in PSet are *not* in $\mathcal{G}$. By requirement (R1), Dcmp next utilizes a sub-routine PathDP to decompose each such path P in Pset into paths $P_1,\ldots,P_k$ to minimize k so that the statistics of every P_i are included in $\mathcal{G}$. Each P_i and its endpoints are then added as an edge to $\mathcal{Q}$, *i.e.,* these P_i's are used by $\mathcal{Q}$ to cover Q. PathDP adopts dynamic programming to obtain a decomposition of P as follows. Let $P = u_0e_1u_1\ldots u_{n-1}e_nu_n$ be an n-path in PSet and $\mathbb{P}$ be the set of path queries that $\mathcal{G}$ is constructed from. Denote by $P_{ij} = u_ie_{i+1}u_{i+1}\ldots u_{j-1}e_ju_j$ the subpath of P between v_i and v_j, where $0 \leq i \leq j \leq n$, and define $f(P)$ as the minimum number of paths in $\mathbb{P}$ to decompose P. We give the recurrence as follows:

$$f(P_{ij}) = \begin{cases} 1 & P_{ij} \in \mathbb{P} \\ \min\{f(P_{ik}) + f(P_{kj}) \mid i \leq k \leq j\} & \text{otherwise.} \end{cases} \quad (1)$$

Clearly, this recurrence can be solved using dynamic programming.

The algorithm Dcmp produces an optimal PSG query $\mathcal{Q}$ for a given graph query Q, as summarized in Proposition 1.

Proposition 1: *Given a query Q and $\mathcal{G}$, the algorithm* Dcmp *produces a* PSG *query $\mathcal{Q}$ of Q such that $\mathcal{Q}$ has the minimum number of vertices and meets the requirements of (R0),(R1), (R2) and (R3).* □

5.2 Estimation Order Generation and Pruning

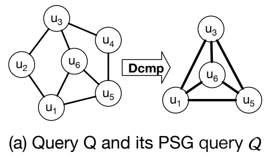
We next discuss how to generate a feasible estimation order for a PSG query $\mathcal{Q}$. Let $\mathcal{S} = [u_1, \ldots, u_k]$ be a vertex order of $\mathcal{Q}$. We say that $\mathcal{S}$ is a *feasible* estimation order for $\mathcal{Q}$ if, in every subquery produced using the scheme $(\mathcal{Q}, \mathcal{S})$, we have $\mathsf{Nbr}(u_i) \leq 2$.

Below, we first introduce the *minimal neighbors strategy* used by GenEstOrder to produce a feasible estimation order $\mathcal{S}$, if such an order exists. Next, we propose a pruning strategy to handle cases where no feasible $\mathcal{S}$ can be generated for $\mathcal{Q}$.

Minimal neighbors strategy. Given $\mathcal{Q}$, GenEstOrder simulates the estimation process and uses a *minimal neighbors* strategy to determine $\mathcal{S}$. In the i-th iteration, GenEstOrder selects a vertex v with the fewest neighbors in $\mathcal{Q}$ as the i-th vertex in $\mathcal{S}$. If multiple vertices have the minimal number of neighbors, it randomly picks one. GenEstOrder then removes u and its incident edges from $\mathcal{Q}$. If u has exactly two neighbors in $\mathcal{Q}$, GenEstOrder also adds a new edge connecting them after removing u, to simulate the estimation process. If u has more than two neighbors, *i.e.,* this would lead to an invalid subquery during estimation, GenEstOrder terminates and the estimation order generation fails. Otherwise, the process continues with the updated $\mathcal{Q}$. If no failure occurs, GenEstOrder returns the vertex sequence $\mathcal{S}$ as the estimation order.

The lemma below shows that (i) GenEstOrder suffices for acyclic queries, and (ii) additional efforts are needed for cyclic queries.

Lemma 2: *(i)* GenEstOrder *always returns a feasible estimation order $\mathcal{S}$ for an acyclic $\mathcal{Q}$. (ii) If* GenEstOrder *fails for a cyclic $\mathcal{Q}$, then any permutation of $\mathcal{Q}$'s vertices is not feasible for $\mathcal{Q}$.* □



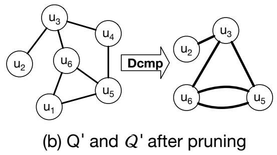


Figure 5: An example query that needs pruning

Clearly, GenEstOrder fails on the query $\mathcal{Q}$ depicted in Fig. 5(a) and any vertex order of $\mathcal{Q}$ is not a feasible estimation order. In light of Lemma 2, we propose a novel pruning strategy to handle cyclic queries when no feasible estimation order can be found.

Pruning strategy. If GenEstOrder fails for a PSG query $\mathcal{Q}$ of Q, we prune edges from Q to obtain a subquery Q', such that its PSG query $\mathcal{Q}'$ has a feasible estimation order $\mathcal{S}'$. Estimation is then performed using $(\mathcal{Q}', \mathcal{S}')$. To ensure the result is still an upper bound on $|Q(G)|$, only edges within a cycle of Q are pruned.

To illustrate, consider the query Q in Fig. 5(a), where no feasible estimation order is available for its PSG query $\mathcal{Q}$. By pruning the edge (u_1, u_2) from Q, we obtain the subquery Q', as shown in Fig. 5(b). Assuming $\mathcal{G}$ includes statistics for all path queries of length up to 2, the new PSG query $\mathcal{Q}'$ for Q' is shown in Fig. 5(b), and $\mathcal{S}' = [u_2, u_3, u_6, u_5]$ is a feasible estimation order for $\mathcal{Q}'$. Let C be the estimated cardinality using $(\mathcal{Q}', \mathcal{S}')$. As will be shown in Sec. 6, $|Q'(G)| \leq C$, which means C remains an upper bound on $|Q(G)|$ since $|Q(G)| \leq |Q'(G)| \leq C$. The pruned edge (u_1, u_2) lies within a cycle of Q, so any match for Q is also a match for Q'.

6 SUBQUERY ESTIMATION

In this section, we show how to estimate the statistics for a subquery Q extracted in an estimation iteration.

We consider the case where Q has two designated endpoints, u_1 and u_2, connected via an eliminated node u; the single-endpoint case is treated as a special case. For each summary vertex pair (V_{u_1}, V_{u_2}) matching (u_1, u_2), we compute an SE-triple (x, y, z) to estimate the summary of $Q[u_1, u_2]$ *w.r.t.* (V_{u_1}, V_{u_2}). We adopt a graph-based approach to compute (x, y, z), based on the observation that matches on the data graph are accurate, while those on the PSG graph serve as approximations. Our approach consists of two steps: PSG query matching and SE-triple estimation.

(1) PSG **query match**. This step is to extract SE-triples encoded in $\mathcal{G}$ for the estimation in the second step. Let $\mathcal{Q}[u_1, u_2]$ be the corresponding PSG query of $Q[u_1, u_2]$. Intuitively, each edge of $\mathcal{Q}[u_1, u_2]$ represents a sub-path of $Q[u_1, u_2]$ (see Fig. 6 for an example). We first run the PSG query $\mathcal{Q}[u_1, u_2]$ *on* PSG *graph* $\mathcal{G}$ and require that u_1 is matched to V_{u_1} and u_2 is matched to V_{u_2}. This is well-defined since $\mathcal{G}$ is also a graph by design. Note that $\mathcal{Q}[u_1, u_2]$ may have dummy nodes, *i.e.,* nodes eliminated during estimation (see the dotted circles in Fig. 6(b)). We require dummy nodes to match the dummy summary vertex U of $\mathcal{G}$.

By the definition of PSG, there are exactly M matches of $\mathcal{Q}[u_1, u_2]$ in $\mathcal{G}$. Specifically, let $V_1, \dots V_M$ be the summary vertices matching u. Each match can be represented as a subgraph $\mathcal{G}_i$ of $\mathcal{G}$, where node u is mapped to V_i in $\mathcal{G}_i$. The vertex set of $\mathcal{G}_i$ includes V_{u_1}, V_{u_2} and V_i, and U (if $\mathcal{Q}[u_1, u_2]$ contains an eliminated node). For the query $\mathcal{Q}[u_1, u_2]$ as shown in Fig. 6(b), a match $\mathcal{G}_i$ is shown

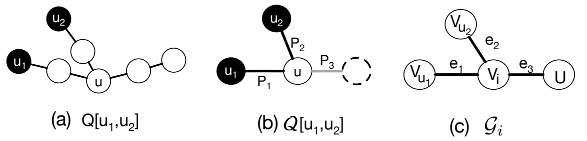


Figure 6: Query $Q[u_1, u_2]$ and its PSG version $\mathcal{Q}[u_1, u_2]$, along with a subgraph $\mathcal{G}_i$ of $\mathcal{G}$ that matches $\mathcal{Q}[u_1, u_2]$.

in Fig. 6(c). The label of each summary edge in $\mathcal{G}_i$ also matches that in $\mathcal{Q}[u_1, u_2]$. For example, the summary edge e_1 of $\mathcal{G}_i$ has label P_1, matching the edge between u_1 and u.

Fix a PSG subgraph $\mathcal{G}_i$ and let $\mathcal{G}'$ be a connected subgraph of $\mathcal{G}_i$. Let Q' be the query formed by connecting the subqueries encoded as edge labels in $\mathcal{G}'$. Note that , Q' must be a subquery of Q, and we have $Q' = Q$ when $\mathcal{G}' = \mathcal{G}_i$. Along the same lines as Definition 3.1, we define the following for each subgraph $\mathcal{G}'$ of $\mathcal{G}_i$:

$$
\begin{aligned}
c(\mathcal{G}') &= \left|\{\rho \in Q'(G) \mid \rho(u_1) \in V_{u_1}, \rho(u_2) \in V_{u_2}, \rho(u) \in V_i\}\right|,\\
d_1(\mathcal{G}') &= \max_{v_1 \in V_{u_1}} \left|\{\rho \in Q'(G) \mid \rho(u_1) = v_1, \rho(u_2) \in V_{u_2}, \rho(u) \in V_i\}\right|,\\
d_2(\mathcal{G}') &= \max_{v_2 \in V_{u_2}} \left|\{\rho \in Q'(G) \mid \rho(u_1) \in V_{u_1}, \rho(u_2) = v_2, \rho(u) \in V_i\}\right|.\\
d(\mathcal{G}') &= \max_{v \in V_i} \left|\{\rho \in Q'(G) \mid \rho(u_1) \in V_{u_1}, \rho(u_2) \in V_{u_2}, \rho(u) = v\}\right|.
\end{aligned}
$$

Then, by Definition 3.1 and the observation that $\{V_1, \dots, V_M\}$ represents a partition of the vertices matching u, we must have

$$
(c, d_1, d_2) = \sum \left\{ (c(\mathcal{G}_i), d_1(\mathcal{G}_i), d_2(\mathcal{G}_i)) \;\middle|\; i = 1, \dots, M \right\},
$$

where (c, d_1, d_2) is the summary of $Q[u_1, u_2]$ *w.r.t.* (V_{u_1}, V_{u_2}).

(2) SE-**triple estimation**. For each PSG subgraph $\mathcal{G}_i$, we first compute a triple (x_i, y_i, z_i) such that $c(\mathcal{G}_i) \leq x_i$, $d_1(\mathcal{G}_i) \leq y_i$, and $d_2(\mathcal{G}_i) \leq z_i$. We then let $(x, y, z) = \sum_{i=1}^{M}(x_i, y_i, z_i)$ be the estimate for (c, d_1, d_2). The following theorem is immediate.

Theorem 3: (1) $(c, d_1, d_2) \leq (x, y, z)$. **(2)** *The estimated cardinality returned by Algorithm 1 is an upper bound on the true cardinality.* □

Fix a PSG subgraph $\mathcal{G}_i$, we next compute a triple (x_i, y_i, z_i) that upper bounds $(c(\mathcal{G}_i), d_1(\mathcal{G}_i), d_2(\mathcal{G}_i))$, which can be done by leveraging the maximal-degree statistics as prior studies [4, 47]. To simplify our discussion, let us consider the $\mathcal{G}_i$ as shown in Fig. 6(c). Let $\mathcal{G}_{e_1}$, $\mathcal{G}_{e_2}$, and $\mathcal{G}_{e_3}$ be the subgraphs of $\mathcal{G}_i$ consisting of the single edges e_1, e_2, and e_3; and $\mathcal{G}'_{e_1}$, $\mathcal{G}'_{e_2}$, and $\mathcal{G}'_{e_3}$ the PSG subgraphs obtained by pruning e_1, e_2, and e_3 from $\mathcal{G}_i$, respectively.

(i) Bounds on maximal-degree statistics can be obtained in a bottom-up manner. For example, for $d_1(\mathcal{G}_i)$, $d_2(\mathcal{G}_i)$, and $d(\mathcal{G}_i)$, we have

$$
\begin{gathered}
d_1(\mathcal{G}_i) \leq d_1(\mathcal{G}_{e_1}) \cdot d(\mathcal{G}'_{e_1}), \quad d_2(\mathcal{G}_i) \leq d_2(\mathcal{G}_{e_2}) \cdot d(\mathcal{G}'_{e_2}),\\
d(\mathcal{G}_i) \leq \min\{d(\mathcal{G}_{e_1}) \cdot d(\mathcal{G}'_{e_1}),\ d(\mathcal{G}_{e_2}) \cdot d(\mathcal{G}'_{e_2}),\ d(\mathcal{G}_{e_3}) \cdot d(\mathcal{G}'_{e_3})\}.
\end{gathered}
$$

Observe that each graph on the RHS is a subgraph of the one on the LHS. By replacing "≤" with "=", we can compute upper bounds for $d_1(\mathcal{G}')$, $d_2(\mathcal{G}')$, and $d(\mathcal{G}')$ for each subgraph $\mathcal{G}'$ of $\mathcal{G}_i$. We then set y_i and z_i to be the upper bounds on $d_1(\mathcal{G}_i)$ and $d_2(\mathcal{G}_i)$, respectively.

(ii) Observe that $c(\mathcal{G}_i) \leq c(\mathcal{G}_{e_1}) \cdot d(\mathcal{G}'_{e_1})$. Indeed, (a) there are at most $c(\mathcal{G}_{e_1})$ matches of P_1 such that u_1 is mapped a vertex in V_{u_1} and u is mapped to a vertex in V_i; (b) and a match can be extended to at most $d(\mathcal{G}'_{e_1})$ matches of Q. Similarly, $c(\mathcal{G}_i) \leq c(\mathcal{G}'_{e_1}) d(\mathcal{G}_{e_1})$.

By considering all three edges e_1, e_2 and e_3 of $\mathcal{G}_i$, we have

$$c(\mathcal{G}_i) \le \min\{c(\mathcal{G}_{e_1})\cdot d(\mathcal{G}'_{e_1}),\ c(\mathcal{G}_{e_2})\cdot d(\mathcal{G}'_{e_2}),\ c(\mathcal{G}_{e_3})\cdot d(\mathcal{G}'_{e_3})\},$$
$$c(\mathcal{G}_i) \le \min\{c(\mathcal{G}'_{e_1})\cdot d(\mathcal{G}_{e_1}),\ c(\mathcal{G}'_{e_2})\cdot d(\mathcal{G}_{e_2}),\ c(\mathcal{G}'_{e_3})\cdot d(\mathcal{G}_{e_3})\}.$$

By replacing "$\le$" with "=" and applying a bottom-up computation, we can derive an upper bound for $c(\mathcal{G}_i)$ and set it as x_i.

7 SCALABLE PSG CONSTRUCTION

We develop a scalable algorithm PSGBuilder for PSG construction.

We first construct a path query set $\mathbb{P}$ for a given graph $G = (V, E, L)$. PSGBuilder enumerates all possible k-path queries for $k \le K$, by utilizing a graph $G_S = (V_S, E_S, L_S)$ that specifies the the valid vertex types and edges types of G [30]. More specifically, G_S is also a labeled graph satisfying the following: (a) $L_S(u_1) \ne L_S(u_2)$ for $u_1 \ne u_2$, and (b) for each edge (u_1, u_2, ℓ) in E, there is an edge (u'_1, u'_2, ℓ) in E_S such that $L(u_1) = L_S(u'_1)$ and $L(u_2) = L_S(u'_2)$. PSGBuilder conducts a series of k-walks on G_S and includes them as k-path queries in $\mathbb{P}$, with duplicate k-path queries omitted.

Challenges. Given a graph G, a graph summary vertex partition Π_V and a path query set $\mathbb{P}$, PSGBuilder next constructs a PSG of G *w.r.t.* Π_V and $\mathbb{P}$. Let P be a path query in $\mathbb{P}$. The goal is to compute all summary edges and their associated SE-triples for P. Recall that PSGBuilder must also compute maximal-degree statistics, which is more challenging. A straightforward approach is to first compute $P(G)$ and then derive the SE-triples. This requires $O(k|E|)$ space and takes $O(|E|^k)$ time, since obtaining the maximal-degree statistics mandates scanning the entire $P(G)$. To address these challenges, PSGBuilder utilizes a compact intermediate representation of SE-triples and computes them in linear time *w.r.t.* $|G|$. Furthermore, the computation is *parallelly scalable* [22], *i.e.,* PSGBuilder guarantees to reduce the running time when working with more processors.

Compact representation. Consider a path query P in $\mathbb{P}$ with h_P and t_P as it head and tail vertices. Let $V_h^1, \ldots, V_h^M$ and $V_t^1, \ldots, V_t^M$ be the summary vertices in $\mathcal{V}$ with label $L(h_P)$ and $L(t_P)$, respectively. For each vertex v in $\bigcup_{i=1}^M V_h^i$, we associate with v a head vector $\mathcal{H}_v^P$ of size M, where the i-th component $\mathcal{H}_v^P[i]$ records the number of matches ρ of P such that $\rho(h_P) = v$ and $\rho(t_P)$ lies in the summary vertex V_t^i. Similarly, we associate with each vertex u in $\bigcup_{i=1}^M V_t^i$ a tail vector $\mathcal{T}_u^P$, where the i-th component $\mathcal{T}_u^P[i]$ tracks the number of matches ρ of P such that $\rho(t_P) = u$ and $\rho(h_P)$ belongs V_h^i.

The benefits of using head and tail vectors are twofold.

(**1**) We can compute the SE-triples for path query P by aggregating these vectors. Indeed, the SE-triple (c, d_1, d_2) associated with the summary edge (V_h^i, V_t^j, P) can be computed by

$$c = \sum\{\mathcal{H}_v^P[j] \mid v \in V_h^i\} = \sum\{\mathcal{T}_u^P[i] \mid u \in V_t^j\}, \quad (2)$$
$$d_1 = \max\{\mathcal{H}_v^P[j] \mid v \in V_h^i\}, \quad d_2 = \max\{\mathcal{T}_u^P[i] \mid u \in V_t^j\}. \quad (3)$$

(**2**) We can compute the head and tail vectors in a bottom-up manner, *without* first computing and storing the path matches. Let e_0 and e_1 be the edges attached to h_P and t_P, and P_0 and P_1 be the path queries obtained by removing e_0 and e_1 from P, respectively. Let v be a vertex in $\bigcup_{i=1}^M V_h^i$ and u be a vertex in $\bigcup_{i=1}^M V_t^i$. We can compute $\mathcal{H}_v^P$ and $\mathcal{T}_u^P$ from the vectors related to P_0 and P_1 by

$$\mathcal{H}_v^P = \sum\{\mathcal{H}_w^{P_0} \mid w \in \mathsf{Nbr}(v) \wedge \mathsf{Match}(v, w, e_0)\}, \quad (4)$$
$$\mathcal{T}_u^P = \sum\{\mathcal{T}_{w'}^{P_1} \mid w' \in \mathsf{Nbr}(u) \wedge \mathsf{Match}(w', u, e_1)\}. \quad (5)$$

Here, $\mathsf{Nbr}(v)$ records the neighbors of v in G, $\sum$ denotes pairwise summation over vector components, and $\mathsf{Match}(v, w, e_0)$ returns true if there is an edge from v to w that matches e_0, similarly for $\mathsf{Nbr}(u)$ and $\mathsf{Match}(w', u, e_1)$. Both P_0 and P_1 are shorter than P, allowing the head and tail vectors to be computed recursively.

Parallelly scalable processing. PSGBuilder computes the head and tail vectors, as well as the SE-triples, recursively in a parallelly scalable manner. Specifically, let $\mathbb{P}_i$ be the set of path queries in $\mathbb{P}$ of length i. In the i-th iteration, PSGBuilder computes all head and tail vectors for path queries in $\mathbb{P}_i$ using Equations (4) and (5). From these vectors, it applies Equations (2) and (3) to obtain all SE-triples for $\mathbb{P}_i$. Observe that the computation of head and tail vectors can be parallelized at the granularity of vertices of G, while SE-triple computation can be parallelized on the basis of summary vertex pairs.

Analysis. The running time is dominated by the computation of SE-triples, which is bounded by $O\left(\frac{1}{C}NM(|E| + |V|)\right)$. Here, C is the number of processors and N is the number of k-path queries in $\mathbb{P}$, bounded by $(|V_S| + |E_S|)^K$. Indeed, PSGBuilder requires $O(M|E|)$ time to process the head and tail vectors for each path query P in $\mathbb{P}$, and $O(M|V|)$ time to compute the SE-triples for P. Note that N depends on the vertex types and edge patterns in G, rather than the size of the data graph G. The $\frac{1}{C}$ factor reflects the effective parallelization of the computation. The overall space cost of the algorithm is $O(NM|V| + NM^2)$. This includes $O(NM|V|)$ space for head and tail vectors and $O(NM^2)$ for the SE-triples. By scheduling the processing of head and tail vectors in a DFS fashion [5], the space cost can be reduced to $O(KM|V| + NM^2)$.

Remark. Computing head and tail vectors is essential for PSG construction, which heavily relies on neighborhood access (see Equations (4) & (5)). Graph systems inherently support efficient neighborhood access, thus enable an efficient implementation of PSGBuilder.

8 EVALUATION

Experimental settings. We start with experimental settings.

Datasets and Queries. We use three well-established datasets and workloads: LDBC [2], IMDB [25], and AIDS [38].

(1) LDBC refers to a set of generated datasets from the Linked Data Benchmark Council's Social Network Benchmark, designed for evaluating GDBMSs. We denote LDBC datasets of scale factors 0.1, 0.3, 1, 3 and 10 by LDBC-0.1, LDBC-0.3, LDBC-1, LDBC-3, and LDBC-10, respectively. Unless specified, LDBC refers to LDBC-1. The queries on LDBC consist of 9 queries from the LSQB subgraph query benchmark (Q1–Q9 in [30]) and 11 GLogS queries (marked Q10–Q20, originally P1–P11 in [23]). We remove the properties from the LDBC datasets. Similarly, for all LDBC queries, we remove predicates and convert optional or negative edges into normal edges.

(2) IMDB is a real-world relational dataset containing 21 tables and around 10 million rows. It has been widely used with the Join Order Benchmark (JOB) [6, 25]. Following [42], we convert IMDB into a labeled graph with 12 vertex labels and 13 edge labels. We use the JOB

Table 2: Dataset statistics: $\#(L_v)$, $\#(L_e)$, $\#(Q)$ and $|\mathbb{P}|$ indicate the number of vertex labels, edge labels, queries evaluated, path queries for PSG construction, respectively.

Datasets	$\|V\|$	$\|E\|$	$\#(L_v)$	$\#(L_e)$	$\#(Q)$	$\|\mathbb{P}\|$
LDBC	3.73M	21.4M	11	25	20	1243
IMDB	52.6M	119M	12	13	33	471
AIDS	254K	548K	50	4	484	297

queries (referred to as IMDB queries hereafter) [25], consisting of 33 SQL templates. These query templates are converted into graph queries by retaining only the join query graph topology while omitting predicates and aggregations [42]. All IMDB queries are acyclic.

(3) AIDS is a real-world dataset widely used for subgraph matching [19, 34, 38], consisting of 43,905 chemical molecules. We use 484 queries provided by the subgraph cardinality estimation benchmark tool G-CARE [34], which includes 3-, 6-, and 9-path queries, as well as randomly generated star, tree, and generic graph queries.

The statistics of the datasets are summarized in Tab. 2. Note that the size ($|V|, |E|$) for LDBC varies with the scale factor, ranging from $(0.45M, 2.18M)$ for LDBC-0.1 to $(33.9M, 211M)$ for LDBC-10.

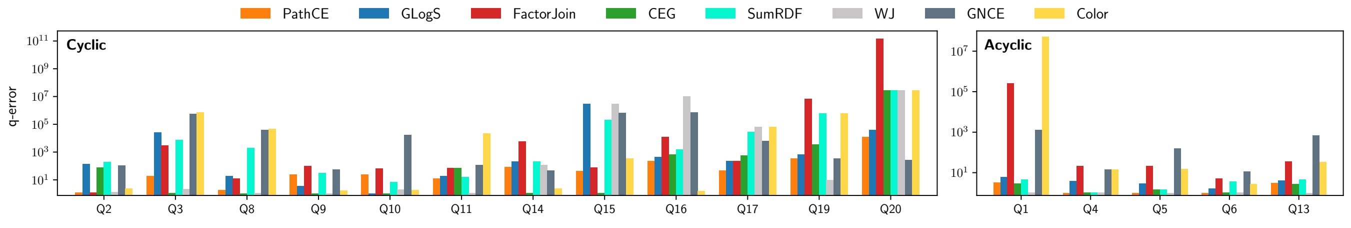
Baselines. We prototype PathCE as a Rust library, built from scratch, including the implementation of cardinality estimation and parallel PSG construction. PathCE adopts the GBSA algorithm developed in [47] to generate summary vertex partitions. GBSA is a greedy partitioning heuristic that divides vertices with label ℓ by minimizing the variance in their participation across path queries in $\mathbb{P}$ that have endpoints labeled ℓ. We follow the same setting as FactorJoin and sets $M = 200$ for fair comparison. For each dataset, PathCE sets $K = 3$ to collect all possible k-path queries for $k \leq K$. The number of path queries used for PSG construction for each dataset is summarized in Table 2. We compare PathCE with seven state-of-the-art cardinality estimators, including five summary-based methods: GLogS [23], CEG [6], FactorJoin [47], SumRDF [39] and Color [9], a sampling-based estimator WJ [26], and a ML-based estimator GNCE [36]. **(1)** GLogS [23] is an estimator whose summary records the counts of small patterns, not limited to paths. **(2)** CEG [6] is an estimation framework consisting of nine estimators. We use the estimator that achieves the most accurate estimation. We collect statistics for small patterns with at most 3 vertices for both GLogS and CEG. **(3)** FactorJoin [47] is an estimator based on the maximal-degree statistics of 1-path queries, *i.e.,* single edge queries. We set the number of bins to 200 for FactorJoin to match the parameter M in PathCE. **(4)** SumRDF [39] is a summary-based estimator designed for RDF graphs. **(5)** WJ [26] is a sampling-based estimator. For both SumRDF and WJ, we use the implementations and default configurations provided by G-CARE, *e.g.,* the sample ratio of WJ is set to 3%. **(6)** GNCE [36] is an ML-based estimator for knowledge graphs that leverages graph embeddings and GNNs. For offline training of GNCE, we sample 5000 path queries and 5000 star queries from each dataset. **(7)** Color [9] is a summary-based estimator utilizing graph coloring. We use its AvgMix32 variant, the default with the best estimation accuracy.

Environment. We conduct all the experiments on a Linux server with 32 physical cores and 256 GB RAM. For offline model training and cardinality estimation for GNCE, we use an NVIDIA RTX 4070 GPU. We use 32 threads for all baselines during summary construction and query execution, unless stated otherwise. Each experiment is run five times, and the average is reported.

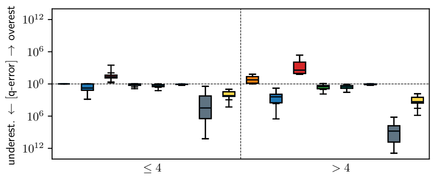
Exp-1: Estimation accuracy. We first evaluate the estimation accuracy of PathCE across all datasets. We adopt the classical q-error metric [31], defined as $\max\{\frac{e}{c}, \frac{c}{e}\}$, where e is the estimated and c the true cardinality for a given query Q. Lower q-error indicates better accuracy. Following prior work [34, 47], (a) each estimator is evaluated using its intended summary or ML model; (b) if a baseline fails to return an estimate within 300 seconds, we set $e = 1$.

(1) Varying query on LDBC. Figure 7(a) depicts the q-error of all tested queries on LDBC, where results are grouped as acyclic and cyclic queries. Note that Q7,Q12,Q18 are omitted in Fig. 7(a) since they are identical to Q4, Q2 and Q3 with predicates removed. (**a**) For acyclic LDBC queries, all summary-based estimators, except FactorJoin and Color, perform consistently well, with an average q-error below 5. (**b**) For cyclic LDBC queries, PathCE consistently outperforms FactorJoin and SumRDF. Note that PathCE, FactorJoin and SumRDF utilize path queries of lengths up to 3, 1 and 1, respectively. PathCE also are more accurate than Color on most cyclic queries. These confirm the effectiveness of using longer path queries to reduce estimation iterations. PathCE also shows better accuracy than GLogS and CEG, except for queries including triangles as subqueries, *e.g.,* $Q3$ and $Q8$. As will be highlighted in Exp-3, collecting triangle statistics is cost-prohibitive. (**c**) WJ produces accurate estimates for acyclic queries and small cyclic queries such as Q2 and Q3. However, WJ suffers from severe underestimation on Q15, Q16, Q17, and Q20, which contain large cycles as subqueries. This is primarily due to its sample-and-validate mechanism [19, 26], which makes it prone to sampling failures on cyclic queries, especially those involving large cycles like Q15. In contrast, for acyclic queries and simple cyclic patterns, *e.g.,* Q2 and Q3, WJ is more likely to obtain valid samples, leading to relatively accurate estimates. Similar behavior is also observed on other tested datasets. (**d**) GNCE in general produces less accurate estimates than others because GNCE is trained using sampled path and star queries, while the evaluated LDBC queries are mostly generic graph queries.

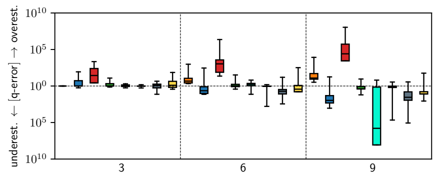
(2) Varying query size. Varying the query sizes of IMDB queries and AIDS path queries, Fig. 7(b) and Fig. 7(c) report the accuracy results. Here, the query size is defined as the number of edges in the query graph. Following an approach adopted by G-CARE [34], we reorder the estimates from the least accurate underestimation to the least accurate overestimation before generating the box plot. We observe the following. (**a**) On both datasets, PathCE, CEG, and WJ are the most accurate estimators. As the query size increases, the q-error of PathCE also increases but at a slower rate compared to FactorJoin and SumRDF. On AIDS, the average q-error of PathCE grows from 1 to 313, varying from 3-path queries to 9-path queries. (**b**) While PathCE consistently produces upper bound estimates, Color tends to underestimate most queries on IMDB and AIDS. Both CEG and GLogS shift from overestimation to underestimation as the path query size increases on AIDS, primarily due to their reliance on independence and uniformity assumptions. (**c**) GNCE is less accurate on IMDB than other baselines, similar to the case of LDBC. In contrast, GNCE has quite decent accuracy on AIDS. This is because we only evaluate path queries on AIDS in this setting.



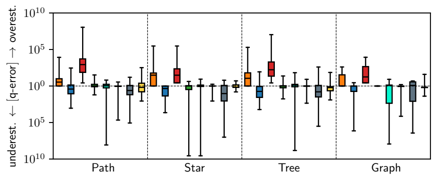
(a) Accuracy (varying query, LDBC)



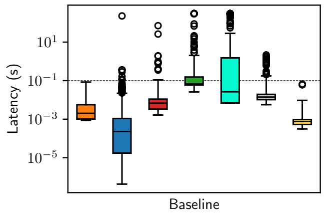
(b) Accuracy (varying query size, IMDB)



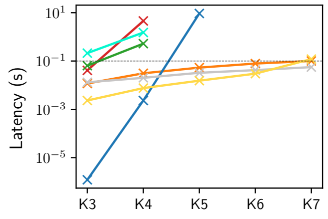
(c) Accuracy (varying path query size, AIDS)



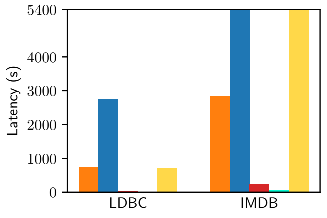
(d) Accuracy (varying query topology, AIDS)



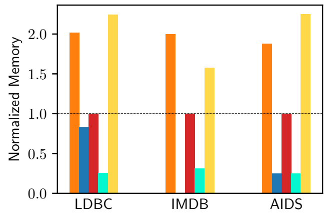
(e) Latency on all datasets



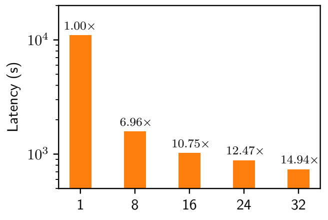
(f) Latency using K3–K7 (LDBC)



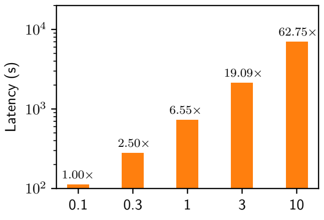
(g) Summary construction efficiency



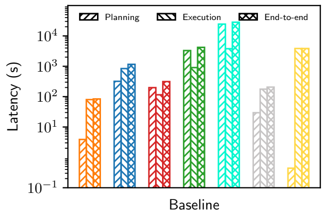
(h) Peak memory usage



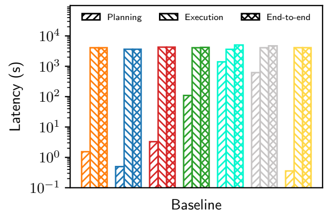
(i) Scalability (vary thread count)



(j) Scalability (vary scale factor)



(k) End-to-end evaluation (LDBC)



(l) End-to-end evaluation (IMDB)

Figure 7: Performance Evaluation

(3) Varying query topology. We divide all the AIDS queries into four topology categories: path, star, tree, and graph, and evaluate the q-error for all baselines. As shown in Fig. 7(d), PathCE, along with GLogS and CEG, performs consistently well across different query topologies. The key difference is that PathCE always produces upper bound estimates, whereas GLogS and CEG tend to underestimate. SumRDF shows good accuracy for path and star queries but suffers from significant accuracy degradation for other query types. WJ produces the most accurate estimates but tends to underestimate due to sampling failures for complex graph queries.

Exp-2: Estimation latency. We compare the estimation latency of all estimators on the LDBC, IMDB, and AIDS datasets. For each query Q, we record the time taken by each estimator to produce an estimate. As in Exp-1, the estimation timeout is set to 300 seconds. The results are shown in Fig. 7(e), where GNCE is omitted since it uses GPU for estimation, while others are CPU-based. In Fig. 7(e), the upper whisker is set as 20 times the IQR from the upper quartile.

(1) PathCE has the lowest average latency among all summary-based competitors, except for Color, which achieves the best estimation latency across nearly all cases. This is primarily due to its use of a topological estimation order, similar to PathCE, along with inference optimizations such as partial aggregation [9]. PathCE is also significantly more efficient than the sampling-based WJ, as expected. On LDBC, PathCE achieves an average latency of 0.02 seconds, significantly lower than GLogS, FactorJoin, CEG, SumRDF, and WJ, which take 13.43, 6.24, 24.57, 35.75, and 0.09 seconds, respectively. A similar trend is also observed on IMDB and AIDS.

(2) PathCE's stability is further highlighted by its low variance, even on cyclic LDBC queries. While GLogS achieves lower latency for most queries compared to PathCE, it struggles with complex queries, *e.g.,* 227.02 seconds for LDBC Q20, which contains 3 diamond subqueries. CEG faces similar challenges as GLogS, mainly due to the exponential search space during estimation. PathCE avoids this through an effective estimation scheme generation, assisted by pruning techniques (Sec. 5.2). In contrast, CEG's high latency limits its practical use, despite its good estimation accuracy. This becomes more evident in the end-to-end tests (see Exp-5).

(3) To explore further, we handcraft five additional clique queries

(K3 to K7) using person nodes and knows edges from the LDBC dataset. As shown in Fig. 7(f), PathCE remains stable with latencies under the practical threshold 0.1 seconds, showing only slight increases as clique size grows. Color also performs consistently well as PathCE. Other estimators, however, exhibit significant latency spikes. FactorJoin, CEG and SumRDF timeout from K5, and GLogS fails for K6 and K7. Although WJ achieves similar performance as PathCE, it encounters severe underestimation due to sampling failure starting from K4. PathCE's superior performance verifies the effectiveness of Dcmp algorithm and pruning techniques.

Exp-3: Summary construction efficiency. The efficiency of summary construction is critical for real-world deployment, as discussed in [4, 10, 34, 47]. We measure *(a) construction latency*, *(b) peak memory usage*, and *(c) on-disk storage requirements* for all summary-based estimators except CEG. We exclude CEG as it constructs summaries tailored only for each tested query. We find the following.

(1) Construction latency. As shown in Fig.7(g), PathCE constructs its summary on LDBC in 734 seconds, significantly faster than GLogS (2758 seconds). The performance gap is even more pronounced on IMDB, where PathCE completes the construction in 2831 seconds, while GLogS fails to finish within 1.5 hours since it enumerates a large number of matches during summary construction. In contrast, PathCE uses a compact representation of intermediate matches, which is much more computationally efficient. FactorJoin and SumRDF have the shortest construction time by using solely single-edge statistics. While PathCE and Color show comparable performance on LDBC, Color suffers from significant slowdown on IMDB, taking over 1.5 hours despite collecting only single-edge statistics. Given that IMDB is 6.8× larger than LDBC, this inefficiency is likely due to the single-threaded implementation of Color, which limits its scalability on large datasets. Since AIDS is much smaller than LDBC and IMDB, all baselines are able to complete summary construction within 60 seconds.

(2) Peak memory usage. Figure 7(h) reports the peak memory usage of all baselines. All values are normalized with FactorJoin treated as the baseline, *i.e.,* set to 1. PathCE incurs higher memory overhead than other summary-based estimators due to two main factors: (a) it maintains more intermediate results to compute both path counts and maximal-degree statistics, and (b) it stores both head and tail vertices in memory during summary construction (see Sec. 7).

(3) On-disk storage size. PathCE requires more disk space compared to FactorJoin and GLogS, *i.e.,* 1145.0 MB, 428.5 MB and 163.3 MB for LDBC, IMDB and AIDS, respectively. This storage requirement is still practically feasible since the PSG storage size of PathCE is independent *w.r.t.* the data graph size (Sec. 3). Moreover, a detailed analysis of the estimation logs reveals that on LDBC, only 68 out of 1243 collected path queries are used in cardinality estimation across our entire evaluation. This indicates that the storage overhead can be reduced by 94.5% through pruning unused path queries based on workload coverage. Similarly, on IMDB, storage can be reduced by 91.3%. In contrast, baselines like FactorJoin rely on single-edge queries and thus use nearly all collected statistics.

Exp-4: Scalability of summary construction. We next evaluate the scalability of PathCE, by varying thread counts and graph sizes. The results are shown in Fig. 7(i) and Fig. 7(j). **(1)** As shown in Fig. 7(i), PathCE achieves a 14.95× speedup when varying thread counts from 1 to 32 on dataset LDBC. **(2)** With LDBC datasets from LDBC-0.1 to LDBC-10, Fig. 7(j) reports PathCE's summary construction latency which grows almost linearly, consistent with its time complexity (Sec. 7). The storage size remains stable, 1145.0 MB (not shown), as it is determined by the vertex/edge types of the data graph. These results confirm PathCE's effective scaling with both graph size and number of threads during summary construction.

Exp-5: End-to-end performance. Following prior work [4, 10, 47], we evaluate the end-to-end query latency to assess the impact of PathCE's estimation accuracy on query optimization. We integrate all evaluated estimators into the bottom-up optimizer of the GLogS system [23], under a simplified abstraction: during plan enumeration, the optimizer issues cardinality requests for subqueries, and the estimator returns cardinality estimates. This abstraction allows decoupling estimation from plan enumeration, and has also been adopted by optimizers as in GOpt [29] and RelGo [27]. The results for LDBC and IMDB, including planning time, execution time and end-to-end time are shown in Fig. 7(k) and Fig. 7(l), respectively.

(1) PathCE beats all other baselines by 2.46× ~ 336.71× on LDBC in end-to-end time. PathCE is able to generate the best query plans especially for complex queries like Q19 and Q20 thanks to its accurate upper bound estimates. Thus, PathCE beats all baselines by 1.44× ~ 48.11× in execution time. PathCE also requires less planning time than all baselines except Color, which, while fastest in planning, suffers from suboptimal plans. Compared to FactorJoin, PathCE achieves a 3.73× speedup in E2E time, with 50.85× and 1.44× improvements in planning and query execution, respectively.

(2) On IMDB, all baselines show similar execution performance, as the queries are acyclic and simple. However, PathCE achieves 2.14× ~ 911.63× lower planning latency than FactorJoin, CEG, SumRDF, and WJ, and remains comparable to GLogS and Color (gap < 1.5s, negligible relative to end-to-end time).

(3) We also conduct a case study on Q19. Regarding planning, PathCE performs only slightly slower than GLogS, but much faster than other baselines. For query execution, PathCE, CEG and WJ share the best plan. FactorJoin's plan differs slightly but does not scale as well as PathCE. Unlike other estimators, GLogS and Color produce underestimations for Q19, resulting in suboptimal expansion-only plans with large intermediate results. The upper bound estimation of PathCE and FactorJoin helps prevent this issue. CEG and WJ do not face the issue due to more accurate estimation.

9 CONCLUSION

We propose PathCE, a path-centric cardinality estimation framework for subgraph matching. An interesting future direction is to apply PathCE to relational DBMSs. Another direction is to explore how to effectively maintain a PSG in response to graph updates.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments that helped us improve the paper. This work was supported by the National Natural Science Foundation of China (62202301).

REFERENCES

[1] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2017. What Do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 429–444.

[2] Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, David Püroja, Mirko Spasić, Benjamin A. Steer, Dávid Szakállas, Gábor Szárnyas, Jack Waudby, Mingxi Wu, and Yuchen Zhang. 2024. The LDBC Social Network Benchmark. arXiv:2001.02299 [cs.DB]

[3] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size Bounds and Query Plans for Relational Joins. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS '08)*. IEEE Computer Society, USA, 739–748.

[4] Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 18–35.

[5] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, Article 32, 12 pages.

[6] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Ken Salem. 2022. Accurate summary-based cardinality estimation through the lens of cardinality estimation graphs. *Proc. VLDB Endow.* 15, 8 (April 2022), 1533–1545.

[7] Yannis Chronis, Yawen Wang, Yu Gan, Sami Abu-El-Haija, Chelsea Lin, Carsten Binnig, and Fatma Özcan. 2024. CardBench: A Benchmark for Learned Cardinality Estimation in Relational Databases. arXiv:2408.16170 [cs.DB]

[8] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[9] Kyle Deeds, Diandre Sabale, Moe Kayali, and Dan Suciu. 2024. Color: A Framework for Applying Graph Coloring to Subgraph Cardinality Estimation. *Proc. VLDB Endow.* 18, 2 (Oct. 2024), 130–143.

[10] Kyle B. Deeds, Dan Suciu, and Magdalena Balazinska. 2023. SafeBound: A Practical System for Generating Cardinality Bounds. *Proc. ACM Manag. Data* 1, 1, Article 53 (May 2023), 26 pages.

[11] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2246–2258.

[12] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *DMTCS Proceedings (DMTCS Proceedings)*, Philippe Jacquet (Ed.), Vol. DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07). Discrete Mathematics and Theoretical Computer Science, Juan les Pins, France, 137–156.

[13] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1433–1445.

[14] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality estimation in DBMS: a comprehensive benchmark evaluation. *Proc. VLDB Endow.* 15, 4, 752–765.

[15] Hazar Harmouch and Felix Naumann. 2017. Cardinality estimation: an experimental survey. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 499–512.

[16] Stephen Harris and Nigel Shadbolt. 2005. SPARQL Query Processing with Conventional Relational Database Systems. In *Web Information Systems Engineering – WISE 2005 Workshops*, Mike Dean, Yuanbo Guo, Woochun Jun, Roland Kaschek, Shonali Krishnaswamy, Zhengxiang Pan, and Quan Z. Sheng (Eds.). Springer, Berlin, Heidelberg, 235–244.

[17] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1035–1050.

[18] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2021. Simplicity Done Right for Join Ordering. In *11th Biennial Conference on Innovative Data Systems Research (CIDR '21)*.

[19] Pan Hu and Boris Motik. 2024. Accurate Sampling-Based Cardinality Estimation for Complex Graph Queries. *ACM Trans. Database Syst.* 49, 3, Article 12 (Sept. 2024), 46 pages.

[20] Yannis E. Ioannidis and Stavros Christodoulakis. 1993. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Trans. Database Syst.* 18, 4 (Dec. 1993), 709–748.

[21] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned Cardinalities:Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research (CIDR '19)*.

[22] Clyde P Kruskal, Larry Rudolph, and Marc Snir. 1990. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science* 71, 1 (1990), 95–132.

[23] Longbin Lai, Yufan Yang, Zhibin Wang, Yuxuan Liu, Haotian Ma, Sijie Shen, Bingqing Lyu, Xiaoli Zhou, Wenyuan Yu, Zhengping Qian, Chen Tian, Sheng Zhong, Yeh-Ching Chung, and Jingren Zhou. 2023. GLogS: Interactive Graph Pattern Matching Query At Large Scale. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 53–69.

[24] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215.

[25] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query Optimization through the Looking Glass, and What We Found Running the Join Order Benchmark. *The VLDB Journal* 27, 5 (Oct. 2018), 643–668.

[26] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 615–629.

[27] Yunkai Lou, Longbin Lai, Bingqing Lyu, Yufan Yang, XiaoLi Zhou, Wenyuan Yu, Ying Zhang, and Jingren Zhou. 2024. Towards a Converged Relational-Graph Optimization Framework. *Proc. ACM Manag. Data* 2, 6, Article 252 (Dec. 2024), 27 pages.

[28] Yao Lu, Srikanth Kandula, Arnd Christian König, and Surajit Chaudhuri. 2021. Pre-training summarization models of structured datasets for cardinality estimation. *Proc. VLDB Endow.* 15, 3 (Nov. 2021), 414–426.

[29] Bingqing Lyu, Xiaoli Zhou, Longbin Lai, Yufan Yang, Yunkai Lou, Wenyuan Yu, Ying Zhang, and Jingren Zhou. 2025. A Modular Graph-Native Query Optimization Framework. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 566–579.

[30] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: A Large-Scale Subgraph Query Benchmark. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. ACM, Virtual Event China, 1–11.

[31] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 982–993.

[32] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1520–1533.

[33] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, USA, 984–994.

[34] Yeonsu Park, Seongyun Ko, Sourav S. Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: A Framework for Performance Benchmarking of Cardinality Estimation Techniques for Subgraph Matching. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1099–1114.

[35] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyuce, and Srikanta Tirthapura. 2018. Butterfly Counting in Bipartite Networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 2150–2159.

[36] Tim Schwabe and Maribel Acosta. 2024. Cardinality Estimation over Knowledge Graphs with Embeddings and Graph Neural Networks. *Proc. ACM Manag. Data* 2, 1, Article 44 (March 2024), 26 pages.

[37] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34.

[38] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 364–375.

[39] Giorgio Stefanoni, Boris Motik, and Egor V. Kostylev. 2018. Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1043–1052.

[40] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned cardinality estimation: a design space exploration and a comparative evaluation. *Proc. VLDB Endow.* 15, 1 (Sept. 2021), 85–97.

[41] Charalampos E. Tsourakakis, Petros Drineas, Eirinaios Michelakis, Ioannis Koutis, and Christos Faloutsos. 2011. Spectral Counting of Triangles via Element-Wise Sparsification and Triangle-Based Link Recommendation. *Social Network Analysis and Mining* 1, 2 (April 2011), 75–81.

[42] Wilco van Leeuwen, George Fletcher, and Nikolay Yakovets. 2023. A General Cardinality Estimation Framework for Subgraph Matching in Property Graphs. *IEEE Trans. on Knowl. and Data Eng.* 35, 6 (June 2023), 5485–5505.

[43] David Vengerov, Andre Cavalheiro Menck, Mohamed Zait, and Sunil P. Chakkappen. 2015. Join size estimation subject to filter conditions. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1530–1541.

[44] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: a normalizing flow based cardinality estimator. *Proc. VLDB Endow.* 15, 1 (Sept. 2021), 72–84.

[45] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are we ready for learned cardinality estimation? *Proc. VLDB Endow.* 14, 9 (May 2021), 1640–1654.

[46] Leonard Wörteler, Moritz Renftle, Theodoros Chondrogiannis, and Michael Grossniklaus. 2022. Cardinality Estimation Using Label Probability Propagation for Subgraph Matching in Property Graph Databases *(EDBT '22)*. OpenProceedings.org.

[47] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: A New Cardinality Estimation Framework for Join Queries. *Proceedings of the ACM on Management of Data* 1, 1 (May 2023), 1–27.

[48] Ziniu Wu, Amir Shaikhha, Rong Zhu, Kai Zeng, Yuxing Han, and Jingren Zhou. 2021. BayesCard: Revitilizing Bayesian Frameworks for Cardinality Estimation. arXiv:2012.14743 [cs.DB]

[49] Tianjing Zeng, Junwei Lan, Jiahong Ma, Wenqing Wei, Rong Zhu, Yingli Zhou, Pengfei Li, Bolin Ding, Defu Lian, Zhewei Wei, and Jingren Zhou. 2024. PRICE: A Pretrained Model for Cross-Database Cardinality Estimation. *Proc. VLDB Endow.* 18, 3 (Nov. 2024), 637–650.

[50] Kangfei Zhao, Jeffrey Xu Yu, Hao Zhang, Qiyan Li, and Yu Rong. 2021. A Learned Sketch for Subgraph Counting. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2142–2155.

[51] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1525–1539.

[52] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: fast, lightweight and accurate method for cardinality estimation. *Proc. VLDB Endow.* 14, 9 (May 2021), 1489–1502.