G-CARE: A Framework for Performance Benchmarking of Cardinality Estimation Techniques for Subgraph Matching

Yeonsu Park[†] Seongyun Ko[†] Sourav S Bhowmick[‡]
Kyoungmin Kim[†] Kijae Hong[†] Wook-Shin Han[†§]
POSTECH, Korea[†], NTU, Singapore[‡]
{yspark, syko, kmkim, kjhong, wshan}@dblab.postech.ac.kr[†], assourav@ntu.edu.sg[‡]

ABSTRACT

Despite the crucial role of cardinality estimation in query optimization, there has been no systematic and in-depth study of the existing cardinality estimation techniques for subgraph matching queries. In this paper, for the first time, we present a comprehensive study of the existing cardinality estimation techniques for subgraph matching queries, scaling far beyond the original experiments. We first introduce a novel framework called G-CARE that enables us to realize all existing techniques on top of it and that provides insights on their performance. By using G-CARE, we then reimplement representative cardinality estimation techniques for graph databases as well as relational databases. We next evaluate these techniques w.r.t accuracy on RDF and non-RDF graphs from different domains with subgraph matching queries of various topologies so far considered. Surprisingly, our results reveal that all existing techniques have serious problems in accuracy for various scenarios and datasets. Intriguingly, a simple sampling method based on an *online aggregation* technique designed for *relational* data, consistently outperforms all existing techniques.

ACM Reference Format:
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 (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389702

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389702

1 INTRODUCTION

Subgraph matching based on *subgraph isomorphism* or *graph homomorphism* is an important class of query primitive for querying data graphs. Given a query graph Q and a data graph G, subgraph matching finds all (isomorphic or homomorphic) *embeddings* of Q in G. A recent survey revealed that subgraph matching is among the top-5 most popular graph computation problem among researchers and practitioners [34]. It is also a fundamental building block of several graph query languages such as SPARQL [17], *Cypher*, and more recently G-CORE [4].

Estimating the cardinality (*i.e.,* the number of embeddings) of subgraph matching queries is paramount to determine the accurate execution cost of a query plan [28] since the cardinality of (intermediate) results and input graphs are used as inputs to the query optimizer cost models. In fact, cost-based query optimizers are used by major graph database systems including Neo4j, Oracle PGX, Amazon Neptune, Virtuoso, RDF-3X, and Apache Jena.

Cardinality estimation has been studied extensively in the context of relational databases for several decades and has been deployed widely in commercial DBMS. However, a similar effort for graph data is relatively scant. The majority of these efforts are studied in the context of RDF data. For instance, Neumann *et al.* proposed the RDF-3X system, which has a histogram-based component for cardinality estimation [31]. Subsequently, they extended this technique by exploiting the statistical information of CharacteristicSets [30] (characterizes *subjects* as a set of properties) to estimate cardinality. Chen *et al.* [9] proposed a random walk-based method called Impr to estimate the cardinality of graphlets, which can be utilized to estimate the cardinality

[§]Corresponding author

of a subgraph matching query. Most recently, Stefanoni *et al.* proposed SumRDF [38], which exploits graph summarization to address the cardinality estimation problem.

Despite a decade of research on cardinality estimation for subgraph matching queries, we do not have a thorough understanding of how good existing techniques really are. Some limitations observed in experiments of the existing literature are as follows: 1) Since they are not described under a common framework, it is hard to compare each technique. 2) The reported comparisons of existing literature are incomprehensive. Thus, it is unclear whether later works outperform earlier work. 3) Although topologies, sizes, and cardinalities of queries are addressed as important query features [21, 26], existing literature overlooks in experiments. Except for Impr, the other two techniques do not consider query topologies in depth. CharacteristicSets reports experimental results using only star-shaped queries. SumRDF classifies queries into only four topology groups (linear, star, joined star, complex). In addition, none of them analyze the results of experiments for each range of cardinalities of queries.

Motivated by these problems, we propose a new guideline to evaluate the performance of cardinality estimation techniques for graph databases. In this paper, we present: (a) A common framework for implementing all existing techniques. (b) Thorough performance evaluation of these techniques using real-world and synthetic datasets, focusing on major query features that are relevant to the cardinality estimation problem. Specifically, this paper aims to address the following open-ended questions that are not adequately addressed in the existing literature:

- How accurate are existing cardinality estimation techniques for subgraph matching queries across different real-world and synthetic graph datasets?
- How do these techniques perform for queries with different size and topology (*e.g.,* chain, cycle, flower, etc.)?
- How well do these techniques perform across a wide range of cardinality of queries?
- How scalable are these techniques?

We present a comprehensive study on the cardinality estimation problem for subgraph matching queries. We present g-care (Graph CARdinality Estimation), the *first framework to benchmark cardinality estimation techniques* for subgraph matching queries. Our framework will facilitate benchmarking techniques studied in this paper against future cardinality estimation techniques for subgraph matching queries.

Since the topological constraints of a subgraph matching query can be expressed using sql join conditions, we observe that cardinality estimation techniques for relational queries essentially solve the same problem. This motivates us to expand our study to explore whether the cardinality estimation problem for subgraph matching queries can be addressed by existing techniques not designed specifically for graph-structured data. To this end, we select two state-of-the-art cardinality estimation techniques for relational queries [41, 44] supporting arbitrary join queries and one basic technique as a baseline. Furthermore, we consider a promising online aggregation technique called WanderJoin [23] for join cardinality estimation. With simple modifications to WanderJoin (i.e., introducing a sampling ratio and using `COUNT` aggregation), we can use it in our study.

Although there have been a large number of works on benchmarks or experimental studies for graph data [3, 5, 13, 15, 20, 21, 24, 32, 33, 42], none of them has focused on the cardinality estimation problem. [21] conducted experimental studies on subgraph isomorphism algorithms. [15, 33, 36] presented a benchmark for rdf query processing. [5, 13] provided a benchmark for query processing over social graphs. [24] proposed a benchmark for graph processing with various classes of operations and different tests. [32, 42] conducted experimental studies on graph partitioning strategies. [3] performed an experimental study on distributed graph analytics systems. [20] conducted an experimental study on graph indexing techniques. To the best of our knowledge, we present the first work dedicated to the cardinality estimation problem for subgraph matching.

Our study on seven representative cardinality estimation techniques unveils intriguing and unexpected findings. Although several techniques in the context of relational databases are not convincingly superior to existing cardinality estimation techniques for subgraph matching queries, surprisingly, WanderJoin [23], a technique designed for online aggregation consistently outperforms the other state-of-the-art techniques across a wide variety of measures! That is, a technique not designed to address the cardinality estimation problem for graph data is, in fact, superior to those designed specifically for graph data. In addition, several results of our study contradict the results reported in the original papers.

The rest of the paper is organized as follows. Section 2 introduces the background information necessary for understanding this paper. We implement the state-of-the-art techniques for cardinality estimation for subgraph matching queries in the common framework g-care in Section 3 and highlight their salient features. In Section 4, we expand our research scope by implementing cardinality estimation techniques designed for relational data in g-care. Section 5 presents the experimental setup and datasets for the evaluation of the cardinality estimation techniques. We conduct an exhaustive evaluation of these techniques in Section 6. In Section 7, we conclude this paper.

2 BACKGROUND

We denote a directed labeled graph as $G = (V_G, E_G, L_G)$, where V_G is a set of vertices, E_G is a set of directed edges,

L_G is a mapping function of G for labels of vertices or edges. That is, $L_G(v)$ and $L_G(u, v)$ are the labels of vertex $v \in V_G$ and edge $(u, v) \in E_G$, respectively. We support various types of graph datasets in G-CARE, such as directed labeled graphs, undirected graphs, unlabeled graphs, and Resource Description Framework (RDF). We represent all types of graphs as directed labeled graphs. To this end, for undirected graphs, we represent each undirected edge as two directed edges. For unlabeled graphs, all edges in a data graph are considered to have label zero. Since RDF data consists of a list of triples $(subject, predicate, object)$, we can express it as a directed labeled graph by mapping from *subjects* and *objects* to vertices, and mapping *predicates* to edges with corresponding labels. We denote $\mathcal{L}_G$ to be a list of labels in a graph G. The *size* of a graph G is defined as $|G| = |E_G|$. The *indegree*, $deg_{in}(v)$, and *outdegree*, $deg_{out}(v)$ of v in G is $|\{v'|(v', v) \in E_G\}|$ and $|\{v'|(v, v') \in E_G\}|$, respectively.

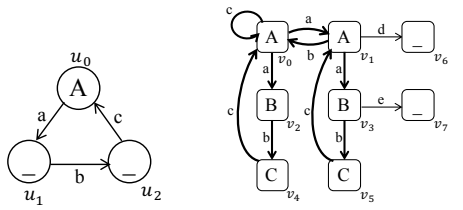
Given a query graph $Q = (V_Q, E_Q, L_Q)$, and a data graph $G = (V_G, E_G, L_G)$, *a graph homomorphism* is a function $M_G : V_Q \rightarrow V_G$, such that, $\forall\ u \in V_Q$, $L_Q(u) \subseteq L_G(M(u))$ and $\forall\ (u_i, u_j) \in E_Q$, $(M(u_i), M(u_j)) \in E_G$ and $L(u_i, u_j) = L_G(M(u_i), M(u_j))$. A graph $G = (V_G, E_G)$ is a *subgraph* of another graph $G' = (V'_G, E'_G)$ if there exists a graph homomorphism from G to G', denoted by $G \subseteq G'$. We may simply say that G' *contains* G. If a query vertex u is unlabeled, $L_Q(u) = \emptyset$, where u can match any data vertices.

If Q is graph homomorphic to G, we call M an *embedding* of Q. Note that there may exist multiple embeddings of Q in G, and we use $\mathcal{E} = \{M^1_G(Q), M^2_G, \ldots, M^n_G\}$ to denote the set of embeddings of Q in G. For example, in Figure 1, there are three graph homomorphic embeddings of Q in G: $M^1_G(Q) = \{(u_0, v_0), (u_1, v_2), (u_2, v_4)\}$, $M^2_G(Q) = \{(u_0, v_1), (u_1, v_3), (u_2, v_5)\}$, and $M^3_G(Q) = \{(u_0, v_0), (u_1, v_1), (u_2, v_0)\}$.

Subgraph matching in property graphs is similar to that in the RDF graphs since a property graph can be represented as an RDF graph [11]. Hence, one can apply the techniques developed for RDF graphs to property graphs. However, query languages for property graph databases such as Cypher support more complex querying features [12] including regular path queries. It would be an interesting future work to estimate cardinalities of such complex queries. Handling the predicates on the properties is related to the cardinality estimation for the selection predicates, which is beyond the scope of this work. We focus on subgraph pattern matching, which is related to join queries in relational databases.

3 CARDINALITY ESTIMATION TECHNIQUES FOR GRAPH DATA

We select three state-of-the-art techniques for our study, namely, CharacteristicSets [30], Impr [9], and SumRDF [38]. We classify the cardinality estimation techniques into two categories: *sampling*-based techniques and *summary*-based techniques. Impr is a sampling-based technique, while CharacteristicSets and SumRDF are summary-based techniques. Except for Impr, all these techniques are designed to estimate the cardinality of graph queries over RDF data. Hence, by transforming each triple in RDF data into a directed labeled edge, we represent the data as a directed labeled graph in our study. Note that we do not consider early efforts such as [19, 31, 39] as they are outperformed by [9, 30, 38].



(a) A query graph Q. (b) A data graph G.

Figure 1: An example of a query and data graph.

We begin by introducing a generic cardinality estimation framework for graph data called G-CARE to implement these techniques. We reimplemented all techniques in G-CARE using C++. Specifically, we made the best efforts to implement CharacteristicSets based on the reference paper. We refer to published codes of Impr and SumRDF implemented in C++ and Java, respectively.

3.1 The G-CARE Framework

Since we must consider both sampling-based techniques and summary-based techniques in a common framework, we introduce a new notion of *target substructure* as follows. If we use a sampling-based technique, a sampling unit with its probability is a target substructure. Otherwise, *i.e.,* if we use a summary-based technique, a matched substructure of a summary is a target substructure. For example, a random walk is a typical target substructure in sampling-based techniques. Suppose that G is summarized into a summary graph S [38]. Then, an embedding of Q in S can be a target substructure.

The G-CARE framework (Algorithm 1) returns an estimated cardinality of Q in G by taking as input a query graph Q, a data graph G, and a sampling ratio p for sampling-based techniques. Note that additional parameters may be required for a specific estimation strategy. G-CARE comprises two key steps, *preparation* for summary structures (Line 1) and *estimation* using target substructures (Line 2-11). In the case of sampling-based techniques, we do not construct summary structures. In estimation, we 1) decompose Q into $(q_1, \ldots, q_m)$ (Line 2), 2) obtain a series of target substructures for q_j and estimate the cardinality of q_j for each target substructure using EstCard, 3) store the estimated cardinality into a vector called $cardVec$, 4) estimate the cardinality of q_j by aggregating over $cardVec$ using aggregation operators such as SUM and AVG (Line 10), and 5) compute the cardinality

of Q by multiplying the cardinalities of subqueries by the selectivity for subqueries. (Line 11). Note that p determines the number of iterations (the number of target substructures).

Algorithm 1: G-CARE framework

```
Input: A query graph Q, a data graph G, a sampling ratio p
Output: Cardinality estimate
1  S ← PrepareSummaryStructure(G, ...);
2  (q_1, ..., q_m) ← DecomposeQuery(Q);
3  subqueryCard ← ∅;
4  for j ← 1 to m do
5      i ← 0;
6      cardVec ← ∅;
7      while (s_i ← GetSubstructure(G, S, q_j, p)) ≠ ∅ do
8          cardVec[i] ← EstCard(q_j, s_i);
9          i ← i + 1;
10     subqueryCard[j] ← AggCard(cardVec);
11 return ∏_{j=1}^{m} subqueryCard[j] · sel(q_1, ..., q_m);
```

We derive the G-CARE framework after exhaustively surveying the existing literature on cardinality estimation. We view the cardinality estimation for subgraph matching as query execution on target substructures instead of the entire data. The target substructures are obtained from sampling or summary structures. G-CARE supports a divide-and-conquer style query execution over the target structures and aggregation of their results, which is general enough to accommodate the existing cardinality estimation techniques. Although the goal of G-CARE is to compare existing techniques fairly in a common framework, users can also enjoy the reduced programming efforts for implementing various cardinality estimation techniques. G-CARE supports various target substructures, storage structures, and a set of basic operations such as pattern matching and random walks.

3.2 Characteristic Sets (C-SET)

CharacteristicSets [30] is a summary-based technique using the summary structures called *characteristic sets*. Each characteristic set counts a specific type of *star-shaped structures* in a data graph. For cardinality estimation, CharacteristicSets decomposes a query into the star-shaped subqueries and estimates the cardinalities of the subqueries using characteristic sets. Finally, it estimates the cardinality of the whole query by an aggregation based on the independence assumption between subqueries.

PrepareSummaryStructure: A characteristic set $cs \in S$ is represented by a set of vertex labels VL and a (possibly empty) set of outgoing (or incoming) edge labels EL. Here, cs stores the number of occurrences of star-shaped structures in the data graph such that each structure has a center vertex with labels VL, and has at least one adjacent outgoing (or incoming) edge with label el for all $el \in EL$. For example, the first column of the first table in Figure 2 shows that the cs represented by vertex label A and outgoing edge labels a and c, appears once (with a center vertex v_0) in the data graph in Figure 1(b). The second and third columns represent the total numbers of occurrences of edges with labels a and c incident to center vertices, respectively. $cs.count$ denotes the number in the first column, while $cs.freq(el)$ denotes the number for edge labels $el \in EL$ among the rest of the columns.

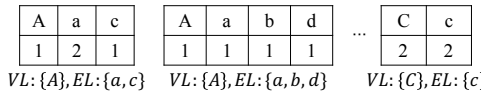


Figure 2: An example of characteristic sets for G.

DecomposeQuery: We decompose Q into subqueries (q_1, ..., q_m) where each subquery corresponds to a star-shaped structure (with a center vertex) or an edge between two unlabeled vertices. For the example query in Figure 1(a), we can decompose it into one star and two edges as in Figure 3. The left star is represented by $VL = \{A\}$ and $EL = \{a\}$ where the center vertex is u_0. The middle and the right ones are edge queries between unlabeled vertices.

GetSubstructure: For each star-shaped subquery q_j (with vertex labels VL_j and edge labels EL_j) from the decomposition, we find a set of characteristic sets $\{(VL, EL)\}$ such that $VL_j \subseteq VL$ and $EL_j \subseteq EL$. Each characteristic set found is returned as s_i. For example, when $j = 1$ (the left subquery in Figure 3), $VL_j = \{A\}$, $EL_j = \{a\}$, and the first and second characteristic sets in Figure 2 are returned as s_1 and s_2. If q_j is an edge query between two unlabeled vertices and has an edge label l, we return a summary (i.e., count) about edges with label l.

EstCard: If s_i is a characteristic set, we estimate its cardinality, i.e., how many star-shaped structures in the data graph correspond to s_i, as follows. $cardVec[i] = s_i.count \cdot \prod_{el \in EL_j} (s_i.freq(el)/s_i.count)$. Otherwise if s_i is a count of edges with a specific label, $cardVec[i] = s_i$.

AggCard: AggCard uses SUM for aggregation.

We have $subqueryCard[j]$ for each subquery q_j (j=1, ..., m). Finally, $sel(q_1, ..., q_m)$ is approximated as the product of pairwise selectivities, i.e., $\prod_{x \neq y} sel(q_x, q_y)$. Here, the selectivity between the two subqueries q_x and q_y, $sel(q_x, q_y)$, is approximated as the product of pairwise selectivities of two incident edges $e_x \in E_{q_x}$ and $e_y \in E_{q_y}$, i.e., $\prod_{e_x \in E_{q_x}, e_y \in E_{q_y}, e_x \cap e_y \neq \emptyset} sel(e_x, e_y)$, where $sel(e_x, e_y)$ is approximated using the basic join selectivity estimation [30].

3.3 SumRDF

SumRDF [38] introduces the *typed* summary graph as a summary where the vertices with the same vertex labels and similar incident edge label distributions are grouped together.

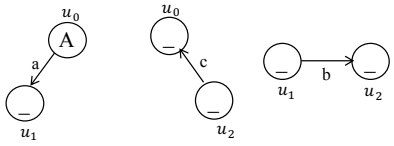
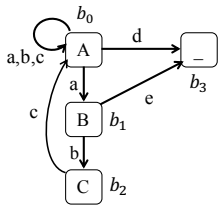


Figure 3: The subqueries decomposed from Q.

Based on the *possible world* semantics, it estimates the cardinality as the expected cardinality over all possible data graphs summarized into the same summary graph.

PrepareSummaryStructure: We define the summary graph S of G as follows. Let μ be a many-to-one mapping from a data vertex v to a summary vertex b (called a bucket). Here, each data vertex v has a *type* that consists of the labels of v and the label distribution of its incident edges. If two data vertices have similar types (i.e., same vertex labels and similar edge label distributions), they are mapped to the same bucket. The similarity of the label distribution can be controlled by a user, and interested readers can refer to [38]. As the number of distinct vertex and edge labels increases, we observe that a summary graph can become very large, leading to significantly slow estimation time. Therefore, we extend the summarization algorithm, so that types with different vertex labels can be merged as well, when the size of the summary graph is larger than a user-defined threshold (i.e., 3% of the data graph size).

A summary vertex (or edge) maintains the weight as the number of data vertices (or edges) that are mapped to the same summary vertex (or edge). Formally, $w(b) = |\{v \mid \mu(v) = b\}|$, and $w(b, b', l) = |\{(v, v') \mid \mu(v) = b, \mu(v') = b', l = L_G(b, b')\}|$. Figure 4 shows a summary graph of the data graph in Figure 1(b). v_0 and v_1 are mapped to the same bucket b_0, and v_2 and v_3 are mapped to b_1. Thus, $w(b_0) = 2$, $w(b_1) = 2$, and $w(b_0, b_1, a) = 2$; two edges (v_0, v_2) and (v_1, v_3) with label a are grouped into (b_0, b_1).



Vertices	Weights
b_0	$w(b_0) = 2$
b_1	$w(b_1) = 2$
b_2	$w(b_2) = 2$
b_3	$w(b_3) = 2$

Edges	Weights
(b_0, b_0)	$w(b_0, b_0, a) = 1$ $w(b_0, b_0, b) = 1$ $w(b_0, b_0, c) = 1$
(b_0, b_1)	$w(b_0, b_1, a) = 2$
(b_0, b_3)	$w(b_0, b_3, d) = 1$
(b_1, b_2)	$w(b_1, b_2, b) = 2$
(b_1, b_3)	$w(b_1, b_3, e) = 1$
(b_2, b_0)	$w(b_2, b_0, c) = 2$

Figure 4: A summary graph in SumRDF.

GetSubstructure: We return an embedding of Q in S (rather than G) as the target substructure s_i by using graph homomorphism. In Figure 4(a), we can find two embeddings of Q in Figure 1(a): s_0={u_0→b_0, u_1→b_1, u_2→b_2} and s_1={ u_0→b_0, u_1→b_0, u_2→b_0}.

EstCard: We estimate the cardinality of Q (in G) using s_i. Since multiple data vertices can be mapped to the same bucket in s_i, we can expand s_i into multiple embeddings (from query vertices to data vertices). For example, s_0 in our running example can be expanded into $w(b_0) \cdot w(b_1) \cdot w(b_2) = 8$ embeddings. Let m be any of these eight embeddings (e.g., {u_0→v_0, u_1→v_3, u_2→v_4}). Using the possible world semantics, we can calculate $P(m)$, the probability that a graph contains the embedding m is selected among all graphs that are summarized into s_0. In our running example, there are $\binom{w(b_0)w(b_1)}{w(b_0,b_1,a)} \cdot \binom{w(b_1)w(b_2)}{w(b_1,b_2,b)} \cdot \binom{w(b_2)w(b_0)}{w(b_2,b_0,c)} = \binom{4}{2}^3 = 216$ possible worlds. Among the 216 worlds, $\binom{w(b_0)w(b_1)-1}{w(b_0,b_1,a)-1} \cdot \binom{w(b_1)w(b_2)-1}{w(b_1,b_2,b)-1} \cdot \binom{w(b_2)w(b_0)-1}{w(b_2,b_0,c)-1} = \binom{3}{1}^3 = 27$ worlds contain m. Therefore, $P(m) = 27/216$. Considering that there are eight embeddings, the expected cardinality of Q is $8 \cdot 27/216 = 1$. This is returned as $cardVec[0]$.

AggCard: AggCard uses SUM for aggregation.

3.4 IMPR

Impr [9] is a sampling-based technique designed to estimate the cardinality of k-node graphlets for $k \in \{3, 4, 5\}$. We extend Impr to count the number of embeddings instead of the number of subgraphs. For example, given a triangle query Q and a subgraph of the same shape $\mathcal{G}$, Impr finds one subgraph in $\mathcal{G}$, but we modify it to find three embeddings of Q in $\mathcal{G}$ based on the graph homomorphism.

Impr performs random walks on a data graph and generates *visible* subgraphs (which we define in EstCard) from the walked vertices and their neighbors. It counts the number of embeddings in these subgraphs that match Q. The weighted sum of the counts is returned as an estimate.

GetSubstructure: The first vertex in the walk is chosen from the stationary probability $d(v)/2|E_G|$ where $d(v)$ denotes the degree of v [16]. After selecting the first vertex, our version of Impr follows random walks considering edge labels in the query. The original algorithm does not consider query label information, where the transition probability from a vertex v to one of its neighbors v' is $1/d(v)$. Each random walk of $k-1$ consecutive vertices is returned as s_i.

EstCard: Let V_{s_i} and E_{s_i} be the set of vertices and edges that appear in s_i, respectively. EstCard generates a *visible* subgraph $g_{s_i} = (V, E, L)$ of G where V consists of vertices in V_{s_i} and their neighbors N_{s_i}, and E consists of edges in E_{s_i} and edges between V_{s_i} and N_{s_i}. L is the same as L_G. Then, it computes $f(s_i)$, the number of embeddings that match Q where each embedding consists of all vertices in V_{s_i} and a vertex in N_{s_i}. Consider Figure 1. $k = 3$ since Q has three vertices. If $s_i = \langle v_0, v_1 \rangle$, $V = V_G \setminus \{v_7\}$ and $E = E_G \setminus \{(v_2, v_4), (v_3, v_5), (v_3, v_7)\}$. Therefore, we find one embedding $\{(u_0, v_0), (u_1, v_1), (u_2, v_0)\}$ that matches Q. Hence, $f(s_i) = 1$.

$cardVec[i]$ is then computed as $W(s_i)f(s_i)$; $W(s_i)$ is chosen so that the estimation is unbiased. Specifically, $W(s_i) =$

$\frac{1}{\beta(Q)} \cdot \frac{|A(s_i)|}{\Sigma_{s \in A(s_i)} \pi(s)}$, where $A(s_i)$ is the set of possible random walks that can visit the same vertices as s_i (with different orders of vertices in s_i), and $\pi(s)$ is the stationary probability of performing a random walk s [16]. $\beta(Q)$ is a normalization factor, equal to the number of possible walks consisting of $|V_Q| - 1$ vertices in Q.

AggCard: AggCard uses AVG for aggregation.

4 CARDINALITY ESTIMATION TECHNIQUES FOR RELATIONAL DATA

Since subgraph queries can be expressed as join queries in relational databases, the problem of estimating their cardinality can be addressed by estimating the cardinality of corresponding join queries in relational databases. Specifically, an edge in a data graph can be represented by a pair of a source vertex and destination vertex (or a vertex) which can be stored as a tuple in a table constructed for the corresponding edge label (or vertex label) [1]. Consequently, a subgraph query can be posed as a join query on the underlying database. For example, in Figure 1(b), a data graph G is stored in 8 relations: $R_A(v), R_B(v), R_C(v)$ for vertex labels, and $R_a(src, dst)$, $R_b(src, dst)$, $R_c(src, dst)$, $R_d(src, dst)$, $R_e(src, dst)$ for edge labels. In Figure 1(a), Q can be posed as a join query on the underlying database.

We select three state-of-the-art cardinality estimation techniques for relational queries and one online aggregation technique [8, 23, 41, 44]. We describe how these techniques are implemented in the G-CARE framework for evaluation. Note that we do not consider the relational-based estimation techniques [10, 29] in our study because their strategies cannot be applied to arbitrary join queries.

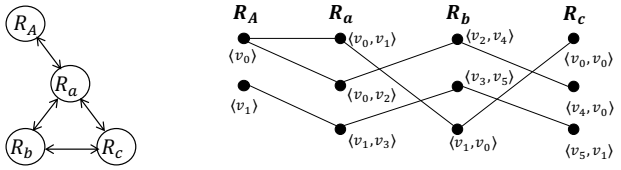
4.1 Correlated Sampling (CS)

CorrelatedSampling [41] is a sampling-based technique. Different from the independent sampling (i.e., Bernoulli Sampling), CorrelatedSampling samples tuples considering the correlation between relations by hashing.

GetSubstructure: Suppose that Q involves n relations (R_1, ..., R_n), and the sampling ratio is p. We create a sample s_0 as a list of relations $\langle \mathcal{S}_1, ..., \mathcal{S}_n \rangle$, where $\mathcal{S}_i$ contains the sampled tuples from R_i.

For each join attribute a of Q, we define an independent, uniform hash function h_a that maps the values for a into the range $[0, 1]$. Let A_{R_i} denote the set of join attributes of the relation R_i. Then, we sample $p|R_i|$ tuples from R_i as follows. For each tuple t in R_i, we sample t iff $h_a(t[a]) < p^{1/|A_{R_i}|}, \forall a \in A_{R_i}$. The probability of sampling t would be $p^{1/|A_{R_i}|} \times ... \times p^{1/|A_{R_i}|}$ ($|A_{R_i}|$ times) $= p$ since all hash functions used are independent.

EstCard: It calculates $cardVec[0]$ as $|\mathcal{S}_1 \bowtie \mathcal{S}_2 \bowtie ... \bowtie \mathcal{S}_n| / P(s_0)$ where $P(s_0)$ denotes the probability that a joined tuple t in $R_1 \bowtie ... \bowtie R_n$ is in $\mathcal{S}_1 \bowtie ... \bowtie \mathcal{S}_n$. Let t_i denote the tuple from $R_i \in t$. $P(s_0)$ is then the probability of sampling t_i from R_i for all i. Using the join attributes $\{a\}$, $P(s_0)$ can be expressed as $P(\bigwedge_a \bigwedge_{A_{R_i} \ni a} h_a(t_i[a]) < p^{1/|A_{R_i}|}) = P(\bigwedge_a h_a(t_i[a]) < \min_{A_{R_i} \ni a} p^{1/|A_{R_i}|}) = \prod_a \min_{A_{R_i} \ni a} p^{1/|A_{R_i}|}$ since all h_as are independent.



(a) A join query graph. (b) A join data graph.

Figure 5: A join query graph Q' for Q and join data graph G' for the join order $\langle R_A, R_a, R_b, R_c \rangle$.

4.2 Wander Join (WJ)

WanderJoin [23] is a random walk-based approach originally designed for online aggregation. In online aggregation, the estimates for aggregation results are updated over time until a certain stop condition is met. We extend it for cardinality estimation by introducing a sampling ratio as the stop condition and using `COUNT` aggregation.

WanderJoin performs random walks such that the walks contributing to the cardinality are more likely to be sampled. Here, a sample is a list of tuples $\langle t_1, ..., t_n \rangle$ and contributes to the cardinality iff $t_1 \bowtie ... \bowtie t_n$ matches Q.

WanderJoin represents the query Q as a join query graph denoted by Q' where each relation is represented as a vertex, and each join condition between a pair of relations is represented as an edge (Figure 5(a)). WanderJoin performs random walks over a join data graph G' where each tuple is regarded as a vertex and each join condition is regarded as an edge. Thus, WanderJoin can be directly implemented by performing random walks over a data graph. Figure 5(b) shows an example join data graph G'.

GetSubstructure: It returns a sample s_i as a list of tuples and its probability $P(s_i)$. A walk order $\langle R_1, ..., R_n \rangle$ is a join order defined over a spanning tree of Q', where only the edges in the spanning tree are walked. Here, the random walk consists of "walk" or "jump." For example, in Figure 5(a), while $\langle R_A, R_a, R_b, R_c \rangle$ consists of only walks, $\langle R_c, R_a, R_A, R_b \rangle$ includes a jump from R_A to R_b.

Before we explain how to choose the walk order, we explain the sampling process given an order $\langle R_1, ..., R_n \rangle$. We sample a tuple t_i from R_i, where t_1 is randomly sampled from R_1, and each subsequent t_i ($i > 1$) is randomly sampled from $t_{p(i)} \bowtie R_i$. Here, $t_{p(i)}$ is the tuple sampled from $R_{p(i)}$ where $R_{p(i)}$ is the relation joinable with R_i and appears before R_i (i.e., $p(i) < i$). Since the order is defined over a spanning tree, there is only one such $R_{p(i)}$. After sampling t_i for every $i = 1, ..., n$, we check whether the join conditions of Q' are all

met by the sampled tuples, i.e., whether $t_1 \bowtie ... \bowtie t_n$ matches the join query Q'. If this holds, the sample $\langle t_1, ..., t_n \rangle$ is *valid*. If it does not hold or it becomes impossible to sample t_i for some $i \leq n$ (when $t_{p(i)} \bowtie R_i = \emptyset$), the sample is *invalid*.

Now we explain how we choose the walk order. We first enumerate all possible walk orders and select each order in a round-robin fashion. Using each order, we sample s and check if s is valid. If valid, WanderJoin increases the *counter* of the current order. When the counter of an order reaches the threshold τ (100 by default), we choose the walk order with the smallest variance (of the estimates from the samples) among the orders with the counter $\geq \tau/2$. WanderJoin expects that this heuristic can generate more valid samples and low-variance estimates than others. Finally, GetSubstructure returns a sample s_i, generated at the initial stage of choosing the walk order or after the walk order is chosen.
EstCard: It estimates the cardinality of Q using *Horvitz-Thompson estimator* [18] which is an unbiased estimator with *inverse probability weighting*. If s_i is valid, $cardVec[i]$ becomes the inverse probability of sampling s_i, i.e., $1/P(s)$. Here, $P(s) = P(\langle t_1, ..., t_n \rangle)$ is calculated as $\frac{1}{|R_1|} \prod_{i=2}^{n} \frac{1}{|t_{p(i)} \bowtie R_i|}$. If s_i is invalid, $cardVec[i] = 0$.
AggCard: AggCard uses AVG for aggregation.

4.3 Join Sampling with Upper Bounds (JSUB)

Zhao *et al.* [44] have proposed a framework for random sampling over joins which can be combined with join cardinality estimation techniques. Their objective is to obtain uniform independent samples efficiently for machine learning, which is not our target. However, similar to BoundSketch [8], they internally use upper bounds for intermediate result sizes which can be adopted for cardinality estimation.

We exploit their framework and generate a sampling-based method for cardinality estimation over joins, namely JSUB. It estimates an upper bound of the cardinality of a join query Q. It first extracts an maximal acyclic subquery q_1 from Q. Then, it estimates $|q_1|$, the cardinality of q_1 using a sampling method similar to that of WanderJoin. Note that when Q is cyclic, $|q_1|$ becomes larger than the cardinality of Q. Finally, JSUB multiplies the estimated $|q_1|$ by $M(q_1)$ to estimate an upper bound of the cardinality of the whole query Q. Here, $M(q_1)$ is the maximum number of the join results between a tuple that matches q_1 and the residual relations. We refer to the remaining relations of Q not in q_1 as residuals, e.g., R_c is a residual if $o = \langle R_A, R_a, R_b \rangle$. In our context, we trivially set $M(q_1) = 1$, as in [44].
DecomposeQuery: We choose q_1 as follows. For each possible q_1 and its join order o, we use WanderJoin to estimate $|q_1|$. Let $e(q_1, o)$ denote the estimate. We choose q_1 and o such that $\arg\min_{q_1, o} e(q_1, o)$. This is a slight modification over [44] which uses a predefined join order. If we fail to obtain a valid sample after trying every possible q_1 and o, we return 0 as the estimate.
GetSubstructure: Let $o = \langle R_1, R_2, ..., R_n \rangle$ denote the join order. Then, JSUB samples a tuple t from the first relation R_1, and returns t and $P(t)$ as s_i, where $P(t)$ is the probability of sampling t from R_1.
EstCard: It returns $W(t)/P(t)$, where $W(t)$ is an upper bound of $w(t)$, which is the cardinality of the join result of t with the rest of the relations (i.e., $w(t) = |t \bowtie R_2 \bowtie ... \bowtie R_n|$). Since we use $W(t)/P(t)$ as an estimate of $|q_1|$ and $W(t) \geq w(t)$, we can over-estimate $|q_1|$ if $W(t) > w(t)$, and only $W(t) = w(t)$ gives an unbiased estimate of $|q_1|$ [18]. Therefore, we use $W(t) = w(t)$ in the same way as the sampling method, Exact Weight (EW), in [44]. To reduce the computation time, JSUB uses dynamic programming, as in [44], and computes $W(t)$ only if t is sampled.

Since $W(t) = w(t)$, AggCard returns an unbiased estimate of $|q_1|$. Thus, if Q is cyclic and $Q \neq q_1$, AggCard estimates an upper bound of the cardinality of Q. One might modify $W(t)$ in EstCard to capture the exact intermediate result sizes considering the residual relations; however, it inevitably includes the overhead of subgraph matching which can be costly. Here, we choose a more efficient way by considering the trade-off of accuracy versus efficiency.
AggCard: AggCard uses AVG for aggregation.

4.4 Bound Sketch (BS)

BoundSketch [8] is a summary-based technique which estimates an upper bound . It utilizes the bounding formulas in [2], where each relation R in Q may appear in a bounding formula as a *count* term ($c_R = |R|$) or a maximum *degree* term (d_R^a for some attribute a in R). To define d_R^a, we first define the *degree* $d_R^a(v)$ as the frequency of a value v on an attribute a of a relation R, i.e., $d_R^a(v) = |\{t \in R \mid t[a] = v\}|$. Then, the maximum degree d_R^a becomes $\max_v d_R^a(v)$.

The constraint for the bounding formulas is that every attribute of Q must be covered by an *appearing* relation. Here, if R appears as a count term c_R, all attributes of R are covered, and if R appears as a maximum degree term d_R^a, all attributes of R except a are covered, assuming that a is already covered by another relation. For our example query $Q(u_0, u_1, u_2) = R_A(u_0) \bowtie R_a(u_0, u_1) \bowtie R_b(u_1, u_2) \bowtie R_c(u_2, u_0)$ in Figure 1(a), a bounding formula can be $c_{R_A} d_{R_a}^{u_0} d_{R_b}^{u_1}$. Appearing relations are R_A, R_a, and R_b, and u_0 is covered by R_A, u_1 by R_a, and u_2 by R_b. $c_{R_A} c_{R_b}$ is another possible bounding formula, where u_0 is covered by R_A, and u_1 and u_2 are covered by R_b.
PrepareSummaryStructure: In a bounding formula, a count term c_R represents the Cartesian product over all tuples in R. A maximum degree term d_R^a represents the maximum number of tuples that can be joined in R using the join attribute a. Therefore, multiplying all these terms can lead to an extremely loose upper bound as the number of relations

increases. Instead, BoundSketch partitions each relation and uses counts and maximum degrees defined on the partitioned relations in order to get a tighter upper bound.

In order to build a summary S, we hash each attribute value in Q to the range $[1...M]$ using a hash function H. Let A_Q and A_R be the set of attributes of Q and a relation R, respectively. We use $m \in [1...M]^{|A_Q|}$ as an index for partitions. That is, $R^{(m)}$ represents a partition of R that contains tuples whose hash values are m. Formally, $R^{(m)} = \{t \in R \mid H(t[a]) = m[a], \forall a \in A_Q \cap A_R\}$. For each partition $R^{(m)}$, we calculate the count and maximum degree as we did for the whole relation R. Therefore, for each relation R, we obtain $M^{|A_Q|}$ counts and $M^{|A_Q|} \times |A_Q|$ maximum degrees. This takes a single pass over R, and the summary S consists of the counts and maximum degrees over all relations.

BoundSketch chooses M based on a *budget*; larger budget increases M and thus tightens the upper bound with a trade-off of summarization time. In our evaluation, we use 4096 for the default number of budget.

Although sketches can be built on-demand during cardinality estimation, we observe that their build time is significantly larger than the query processing time. Therefore, we populate the sketches of all relations before the query processing and use them at run-time for experiments. Regarding queries with selection predicates, we do not build sketches on-demand and use the pre-built ones without the selection predicates, since the on-demand build time takes up to 79.6 times longer than the query processing time of RDF-3X for the LUBM benchmark in our experimemts.

GetSubstructure: It returns a bounding formula bf_i and all counts and maximum degrees that are related to bf_i. For example, if $bf_i = c_{R_A} d_{R_a}^{u_0} d_{R_b}^{u_1}$, related counts are $c_{R_A^m}$, and related maximum degrees are $d_{R_a^m}^{u_0}$ and $d_{R_b^m}^{u_1}$ for $m \in [1...M]^{|A_Q|}$.

EstCard: It returns the summation of bf_i instantiated with counts and maximum degrees of partitions in s_i. For example, if $bf_i = c_{R_A} d_{R_a}^{u_0} d_{R_b}^{u_1}$, $cardVec[i] = \sum_{m \in [1...M]^{|A_Q|}} c_{R_A^m} d_{R_a^m}^{u_0} d_{R_b^m}^{u_1}$. Each term inside the summation represents an upper bound of the cardinality of $Q^{(m)}$, the join result of Q over the partitions $R^{(m)}$. Since $R^{(m)}$s are disjoint, $Q^{(m)}$ is a partition of the join result of Q which is also disjoint. Thus, the summation of the upper bounds of the cardinality of $Q^{(m)}$ becomes an upper bound of the cardinality of Q.

AggCard: AggCard uses MIN for aggregation, selecting the smallest upper bound among the possible bf_i.

5 EXPERIMENTAL SETUP

We describe the setup for evaluating the cardinality estimation techniques for subgraph matching queries. In order to select the relevant features for evaluation, we have reviewed 54 papers on cardinality estimation. All papers consider a couple of features from the following four: dataset, query topology, query size, and query result size (or query selectivity). We conduct experiments varying all of the features. Table 1 gives an overview. For experiments, we use a machine with 2.10GHz Intel Xeon E7-8870 v4 processors and 1.5TB memory.

Our G-CARE framework is publicly available as an open-source test suite with codes, data, and queries and is easily extensible for new cardinality estimation techniques for subgraph matching queries*.

Table 1: Parameters used in the experiments.

Dataset	RDF: LUBM, YAGO, DBpedia Non-RDF: AIDS, Human
Query Topology	Chain, Star, Tree, Cycle, Clique, Petal, Flower, Graph
Query Result Size	$(0, 10]$, $(10, 10^2]$, $(10^2, 10^3]$, $(10^3, 10^4]$, $(10^4, 10^5]$, $(10^5, 10^6]$
Query Size	3, 6, 9, 12
Sampling Ratio	3, 1, 0.3, 0.1, 0.03, 0.01 [%]

5.1 Evaluation Measure

We measure the accuracy and efficiency of the cardinality estimation techniques. For accuracy, we use *q-error* [28] which is widely used to measure accuracy of cardinality estimation techniques. The *q-error* quantifies the ratio between the estimated cardinality, $\hat{c}$, and the true cardinality, c, and computed as $q\text{-}error = max(max(1, c)/max(1, \hat{c}), max(1, \hat{c})/max(1, c))$.

We run each query 30 times. We report the average and standard deviation for LUBM and the 5%, 25%, 50%, 75%, and 95% percentiles of the q-errors for other datasets. Note that since the *q-error* alone does not differentiate the under-/overestimation, we represent it explicitly on the y-axis of the result figures. For efficiency, we report the elapsed times of the off-line preprocessing for building summary structures and the on-line per-query processing for estimation.

5.2 Datasets

We use five synthetic and real-world datasets for evaluation: LUBM [15], YAGO [40], DBpedia [6], AIDS (used in [37]), and Human (used in [43]). Note that these datasets are used by several existing cardinality estimation techniques for subgraph matching queries. Specifically, LUBM is a standard RDF benchmark which can be used to generate synthetic RDF data. We populate the LUBM graph data with a scaling factor of 80. YAGO and DBpedia are real-world RDF datasets. The AIDS dataset consists of 10,000 small- or medium-sized real graphs. Since it contains multiple data graphs, we aggregate the number of embeddings from all graphs as estimates. Lastly, the Human dataset contains a graph representing the protein-protein interaction network of humans. Table 2 shows the summary of the statistical information of these datasets.

*https://github.com/yspark-dblab/gcare

Table 2: Statistics of datasets.

	LUBM	YAGO	DBpedia	AIDS	Human
# of graphs	1	1	1	10K	1
# of vertices	2.6M	12.8M	66.9M	254K	4.7K
# of edges	12.3M	15.8M	225M	548K	86K
Avg. degree	9.35	2.47	6.75	4.31	36.92
Max. degree	0.9M	0.25M	7.3M	22	771
# of distinct v. labels	35	188K	244	50	89
# of distinct e. labels	35	91	39.6K	4	0
Max triples per pred.	2.3M	8.3K	98.7M	270K	-
Min triples per pred.	1	2	1	2.6K	-

5.3 Generation of Test Queries

To the best of our knowledge, there is no publicly-available benchmark queryset for the cardinality estimation problem for graph data. For LUBM, among the 14 queries, we use 6 queries (Q2, Q4, Q7, Q8, Q9, and Q12). We exclude the simple queries with at most two triple patterns in their SPARQL statements due to space constraints.

For the remaining datasets, we generate test queries by varying query topology, query size, and the result size. Based on the frequently used query topology in real-world graph queries [7], we generate 8 classes of queries: chain, star, tree, cycle, clique, petal, flower, and graph. A *chain* is a sequence of vertices $\langle u_0, u_1, \ldots, u_n \rangle$ such that u_{i-1} and u_i are connected for $i \in [1, n]$. A *star* is a sequence of vertices $\langle u_0, u_1, \ldots, u_n \rangle$ such that u_i is connected to a source vertex u_0 for $i \in [1, n]$. Tree queries are arbitrary queries not containing any cycle. Cycle, clique, flower, petal, and graphs are cyclic queries. A *cycle* is a sequence of vertices $\langle u_0, u_1, \ldots, u_n \rangle$ where u_{i-1} and u_i are connected for $i \in [1, n]$ and u_n is connected to u_0. A *clique* is a complete graph. A *petal* consists of a source vertex, a destination vertex and a set of at least two vertex-disjoint paths between them. A *flower* consists of a source vertex with three types of attachments: chains, trees, and petals. A *graph* query includes arbitrary cyclic and acyclic queries.

Table 1 shows the parameters used in the experiments. Given a query topology, query size, and result size, we generate queries by traversing the schema graph randomly for each data graph matching a target topology. For YAGO, AIDS, and Human datasets, we generate 1366, 780, and 49 queries, respectively. For each dataset, we report the results by grouping them by the query result size, query size, and query topology, respectively. For sampling-based techniques, we use the sample ratio of 3% as the default. We also conduct experiments varying the sample ratio as in [41]: {0.01, 0.03, 0.1, 0.3, 1, 3%}. The timeout is set to 5 minutes.

6 PERFORMANCE COMPARISON

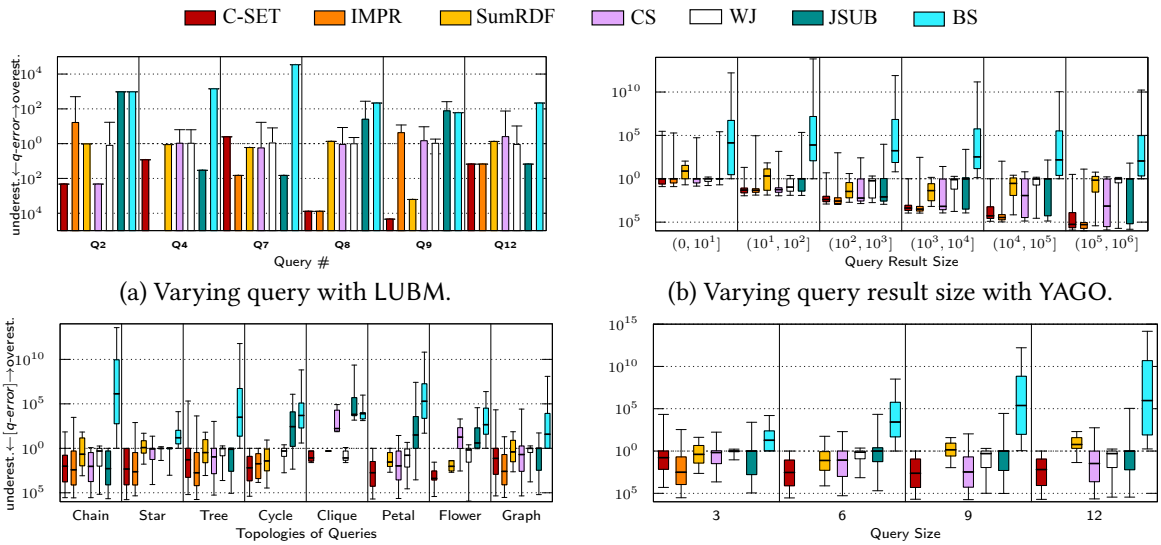
6.1 Accuracy on RDF Graphs

6.1.1 ***Evaluation Using the LUBM Benchmark.*** Figure 6 (a) shows the accuracy test results with LUBM. Surprisingly, WanderJoin outperforms all other techniques with its q-error values close to 1 although it is designed for online aggregation for the relational data.

We compare WanderJoin against the sampling-based techniques. CorrelatedSampling suffers from the underestimation problem for Q2. This is because it fails to sample the tuples which actually contribute to the join results. Impr underestimates the cardinalities for Q7, Q8, and Q12. Its sampling method fails to find the random walks satisfying the join conditions. For example, it finds no single random walk satisfying the join conditions in Q7, Q8, and Q12. Impr cannot process Q4 due to its restriction on the query topology. JSUB tends to overestimate the cardinalities for Q2, Q8, and Q9. This is expected because it estimates an upper bound of the cardinality. However, for Q4, Q7, and Q12, it shows significant underestimation due to the sampling failure at DecomposeQuery. WanderJoin avoids the underestimation problem by sampling tuples which are more likely to satisfy the join conditions. It does not suffer from the overestimation problem because it does not use the upper bounds.

We compare WanderJoin against the summary-based techniques. Among the summary-based techniques, SumRDF shows the highest accuracy. However, compared to WanderJoin, it underestimates the cardinality for Q9. The main cause is the uniformity assumption used for computing the expected number of embeddings. For Q9, we observe that the number of possible worlds, $|\{g_S\}|$, is orders of magnitude larger than that for the other queries. CharacteristicSets shows a more severe underestimation problem than SumRDF except for $Q7$. We observe large errors in its estimation of the query selectivity due to the independence assumption between the decomposed queries, which does not hold. BoundSketch consistently overestimates because it computes the upper bounds of cardinality using the bounding formulas with the sketches. For the summary-based techniques, we observe that the loss of information due to summarization is unavoidable, and the assumptions made to estimate the cardinality lead to large errors. WanderJoin does not create a summary structure in advance but computes the probabilistic statistics via sampled tuples satisfying the join conditions in a query.

6.1.2 ***Varying Query Result Size.*** We test using YAGO. Figure 6(b) shows the accuracy test results. Surprisingly, WanderJoin outperforms all other tested techniques. It provides accurate estimation results for both the small and large number of query results. As the query result size increases, most of the tested techniques yield larger q-errors showing the underestimation problem. In many of such cases, the estimated cardinality values are close to 0, which leads to larger q-error values for queries with larger result sizes. CharacteristicSets suffers from the underestimation problem due to the independence assumption as we observed in



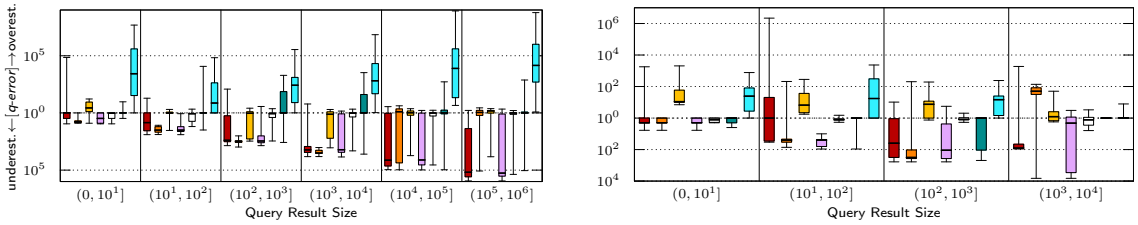
(a) Varying query with LUBM.

(b) Varying query result size with YAGO.

(c) Varying query topology with YAGO.

(d) Varying query size with YAGO.

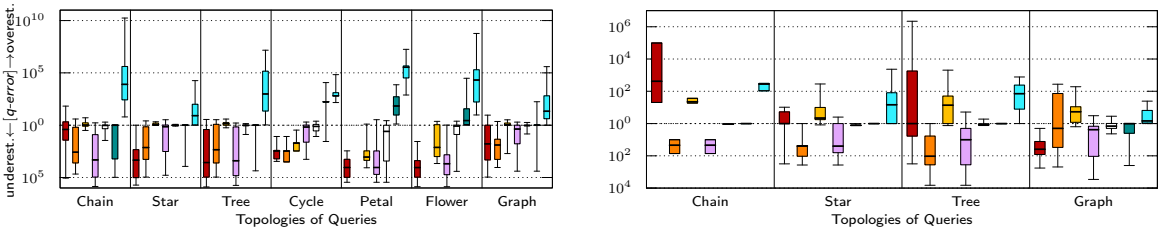
Figure 6: The accuracy tests for RDF datasets.



(a) AIDS.

(b) Human.

Figure 7: Varying query result size with Non-RDF datasets.



(a) AIDS.

(b) Human.

Figure 8: Varying query topology with Non-RDF datasets.

Section 6.1.1. While CharacteristicSets was evaluated using only star-shaped queries in the original work, we use queries of various shapes. Impr and CorrelatedSampling underestimate the cardinality because of the sampling failure, as we already observed. Overall, JSUB outputs accurate estimation results for the 50% percentile. However, it shows a high variance on the q-errors and underestimation problem due to the sampling failure. BoundSketch does not show clear correlations on its accuracy with the result size.

6.1.3 ***Varying Query Topology.*** Note that the minimum query size is six for clique, petal, and flower. Impr cannot process such queries because it can process 3,4,5-node queries only. Figure 6(c) reports the accuracy test results. Overall, WanderJoin outperforms all other tested techniques. It provides accurate estimation results on various query topologies. However, it exhibits large errors especially on chain, tree, petal, and flower due to the sampling failure. We measure the sampling failure rates for queries of sizes larger than 10.

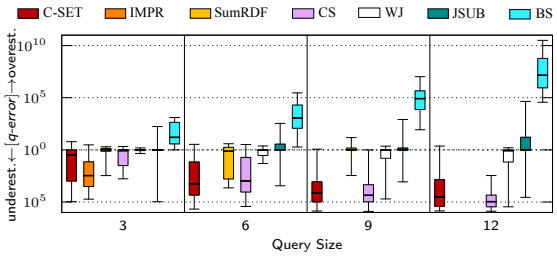


Figure 9: Varying query size with AIDS.

While the successful sampling rates for star and cycle queries are 27.7% and 16.8%, respectively, those for chain, tree, and petal queries are only 0.16%, 8.6%, and 0.04%, respectively. Despite that, WanderJoin outperforms all techniques.

6.1.4 ***Varying Query Size.*** Figure 6(d) shows the accuracy results. WanderJoin outperforms all other tested techniques. It provides much more accurate estimation results on both small and large query sizes than the other techniques. CharacteristicSets, Impr, and CorrelatedSampling suffer from the underestimation problem due to the reasons previously explained in Section 6.1.1. BoundSketch shows clear trends of increasing errors for larger queries. This is because the upper bound computation involves the product of counts or degrees of relations in a query. There are more summation terms in computing $cardVec$ for larger queries.

6.2 Accuracy on Non-RDF Graphs

We test using non-rdf graphs such as AIDS and Human datasets. We use 780 and 49 generated queries for AIDS and Human, respectively. Note that the schema graphs of AIDS and Human are much smaller compared to that of YAGO. As a result, we are only able to generate a restricted set of queries. We highlight the key observations compared to the results with rdf graphs. An important difference between rdf and non-rdf graphs is distributions of labels. For example, YAGO has 91 distinct edge labels, while Human and AIDS have 0 and 4 distinct edge labels, respectively.

6.2.1 ***Varying Query Result Size.*** Figures 7(a) and (b) show the accuracy test results for AIDS and Human, respectively. WanderJoin outperforms all other tested techniques on non-rdf graphs as well. The q-error values of 95%, 75%, 50%, and 25% percentiles are close to 1. CharacteristicSets tends to underestimate as the result size increases due to the independence assumption. SumRDF overestimates for Human compared to AIDS. This is because the number of distinct edge labels of Human is 0, therefore, when merging two summary vertices, the edge weights between them are all aggregated leading to the overestimation.

6.2.2 ***Varying Query Topology.*** Figures 8(a) and (b) show the accuracy test results for AIDS and Human, respectively. WanderJoin outperforms all other tested techniques. Impr provides more accurate estimation at AIDS and Human than at YAGO. This is because AIDS and Human have fewer labels; random walks of Impr can generate more visible graphs that contain embeddings of a query, resulting less sampling failure. JSUB overestimates for the cycle, petal, and flower queries, since an acyclic query q_1 is extracted from each such query, and the upper bound of the cardinality of the query is estimated as $|q_1|$. No cyclic query is generated for Human.

6.2.3 ***Varying Query Size.*** Figure 9 shows the accuracies for AIDS. Note that, in Human, we cannot generate a sufficient number of queries with the different number of embeddings for sizes 6, 9, and 12 due to its restriction of labels. Hence, we omit its results in our study.

Again, WanderJoin is the best performing one among all. Impr cannot process queries whose sizes are greater than five. SumRDF fails to process queries with 12 edges due to the timeout. CharacteristicSets and CorrelatedSampling underestimate for larger queries due to the independence assumption and sampling failure, respectively. BoundSketch shows the clear trends of increasing errors for larger queries.

6.3 Varying Sampling Ratio

The performance of the sampling-based techniques is affected by the sampling ratio. We conduct a sensitivity analysis by varying the sampling ratio in {0.01, 0.03, 0.1, 0.3, 1, 3}% for the sampling-based techniques with YAGO and AIDS. We omit the result figures due to space constraints.

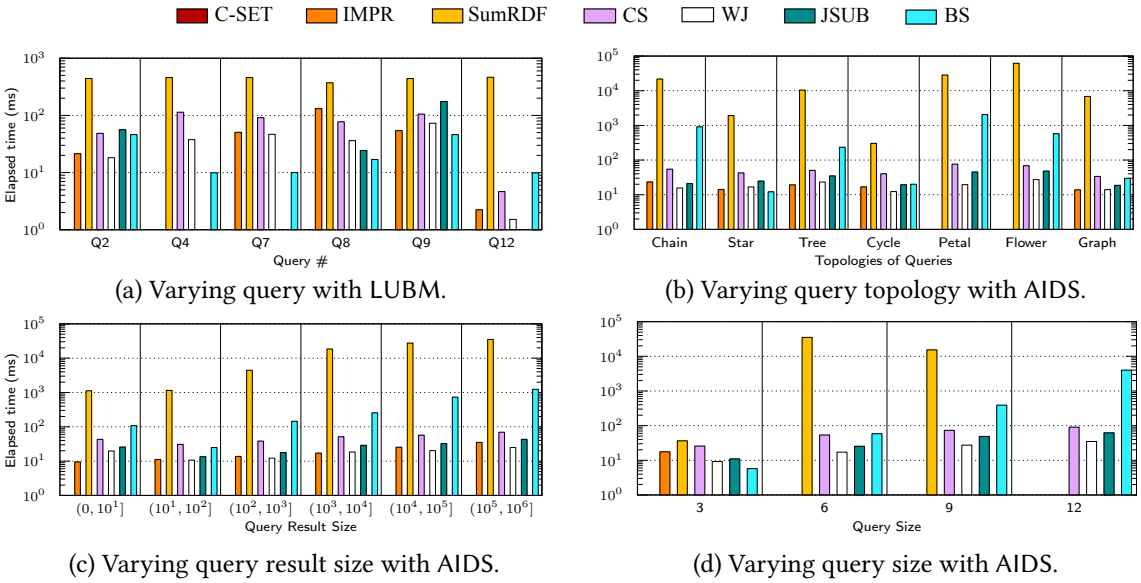
Overall, WanderJoin is the best performing technique. It shows its robust performance even with very small sampling ratios such as 0.01% compared to others. For all tested sampling ratios, CorrelatedSampling and Impr consistently underestimate the cardinalities, and JSUB shows significantly large variance on its q-errors.

6.4 Efficiency

We evaluate the efficiency with LUBM and AIDS using a small capacity server machine (Intel Xeon E5-2450 CPU and 64 GB RAM) which is sufficient for loading the entire graph and summary structure in memory. We report both the elapsed times for the off-line preprocessing and on-line per-query processing.

For the off-line preprocessing, we measure the elapsed time for constructing the summary structure in memory once the input graph data is loaded. For LUBM, CharacteristicSets, SumRDF, and BoundSketch spend 0.96, 12.26, and 160.8 seconds, respectively. For AIDS, CharacteristicSets, SumRDF, and BoundSketch spend 0.07, 0.64, and 3.93 seconds, respectively.

Figure 10 shows the efficiency test results for the on-line per-query processing. We highlight the key observations. Among the summary-based techniques, SumRDF is the slowest one. This is because SumRDF spends most of the time on GetSubstructure and EstCard procedures. It searches for embeddings in a summary graph S by checking whether



(a) Varying query with LUBM.

(b) Varying query topology with AIDS.

(c) Varying query result size with AIDS.

(d) Varying query size with AIDS.

Figure 10: The efficiency tests for LUBM and AIDS datasets.

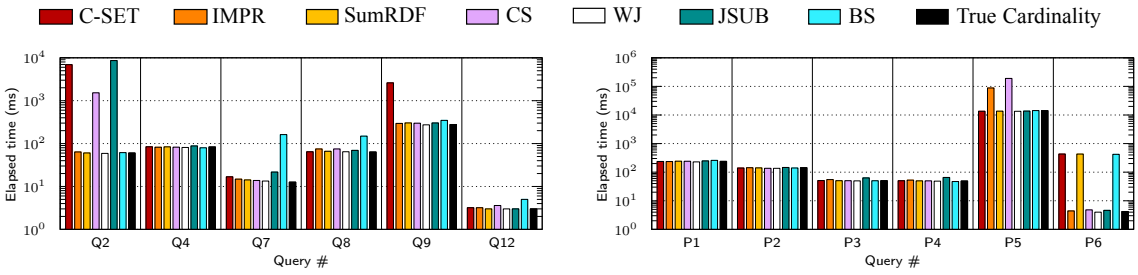
each edge in Q can be matched with edges in S, which takes $O(|E_S|^{|E_Q|})$ time. Its processing time increases drastically for larger result sizes and larger queries as shown in Figure 10(c) and (d). Among the sampling-based techniques, CorrelatedSampling is the overall slowest one because it has to process the join query Q over the samples. Therefore, its cost increases as the result size increases as shown in Figure 10(c). The other sampling techniques are based on the random-walks which consume mostly linear time with regard to the number of edges in Q as shown in Figure 10(d).

6.5 Impact on Plan Quality

An interesting research question is *"Given the same set of queries and various estimators, do the different estimates impact the query plans significantly?"* There is an excellent paper [22] about this issue in the context of relational query optimization. The following experiments confirm the same conclusions as in [22]: 1) different cardinalities could impact query plans significantly; 2) a plan based on the true cardinality could be marginally worse than a plan based on a bad estimate due to inaccuracy of the cost model; 3) accurate cardinality estimation should be the first priority to research.

We chose RDF-3X, a popular open-source RDF database management system, which uses merge join as much as possible. First, we made the following changes in the original code. 1) G-CARE feeds cardinality estimates to RDF-3X so that it generates plans based on those cardinalities. 2) The cost model of RDF-3X has coefficient numbers to accommodate CPU and disk costs in a single model. We perform calibration experiments to gather accurate coefficient numbers [14], 3) In RDF-3X, when both inputs are sorted, it always uses merge join. Otherwise, it always uses hash join. This often leads to suboptimal plans, especially when one input is sorted, but the other input is unsorted. In this case, if the unsorted input is small, adding a sort operator on top of the unsorted input can lead to merge join instead of hash join. This is a standard plan generation strategy in commercial relational DBMSs. We implemented this strategy in RDF-3X.

Note that using the q-error is important since it is a *theoretical upper bound* for the plan quality [22, 28]. In practice, in order to understand the relationship between cardinality estimates and plan quality, we use the concept of *validity range* [27]. Consider a join plan P having two child nodes. The validity range for a child node is a range on the number of rows flowing through, such that if the range is not violated at runtime, we can guarantee that P is optimal with respect to the cost model. Thus, suppose that we have a plan P with true cardinalities. Depending on the query graph and data graph, P has different validity ranges for its intermediate results. If a validity range is wide, a significantly inaccurate estimate could be within this range so we can guarantee that this bad estimate still generates the same plan as P. If a validity range is narrow, a slightly inaccurate estimate can lead to a different suboptimal plan. Therefore, if we have accurate estimates, the optimizer is likely to generate the best plan with true cardinality. A query optimizer requires estimates for the subqueries of an input query. In order to analyze the relationship with q-error and plan quality, we need to analyze the q-errors for all subqueries of Q. However, since the q-error for the query itself typically contains propagated errors from the subqueries of Q, we can analyze plan quality using the q-error for Q.



(a) LUBM. (b) DBpedia.

Figure 11: Query processing time in RDF-3X. TC represents the elapsed times of plans with true cardinality.

We executed two sets of queries: LUBM queries and DBpedia queries. Here, the DBpedia queries are real ones extracted from query logs of various endpoints [35]. We present six representative queries due to space limitations, since the other extracted queries have similar performance trends.

Figure 11(a) shows the elapsed times for the LUBM queries. For $Q2$, underestimates/overestimates lead to performance degradation. One exception is BoundSketch. BoundSketch shows accurate estimates of subgraphs of size two, although it shows overestimation for $Q2$ itself. CharacteristicSets, Impr, CorrelatedSampling, and JSUB show significantly inaccurate estimates for the subgraphs of size two or three, and thus, their execution times are slow. $Q4$ is a star-shaped query and thus, the optimizer in RDF-3X generates a robust query plan. That is, the validity ranges for the optimal plan for $Q4$ are wide. For $Q7$ and $Q8$, underestimates/overestimates lead to performance degradation. For $Q7$, overestimation by BoundSketch makes RDF-3X choose hash join over sort-merge join which requires sorting. However, the actual cardinality of intermediate results to sort is very small and, thus, the generated plan with hash join is slower than the plan with sort-merge join. For $Q8$, although CharacteristicSets shows underestimates, the optimizer generates a good plan using merge join with a sort operator at the root of the plan. In terms of cost, this plan is worse than the optimal plan. However, the merge join does not scan all inputs if one input is completely consumed. Thus, the execution time of CharacteristicSets is similar to that of the optimal plan. For $Q9$, CharacteristicSets shows the worst query performance since the q-errors for the subgraphs of $Q9$ are very large. For $Q12$, although plans generated by all estimators are different, their query processing times are all small. Overall, WanderJoin and SumRDF perform the best in terms of execution time.

Figure 11(b) shows the elapsed times for the DBpedia queries. $P1$ and $P2$ are star-shaped queries. Thus, RDF-3X generates a robust query plan regardless of cardinalities. We already observed the same phenomenon for $Q4$ in LUBM. If we consider more diverse plans, such as nested loop join, bad estimates can easily lead to suboptimal plans. $P3$ and $P4$ are graph-shaped. In this case, WanderJoin outperforms JSUB by 31% in terms of query execution time. $P5$ is tree-shaped. Impr performs worse than the other estimators except for CorrelatedSampling since it consistently generates large q-errors for the subqueries. In CorrelatedSampling, the q-errors for the subqueries fluctuate around one. This significant variance generates the worst plan. For $P6$, in terms of q-error, WanderJoin, Impr, JSUB, and CorrelatedSampling generate the optimal plan. Note that the other estimators including SumRDF are about 105 times slower. BoundSketch overestimates and generates a worse plan, which we have already observed for $Q7$ in LUBM. Although CharacteristicSets achieves good accuracy in terms of q-error, the validity ranges for the optimal plan are very narrow. CharacteristicSets generates a significantly worse plan. This indicates that accurate cardinality estimation should play a pivotal role in subgraph query optimization.

6.6 Summary

We make the following observations. First, in the original paper, the high accuracy of CharacteristicSets was demonstrated using only star-shaped queries. However, our analysis revealed that it suffers from significant underestimation due to the independence assumption. Second, Impr is originally designed for unlabeled graphs. Hence, it cannot be generalized to process directed, labeled graphs (such as RDF graphs) due to the underestimation problem. Third, the original paper of SumRDF shows high accuracy on LUBM, which is consistent with our study. However, it shows inaccurate estimation results for other datasets. Furthermore, it fails to handle large queries due to prohibitive computation cost.

Our experimental study is the first work which evaluates and analyzes the state-of-the-art cardinality estimation techniques for graph and relational data. Our investigation reveals unexpected results. We observe that for both RDF and non-RDF datasets, WanderJoin, which is designed for online aggregation, consistently outperforms state-of-the-art cardinality estimation techniques for subgraph matching queries in terms of accuracy across a variety of parameters.

Table 3: A summarized comparison of graph- and relational-based techniques (✅: accurate, ❌: inaccurate).

Technique		Key Results						
		LUBM Queryset	Query Features of Test Queries					
			# of Embeddings		Size		Topology	
			$\sim 10^3$	$10^3 \sim$	3 ~ 6	9 ~ 12	Tree	Graph
Graph-based	CSet	❌	❌	❌	❌	❌	❌	❌
	Impr	❌	❌	❌	❌	❌	❌	❌
	SumRDF	✅	❌	❌	❌	❌	❌	❌
Relational-based	CS	❌	❌	❌	❌	❌	❌	❌
	WJ	✅	✅	✅	✅	✅	✅	✅
	JSUB	❌	❌	❌	❌	❌	❌	❌
	BS	❌	❌	❌	❌	❌	❌	❌

Why does WanderJoin perform well? The key task for estimating cardinality for subgraph matching is to obtain high quality of samples. WanderJoin selects the best join order and samples tuples which satisfy the join conditions. By randomly selecting the start vertex for random walks for every sampling, it obtains s_i from the entire data graph. Different from the summary-based techniques, WanderJoin does not rely on the summary which inherently has some loss of information. Instead, it obtains the necessary probabilistic statistics based on sampling after the query is given.

It is worth noting that not all cardinality estimation techniques for relational data perform well for graph cardinality estimation. CorrelatedSampling has difficulty in finding embeddings as the number of joins increases. This is due to the low quality of samples, which does not consider correlations between sets of attributes to be joined. JSUB does not perform well as far as accuracy is concerned. It shows high variance on the q-errors and the underestimation problem.

In terms of efficiency, WanderJoin and CharacteristicSets are much faster than the other methods. We believe there is still room to improve efficiency, which is an interesting future work. We also confirm that different estimates could impact plan quality significantly. However, depending on the validity ranges, bad estimates could generate good plans. Nevertheless, accurate cardinality estimation should play a pivotal role in subgraph query optimization.

7 CONCLUSIONS

We present G-CARE to facilitate an in-depth and systematic experimental evaluation of state-of-the-art cardinality estimation techniques for graph data. We conducted exhaustive experiments on three graph-based, three relational-based, and one online aggregation techniques with a variety of features of queries on both RDF and non-RDF graphs.

Table 3 summarizes the observed performance in our experiments. In contrast to the results reported in the original papers, we conclude that none of the three cardinality estimation techniques for subgraph matching queries is clearly the best for all datasets and all scenarios.

In contrast to the graph-based techniques, the relational-based techniques are able to process almost all test queries successfully across datasets. However, CorrelatedSampling is inaccurate for all scenarios except on LUBM dataset. JSUB is inaccurate in almost all cases. Surprisingly, WanderJoin, which is designed for the online aggregation problem, shows superior performance consistently in terms of accuracy and outperforms graph-based techniques. Given that all existing efforts on cardinality estimation for graph data do not compare the approaches with cardinality estimation techniques for relational data, we believe that the results of our study are insightful to the graph data management community.

The superiority of WanderJoin w.r.t graph-based techniques opens up the issue of whether RDBMS techniques can be leveraged to address the cardinality estimation problem for graph data. Note that a recent survey [34] revealed the prevalence of relational-based databases among practitioners for managing and processing graph data. However, recent studies have shown that native graph systems are superior to relational-based systems for graph traversal and querying [25], which leads us to two interesting avenues of research: (a) Is it possible to design cardinality estimation techniques for subgraph matching queries which integrate the benefits of WanderJoin with native graph-based techniques? (b) Is a hybrid system which leverages native graph stores for query processing but utilizes a relational framework for cardinality estimation a viable strategy? Our G-CARE framework can be utilized to benchmark performances of the future cardinality estimation techniques for subgraph matching queries.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2017R1A2B3007116) and Institute of Information communications Technology Planning Evaluation(IITP) grant funded by the Korea government(MSIT) (No. 2018-0-01398, Development of a Conversational, Self-tuning DBMS).

REFERENCES

[1] Daniel J Abadi, Adam Marcus, Samuel R Madden, et al. 2007. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*. 411–422.

[2] 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*. 429–444.

[3] Khaled Ammar and M Tamer Özsu. 2018. Experimental analysis of distributed graph systems. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1151–1164.

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, et al. 2018. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data*. 1421–1432.

[5] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, et al. 2013. LinkBench: a database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1185–1196.

[6] Sören Auer, Christian Bizer, Georgi Kobilarov, et al. 2007. Dbpedia: A nucleus for a web of open data. In *The semantic web*. Springer, 722–735.

[7] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An analytical study of large SPARQL query logs. *Proceedings of the VLDB Endowment* 11, 2 (2017), 149–161.

[8] 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*. 18–35.

[9] Xiaowei Chen and John CS Lui. 2016. Mining Graphlet Counts in Online Social Networks. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 71–80.

[10] Yu Chen and Ke Yi. 2017. Two-level sampling for join size estimation. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 759–774.

[11] Souripriya Das, Jagannathan Srinivasan, Matthew Perry, et al. 2014. A Tale of Two Graphs: Property Graphs as RDF in Oracle.. In *EDBT*. 762–773.

[12] Alin Deutsch and Yannis Papakonstantinou. 2018. Graph data models, query languages and programming paradigms. *Proceedings of the VLDB Endowment* 11, 12 (2018), 2106–2109.

[13] Orri Erling, Alex Averbuch, Josep Larriba-Pey, et al. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[14] Georges Gardarin, Fei Sha, and Zhao-Hui Tang. 1996. Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System.. In *VLDB*, Vol. 96. Citeseer, 3–6.

[15] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics* 3, 2-3 (2005), 158–182.

[16] Olle Häggström et al. 2002. *Finite Markov chains and algorithmic applications*. Vol. 52. Cambridge University Press.

[17] Stephen Harris and Nigel Shadbolt. 2005. SPARQL query processing with conventional relational database systems. In *International Conference on Web Information Systems Engineering*. Springer, 235–244.

[18] Daniel G Horvitz and Donovan J Thompson. 1952. A generalization of sampling without replacement from a finite universe. *Journal of the American statistical Association* 47, 260 (1952), 663–685.

[19] Hai Huang and Chengfei Liu. 2011. Estimating selectivity for joined RDF triple patterns. In *Proceedings of the 20th ACM international conference on Information and knowledge management*. 1435–1444.

[20] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. 2015. Performance and scalability of indexed subgraph query processing methods. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1566–1577.

[21] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, et al. 2012. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the VLDB Endowment* 6, 2 (2012), 133–144.

[22] Viktor Leis, Andrey Gubichev, Atanas Mirchev, et al. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[23] Feifei Li, Bin Wu, Ke Yi, et al. 2016. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*. 615–629.

[24] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond macrobenchmarks: microbenchmark-based graph database evaluation. *Proceedings of the VLDB Endowment* 12, 4 (2018), 390–403.

[25] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond macrobenchmarks: microbenchmark-based graph database evaluation. *Proceedings of the VLDB Endowment* 12, 4 (2018), 390–403.

[26] Guy Lohman. 2014. Is query optimization a âĂIJsolvedâĂİ problem. In *Proc. Workshop on Database Query Optimization*, Vol. 13. Oregon Graduate Center Comp. Sci. Tech. Rep.

[27] Volker Markl, Vijayshankar Raman, David Simmen, et al. 2004. Robust query processing through progressive optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 659–670.

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

[29] Magnus Müller, Guido Moerkotte, and Oliver Kolb. 2018. Improved selectivity estimation by combining knowledge from sampling and synopses. *Proceedings of the VLDB Endowment* 11, 9 (2018), 1016–1028.

[30] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 984–994.

[31] Thomas Neumann and Gerhard Weikum. 2008. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment* 1, 1 (2008), 647–659.

[32] Anil Pacaci and M Tamer Özsu. 2019. Experimental Analysis of Streaming Algorithms for Graph Partitioning. In *Proceedings of the 2019 International Conference on Management of Data*. 1375–1392.

[33] Shi Qiao and Z Meral Özsoyoğlu. 2015. Rbench: Application-specific RDF benchmarking. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1825–1838.

[34] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, et al. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment* 11, 4 (2017), 420–431.

[35] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. 2015. Feasible: A feature-based sparql benchmark generation framework. In *International Semantic Web Conference*. Springer, 52–69.

[36] Michael Schmidt, Thomas Hornung, Georg Lausen, et al. 2009. SPˆ 2Bench: a SPARQL performance benchmark. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 222–233.

[37] Haichuan Shang, Ying Zhang, Xuemin Lin, et al. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment* 1, 1 (2008), 364–375.

[38] 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*. 1043–1052.

[39] Markus Stocker, Andy Seaborne, Abraham Bernstein, et al. 2008. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*. 595–604.

[40] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2008. Yago: A large ontology from wikipedia and wordnet. *Journal of Web Semantics* 6, 3 (2008), 203–217.

[41] David Vengerov, Andre Cavalheiro Menck, Mohamed Zait, et al. 2015. Join size estimation subject to filter conditions. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1530–1541.

[42] Shiv Verma, Luke M Leslie, Yosub Shin, et al. 2017. An experimental comparison of partitioning strategies in distributed graph processing. *Proceedings of the VLDB Endowment* 10, 5 (2017), 493–504.

[43] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. 192–203.

[44] Zhuoyue Zhao, Robert Christensen, Feifei Li, et al. 2018. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data*. 1525–1539.