Accurate Sampling-Based Cardinality Estimation for Complex Graph Queries

PAN HU, School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China
BORIS MOTIK, Department of Computer Science, Oxford University, Oxford, United Kingdom of Great Britain and Northern Ireland

Accurately estimating the cardinality (i.e., the number of answers) of complex queries plays a central role in database systems. This problem is particularly difficult in graph databases, where queries often involve a large number of joins and self-joins. Recently, Park et al. [55] surveyed seven state-of-the-art cardinality estimation approaches for graph queries. The results of their extensive empirical evaluation show that a sampling method based on the *WanderJoin* online aggregation algorithm [47] consistently offers superior accuracy.

We extended the framework by Park et al. [55] with three additional datasets and repeated their experiments. Our results showed that WanderJoin is indeed very accurate, but it can often take a large number of samples and thus be very slow. Moreover, when queries are complex and data distributions are skewed, it often fails to find valid samples and estimates the cardinality as zero. Finally, complex graph queries often go beyond simple graph matching and involve arbitrary nesting of relational operators such as disjunction, difference, and duplicate elimination. Neither of the methods considered by Park et al. [55] is applicable to such queries.

In this article, we present a novel approach for estimating the cardinality of complex graph queries. Our approach is inspired by WanderJoin, but, unlike all approaches known to us, it can process complex queries with arbitrary operator nesting. Our estimator is strongly consistent, meaning that the average of repeated estimates converges with probability one to the actual cardinality. We present optimisations of the basic algorithm that aim to reduce the chance of producing zero estimates and improve accuracy. We show empirically that our approach is both accurate and quick on complex queries and large datasets. Finally, we discuss how to integrate our approach into a simple dynamic programming query planner, and we confirm empirically that our planner produces high-quality plans that can significantly reduce end-to-end query evaluation times.

CCS Concepts: • **Information systems** → **Database query processing;**

Additional Key Words and Phrases: Cardinality estimation, conjunctive queries, sampling, query planning

ACM Reference Format:
Pan Hu and Boris Motik. 2024. Accurate Sampling-Based Cardinality Estimation for Complex Graph Queries. *ACM Trans. Datab. Syst.* 49, 3, Article 12 (September 2024), 46 pages. https://doi.org/10.1145/3689209

This work was funded by the NSFC grant 62206169 and the EPSRC grant AnaLOG (EP/P025943/1).
Authors' Contact Information: Pan Hu, School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, China; e-mail: pan.hu@sjtu.edu.cn; Boris Motik, Department of Computer Science, Oxford University, Oxford, United Kingdom of Great Britain and Northern Ireland; e-mail: boris.motik@cs.ox.ac.uk.

ACM 0362-5915/2024/09-ART12
https://doi.org/10.1145/3689209

1 Introduction

Estimating query cardinality (i.e., the number of answers) plays a central role in database systems. Query planners use cardinality estimates to determine the cost of candidate query plans, and estimation accuracy can significantly influence the resulting plan quality [46]. At the same time, thousands of candidate plans can be considered during planning, so, to be useful, estimation must be orders of magnitude faster than query evaluation. Thus, striking the right balance between speed and accuracy is key to designing effective cardinality estimation algorithms.

Background. Numerous approaches summarise the data in the database using a *synopsis*—a data structure that can estimate the cardinality of certain types of queries. One-dimensional synopses, such as one-dimensional histograms [38, 59] and wavelets [50], summarise one attribute of one relation, so they can process queries involving one selection over a single relation. Multidimensional synopses, such as multidimensional histograms [2, 12, 24, 58], multidimensional wavelets [14, 21], discrete cosine transforms [45], and kernel methods [23, 32], summarise several attributes of one relation, so they can process several selections over a single relation. Finally, schema-level synopses, such as join synopses [3], graphical models [22, 69], TuG synopses [62], statistical views [11, 19, 66], Bayesian networks [68], and correlated sample synopses [73], summarise results of joins of several relations. Queries whose cardinality cannot be estimated using the available synopses are typically broken into subqueries that can be estimated, and partial estimates are combined using ad hoc assumptions [10, 20]: the *independence assumption* means that each selection or join affects the query answers independently; the *preservation assumption* means that each attribute value of any joined relation is present in the join result; and the *containment assumption* means that, for each pair of joined attributes, all values of one attribute are contained in the other attribute. Chen et al. [16] recently presented a systematic analysis of the space of such assumptions. However, these assumptions usually do not hold in practice, which often leads to significant estimation errors.

Several recent approaches train an ML model that can be understood as an advanced schema-level synopsis capturing statistical properties of the queries and/or the data. ML-based approaches can be broadly divided into two groups. In the first group, training is performed on examples of queries and corresponding cardinalities [43, 51, 56]. Producing training data thus typically requires computing the exact cardinality of a large number of queries, which can be cumbersome. In the second group, the objective is to learn an approximation of the distribution of tuples in either one [31, 72] or several [35, 71] database relations. Such approaches have proved effective in practice, but they typically rely on a *join schema*—an explicit list of joins that are expected in a subsequent query workload. Queries with joins not covered by the join schema are broken into parts that can be estimated independently, and the results are combined using ad hoc assumptions.

Finally, query cardinality can be estimated by sampling the data in the database. The objective is usually to produce *unbiased* estimates, which means that the estimate expectation is equal to the query cardinality. The average of independent unbiased estimates converges to the actual cardinality as the number of estimates increases, so the number of samples provides a natural way to control the efficiency vs. accuracy tradeoff. Lipton and Naughton [48] presented a general framework for unbiased sampling-based cardinality estimation, and they showed how to choose the number of samples for certain query classes. Query cardinality can also be estimated using *online aggregation* algorithms, which can compute unbiased estimates of aggregation results [26, 27, 33]. *WanderJoin* [47] is a recent online aggregation algorithm that typically offers superior performance to earlier approaches. Most sampling-based algorithms do not depend on a join schema and provide unbiased estimates for queries with arbitrary combinations of (self-)joins.

Query cardinality estimation is also used in *graph databases*—systems that organise data as labelled graphs [6]. Graph queries typically enumerate all embeddings of a *graph pattern* into

the database graph, which corresponds to evaluating select–rename–join queries. Graph patterns often include joins over dozens of edges, and they often express connectivity patterns (e.g., "friends of friends") that frequently involve self-joins. This makes cardinality estimation in graph databases particularly challenging: joins are frequently not covered by schema-level synopses (e.g., References [68, 71, 73]) so ad hoc assumptions are frequently needed; furthermore, the incurred errors are known to compound exponentially with the number of joins [39].

To systematically compare existing approaches to cardinality estimation in graph databases, Park et al. [55] recently presented the G-CARE framework consisting of five datasets, an extensive set of accompanying queries, and an implementation of seven known approaches to cardinality estimation in graph databases. Three methods (Characteristic Sets [52], SumRDF [63], and Impr [17]) were specifically developed for graph data, and four (Correlated Sampling [70], WanderJoin [47], JSUB [76], and Bounded Sketch [13]) were adapted from relational databases. After an extensive comparison of the accuracy and efficiency of these approaches, Park et al. [55] identified WanderJoin as consistently outperforming the other approaches.

Limitations of the Existing Approaches. We repeated the experiments by Park et al. [55] using a much larger version of the LUBM [25] dataset, the WatDiv [4] benchmark, and a graph version of DBLP.[1] Our results confirm that WanderJoin is significantly more accurate than the other six approaches, but they also revealed several drawbacks. Specifically, WanderJoin can be quite slow: the median estimation time for a single run of the WanderJoin implementation by Park et al. [55] was 5.1 s, 160 ms, and 1.5 s on our three new datasets, respectively, which is too slow to be effective in query planning. Moreover, our investigation showed that, when datasets are large and queries are complex and selective, WanderJoin often produces zero estimates. In such situations, query optimisers typically fall back to heuristics that ignore attribute correlations, which can lead to large estimation errors and consequently result in poor query plans [34].

Furthermore, graph queries often go beyond graph pattern matching and involve arbitrary nesting of graph matching and operators such as union, projection, and duplicate elimination.

Example 1.1. The following is a simplified version of a SPARQL [30] query we encountered while applying a leading RDF data management system to a product configuration use case.

```
SELECT ?axis ?motor WHERE {
  { SELECT DISTINCT ?axis ?motor WHERE {
      ?axis :compatible-mounting-kit ?mounting .
      { SELECT DISTINCT ?mounting WHERE { ?mounting :compatible-motor ?motor } }
    } }
  { ?axis rdf:type ?cat . ?cat rdfs:subClassOf ?cls .
    VALUES ?cls { :AxisFamily :AxisConsturction ... } }
  UNION
  { ?axis rdf:type :Axis }
}
```

The aim of the nested `DISTINCT` operators was to reduce the number of bindings for the `?mounting` and `?motor` variables and thus ensure efficient query evaluation. However, the RDF system we used struggled to produce an efficient query plan because of its inability to accurately estimate the cardinality of the `UNION` and `DISTINCT` subqueries, as well as of their join. ◁

Approaches to cardinality estimation we are aware of typically handle only select–rename–join queries—that is, queries over joins of several relations with equality or range conditions on relation attributes. This presents a significant obstacle to the planning of complex graph queries.

[1] https://blog.dblp.org/2022/03/02/dblp-in-rdf/

Our Contribution. We present a novel WanderJoin-inspired method for estimating cardinality of complex graph queries. Unlike WanderJoin, our method is applicable to queries involving arbitrary nesting of join, union, difference, projection, filter, bind, and duplicate elimination operators with bag semantics. Its estimates are strongly consistent in the sense that the average of repeated invocations converges with probability one to the actual cardinality; moreover, for queries without duplicate elimination, estimates are unbiased. We show that our approach can intuitively be understood as "sampling the loops" of query evaluation with *sideways information passing* [40, 53, 74].

We also present several optimisations that aim to improve estimation accuracy without incurring significant overheads. In particular, we show that we can partition the sample space without affecting the statistical properties of the estimator. Furthermore, we discuss ways to identify conjunct orders that are more likely to produce accurate estimates.

We then discuss how our cardinality estimation approach can be integrated into a simple dynamic programming query planning algorithm. We discuss the challenges that zero estimates pose for query planning, and we present ways to overcome these issues without significant overheads.

Finally, we present the results of an extensive empirical evaluation of our results. We show that our approach produces highly accurate estimates of graph pattern queries on the extended G-CARE framework, but using a fraction of time of the WanderJoin variant by Park et al. [55]. We also show that our approach can also efficiently and accurately estimate the cardinality of complex queries, such as the one described in Example 1.1. We also conduct end-to-end experiments and show that accurate cardinality estimations allow our query planning approach to speed up overall query evaluation by several orders of magnitude on complex queries. Finally, we compare our approach with NeuroCard [71], a prominent approach based on deep learning. We show that our algorithms produce estimates of comparable accuracy but in a fraction of the time; moreover, our approaches do not require a join schema and thus can be used without anticipating the query workload in advance, and they can also process cyclic queries. Thus, our results significantly improve the state-of-the-art of sampling-based cardinality estimation methods.

All code and datasets used in our experiments, as well as the detailed results of our experiments, are available for download online [37].

2 Preliminaries

In this section, we formally define the graph data model and the corresponding query language, we introduce the notion of an estimator, and we define the problem we consider in this article.

2.1 Data Model and Query Language

Angles et al. [6] recently classified the data models and query languages used in graph databases into two main groups. In the first group, data is modelled as *edge-labelled graphs*—that is, only edges can be labelled. *Resource Description Framework (RDF)* [44] is an example of such a model. An RDF graph consists of finitely many *triples* of the form $\langle s, p, o \rangle$, each representing a p-labelled edge from the *subject* vertex s to the *object* vertex o. SPARQL [30] is the standard language for querying RDF databases. The second group consists of *property graphs*, where each vertex and edge is associated with a unique identifier, zero or more types, and a set of key–value pairs. Standardisation of query languages for property graphs is still ongoing, but Cypher [65] and Gremlin [67] are commonly used in practice. *Graph pattern matching* is a key feature of virtually all graph query languages. A *graph pattern* is a graph in which some parts (e.g., a vertex, an edge, or a label) are replaced by variables. The objective of graph pattern matching is to find all combinations of variable values for which the graph pattern becomes a subset of the data graph. In addition, graph query languages

often provide algebraic operations such as union, difference, or optionals, as well as *regular path queries*, which select pairs of vertices connected by paths matching a regular expression.

The syntaxes of SPARQL and Cypher are complex and cumbersome. When formalising the semantics of SPARQL, Pérez et al. [57] introduced an algebraic query language that captures the essence of SPARQL, but is more suited to formal presentation. We follow their approach and use a slight variation of their query language in this article. Moreover, the principles we discuss are independent from the details of the data model so, for simplicity, we assume that graph data is represented *relationally*. Such a representation can easily capture both edge-labelled graphs and property graphs, so our results can easily be applied in both kinds of databases.

We use the notion of a *multiset* M over a domain set D, which is a function that assigns to each element $d \in D$ a nonnegative number $M(d)$ of occurrences of d in M. We use double braces to distinguish sets from multisets; for example, $\{\!\{1, 1, 2\}\!\}$ is a multiset containing number 1 twice and number 2 once. For M_1 and M_2 multisets over the same domain D, multisets $M_1 \cap M_2$, $M_1 \cup M_2$, and $M_1 \setminus M_2$ are defined as $(M_1 \cap M_2)(d) = \min(M_1(d), M_2(d))$, $(M_1 \cup M_2)(d) = M_1(d) + M_2(d)$, and $(M_1 \setminus M_2)(d) = \max(0, M_1(d) - M_2(d))$ for each $d \in D$. Finally, when these operations are applied to a multiset and a set, we implicitly treat the set as a multiset in which all elements occur once.

Data Model. A *database schema* consists of finitely many relations, each associated with a nonnegative integer *arity*. A *database instance* I over a database schema maps each n-ary relation R to a *relation instance* $I(R)$, which is a finite set of n-tuples of constants. If desired, one can require constants occurring at different tuple position to be drawn from appropriate domains (e.g., strings or integers), but such constraints do not play any role in our work. We assume that relation instances are sets (i.e., they do not contain repeated tuples), because such models are commonly used in graph databases; however, our results can be easily extended to multiset relation instances. Sometimes, it is convenient to represent $\langle c_1, \ldots, c_n \rangle \in I(R)$ as a *fact* $R(c_1, \ldots, c_n)$ that combines the n-tuple of constants with the relation name, and to view a database instance I as a finite set of facts $R(c_1, \ldots, c_n)$ for each relation R and each n-tuple $\langle c_1, \ldots, c_n \rangle \in I(R)$.

Query Language. Our queries are constructed using countably infinite, disjoint sets of *constants* and *variables*. A *term* is a constant or a variable. Unless stated otherwise, we use possibly subscripted lowercase letters from the front (e.g., $a, b, c, \ldots$), the middle ($s, t, \ldots$), and the end ($x, y, z, \ldots$) of the alphabet for constants, terms, and variables, respectively. A *builtin expression* is constructed in the usual way from terms and *builtin functions*; for example, $x + 2$ is a builtin expression where + is a builtin function. An *atom* is an expression of the form $R(t_1, \ldots, t_n)$, where R is an n-ary relation and $t_1, \ldots, t_n$ are terms. A *query* is defined inductively as shown in the first column of Table 1, where A is an atom, Q_1 and Q_2 are queries, X is a set of variables, and E is a builtin expression. The table also defines a function $\mathsf{v}(Q)$ that assigns to each query Q a set of *free variables*. Each query must satisfy the constraint from the third column. We sometimes abbreviate Q_1 AND (Q_2 AND (... AND Q_n)) and Q_1 UNION (Q_2 UNION (... UNION Q_n)) as AND$(Q_1, \ldots, Q_n)$ and UNION$(Q_1, \ldots, Q_n)$, respectively.

A *substitution* σ is a function mapping finitely many variables to constants. We sometimes write σ as a set $\{x_1 \mapsto c_1, \ldots, x_n \mapsto c_n\}$. The domain $\mathrm{dom}(\sigma)$ of σ is the set of variables on which σ is defined. For a term $t \notin \mathrm{dom}(\sigma)$, let $\sigma(t) = t$; and for an atom $A = R(t_1, \ldots, t_n)$, let $\sigma(A) = R(\sigma(t_1), \ldots, \sigma(t_n))$. For X a set of variables, $\sigma|_X$ is the substitution obtained from σ by removing all mappings for variables outside X—that is, for each $x \in X$, substitution $\sigma|_X$ satisfies $\mathrm{dom}(\sigma|_X) = X$ and $\sigma|_X(x) = \sigma(x)$. To simplify the notation, for Q a query, we often abbreviate $\sigma|_{\mathsf{v}(Q)}$ as $\sigma|_Q$. Substitution σ is a *matcher* of an atom A to a fact F if $\mathrm{dom}(\sigma) = \mathsf{v}(A)$ and $\sigma(A) = F$; when such σ exists, it is unique. Substitutions σ_1 and σ_2 *join*, written $\sigma_1 \sim \sigma_2$, if

Table 1. The Syntax and the Semantics of the Query Language

Q	$\mathsf{v}(Q)$	Constraint	$\mathsf{ans}_I(Q)$
A	the vars of A		$\{\sigma \mid \sigma$ is a matcher of$A$ to some fact $F \in I\}$
Q_1 AND Q_2	$\mathsf{v}(Q_1) \cup \mathsf{v}(Q_2)$		$\{\!\{\sigma_1 \cup \sigma_2 \mid \sigma_1 \in \mathsf{ans}_I(Q_1),\ \sigma_2 \in \mathsf{ans}_I(Q_2),$ and $\sigma_1 \sim \sigma_2\}\!\}$
Q_1 UNION Q_2	$\mathsf{v}(Q_1)$	$\mathsf{v}(Q_1) = \mathsf{v}(Q_2)$	$\mathsf{ans}_I(Q_1) \cup \mathsf{ans}_I(Q_2)$
Q_1 MINUS Q_2	$\mathsf{v}(Q_1)$		$\{\!\{\sigma_1 \in \mathsf{ans}_I(Q_1) \mid \nexists \sigma_2 \in \mathsf{ans}_I(Q_2)$ such that $\sigma_1 \sim \sigma_2\}\!\}$
Q_1 FILTER E	$\mathsf{v}(Q_1)$	$\mathsf{v}(Q_1) \supseteq \mathsf{v}(E)$	$\{\!\{\sigma \mid \sigma \in \mathsf{ans}_I(Q_1)$ and $\sigma(E)$ evaluates to true$\}\!\}$
Q_1 BIND $x := E$	$\mathsf{v}(Q_1) \cup \{x\}$	$x \notin \mathsf{v}(Q_1) \supseteq \mathsf{v}(E)$	$\{\!\{\sigma \cup \{x \mapsto \sigma(E)\} \mid \sigma \in \mathsf{ans}_I(Q_1)$ and $\sigma(E) \neq \epsilon\}\!\}$
$\mathsf{PROJECT}_X(Q_1)$	X	$\mathsf{v}(Q_1) \supseteq X$	$\{\!\{\sigma\vert_X \mid \sigma \in \mathsf{ans}_I(Q_1)\}\!\}$
$\mathsf{DISTINCT}(Q_1)$	$\mathsf{v}(Q_1)$		$\{\sigma \mid \sigma \in \mathsf{ans}_I(Q_1)\}$

$\sigma_1(x) = \sigma_2(x)$ for each $x \in \mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2)$; in such cases, $\sigma_1 \cup \sigma_2$ is a substitution with domain $\mathsf{dom}(\sigma_1) \cup \mathsf{dom}(\sigma_2)$.

Evaluating a builtin expression E using a substitution σ satisfying $\mathsf{v}(E) = \mathsf{dom}(\sigma)$ produces a constant $\sigma(E)$ obtained by replacing in E all variables with their image in σ and evaluating the builtin functions as usual; if any of the builtin functions cannot be evaluated, then evaluation produces a special *error* value ϵ. For example, for $E = x + 2$ and substitutions $\sigma_1 = \{x \mapsto 3, y \mapsto c\}$ and $\sigma_2 = \{x \mapsto c\}$, we have $\sigma_1(E) = 5$ and $\sigma_2(E) = \epsilon$; the latter is because + cannot be applied to the constant c.

Evaluating a query Q over a database instance I produces a multiset of substitutions $\mathsf{ans}_I(Q)$ as specified in Table 1. Proposition 2.1 can be proved by a simple induction on the query structure.

Proposition 2.1. *For each database instance I, query Q, and substitution $\sigma \in \mathsf{ans}_I(Q)$, it is the case that $\mathsf{v}(Q) = \mathsf{dom}(\sigma)$.*

Our query language covers all of relational algebra with bag semantics, apart from a small detail in the definition of MINUS: if $\mathsf{ans}_I(Q_1) = \{\!\{x \mapsto a, x \mapsto a, x \mapsto a\}\!\}$ and $\mathsf{ans}_I(Q_2) = \{\!\{x \mapsto a\}\!\}$, then $\mathsf{ans}_I(Q_1$ MINUS $Q_2) = \emptyset$; in contrast, Q_1 MINUS Q_2 evaluates to $\{\!\{x \mapsto a, x \mapsto a\}\!\}$ under standard bag semantics. Our definition follows SPARQL 1.1 [30, Section 18.5]. Although $\mathsf{PROJECT}_X(Q_1)$ and Q_1 UNION Q_2 do not eliminate duplicates, the set variants of these operators can be expressed as $\mathsf{DISTINCT}(\mathsf{PROJECT}_X(Q_1))$ and $\mathsf{DISTINCT}(Q_1$ UNION $Q_2)$, respectively. Moreover, neither SPARQL nor Cypher requires $\mathsf{v}(Q_1) = \mathsf{v}(Q_2)$ in Q_1 UNION Q_2; for example, evaluating a SPARQL query

```
SELECT ?X ?Y WHERE { { ?X rdf:type :Person } UNION { ?X :hasName ?Y } }
```

can produce substitutions that are defined just on ?X, or on both ?X and ?Y. A similar problem arises with optional matches. Such features can be incorporated into our approach, but we do not discuss the details for the sake of simplicity. Furthermore, both SPARQL and Cypher provide grouping and aggregation. When aggregation and grouping are used at the top level of a query, their cardinality is equivalent to the cardinality of PROJECT followed by DISTINCT. In contrast, queries that join the result of aggregation with another subquery seem to be intrinsically difficult; we discuss this issue in detail in Section 5.3. Finally, we do not consider path queries in this article, although our preliminary investigation suggests that such features can be handled as well.

We next illustrate how to transform an RDF graph into our framework. We do *not* suggest that a graph database should be physically converted to apply our algorithms; rather, the objective of these transformations is to illustrate how to modify our algorithms so they are directly applicable to the RDF data model. Property graphs can be handled analogously.

Example 2.2. Consider the RDF graph shown in Figure 1, where each triple $\langle s, p, o\rangle$ is shown as a p-labelled edge from the vertex s to the vertex o. The following SPARQL query selects all

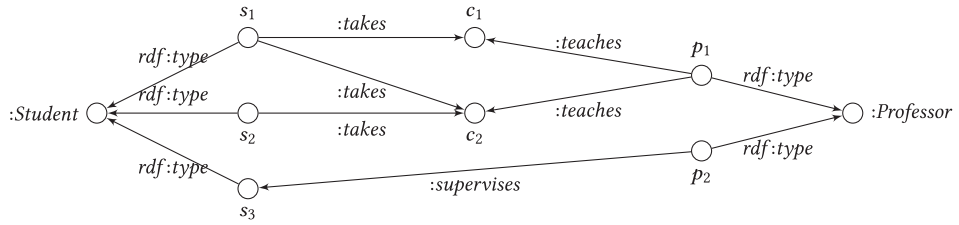


Fig. 1. RDF graph from Example 2.2.

distinct pairs of students and professors such that the student is enrolled in a course taught by the professor, or the student is supervised by the professor:

```
SELECT DISTINCT ?S ?P WHERE { ?S rdf:type :Student . ?P rdf:type :Professor .
    { SELECT ?S ?P WHERE { ?S :takes ?C . ?P :teaches ?C } } UNION { ?P :supervises ?S }
}
```

A straightforward way to encode any RDF graph into a relational model is to transform each triple $\langle s, p, o\rangle$ to a fact $T(s, p, o)$, where T is a fixed ternary relation. Using such an approach, the above SPARQL query corresponds to the following query in our language:

```
DISTINCT( AND( T(x,rdf:type,:Student), T(y,rdf:type,:Professor),
    PROJECT{x,y}( T(x,:takes,z) AND T(y,:teaches,z) )
    UNION
    T(y,:supervises,x)
) )
```

Note that projection is necessary to ensure that the two sides of the union have the same variables, as required by Table 1. When evaluated over facts corresponding to the RDF graph from Figure 1, this query returns the following three mappings:

$\{ \texttt{?S} \mapsto \texttt{:s1}, \texttt{?P} \mapsto \texttt{:p1} \}$ $\{ \texttt{?S} \mapsto \texttt{:s2}, \texttt{?P} \mapsto \texttt{:p1} \}$ $\{ \texttt{?S} \mapsto \texttt{:s3}, \texttt{?P} \mapsto \texttt{:p2} \}$

Without `DISTINCT`, the first mapping would be returned twice. *Vertical partitioning* [1, 5] is another well-known method for transforming an RDF graph into a relational model: each triple $\langle s, p, o\rangle$ is transformed to a unary fact $o(s)$ if p is equal to the *rdf:type* predicate, and otherwise it is transformed into a binary fact $p(s, o)$. Query atoms are transformed analogously.

To apply our results to the RDF data model directly, one can simply "invert" these transformations and adapt the notion of atoms to triples. ◁

2.2 Estimators

An *estimator* is a rule for calculating an estimate of an unknown quantity θ from observed data. For example, one can estimate the average height of a student population as the average of a randomly selected student sample, and one can use the sample variance to evaluate the estimate quality.

The estimation process is often modelled as a random variable, so we next recapitulate the relevant terminology and notation. A *random variable* $\hat{\theta}$ on a *sample space* Ω assigns to each *outcome* $\omega \in \Omega$ a value $\hat{\theta}(\omega) \in \mathbb{R}$. We shall consider only finite sample spaces, so we can associate each $\omega \in \Omega$ with a probability $\mathrm{P}(\omega)$. The expectation and variance of $\hat{\theta}$ are defined by

$$\mathbb{E}[\hat{\theta}] = \sum_{\omega\in\Omega} \mathrm{P}(\omega) \cdot \hat{\theta}(\omega) \qquad \text{and} \qquad \mathrm{Var}[\hat{\theta}] = \sum_{\omega\in\Omega} \mathrm{P}(\omega) \cdot (\hat{\theta}(\omega) - \mathbb{E}[\hat{\theta}])^2. \tag{1}$$

An estimator is *unbiased* if $\mathbb{E}[\hat{\theta}] = \theta$—that is, if its expectation is equal to the value being estimated.

The process of taking repeated estimates can be formally represented as an infinite sequence of random variables $\hat{\theta}_1, \hat{\theta}_2, \ldots$ on the same sample space Ω. Note that $\hat{\theta}_i$ need not be produced by the

same estimation rule, and in fact they can be correlated. Such a sequence is a *strongly consistent* estimator of θ if it converges to θ with probability one—that is,

$$\mathrm{P}\left(\omega \in \Omega \mid \lim_{n\to\infty} \hat{\theta}_n(\omega) = \theta\right) = 1. \tag{2}$$

An estimator's accuracy can often be improved by taking the average of several estimates. Formally, given a sequence $\hat{\theta}_1, \hat{\theta}_2, \ldots$, let $\hat{\mu}_n = \frac{1}{n} \cdot \sum_{i=1}^{n} \hat{\theta}_i$ be the sequence of random variables representing estimate averages. By the Kolmogorov's strong law of large numbers, if (i) $\hat{\theta}_i$ are independent and unbiased estimators of θ, (ii) for each $i \geq 1$, we have $\mathrm{Var}[\hat{\theta}_i] < \infty$, and (iii) $\sum_{i=1}^{\infty} \mathrm{Var}[\hat{\theta}_i]/i^2 < \infty$, then the sequence $\hat{\mu}_1, \hat{\mu}_2, \ldots$ is a strongly consistent estimator of θ. Moreover, all $\hat{\theta}_i$ are independent, so $\mathrm{Var}[\hat{\mu}_n] = \sum_{i=1}^{n} \mathrm{Var}[\hat{\theta}_i]/n^2$. Variance $\mathrm{Var}[\hat{\theta}_i]$ is often bounded in practice so, as the number of estimates increases, the average converges to the true value and the variance converges to zero.

If each outcome $\omega \in \Omega$ corresponds to a quantity $\phi(\omega)$ that can be determined from the outcome such that $\sum_{\omega\in\Omega} \phi(\omega) = \theta$, then $\hat{\theta}(\omega) = \phi(\omega)/\mathrm{P}(\omega)$ is the *Horvitz–Thompson estimator* [36] of θ. It is straightforward to see that a Horvitz–Thompson estimator is always unbiased.

Given n independent samples $t_1, \ldots, t_n$ produced by an unbiased estimator $\hat{\theta}$ of some unknown value θ, the sample average $\bar{t}$ and sample variance S^2 are given by

$$\bar{t} = \frac{\sum_{i=1}^{n} t_i}{n} \qquad \text{and} \qquad S^2 = \frac{\sum_{i=1}^{n} (t_i - \bar{t})^2}{n-1}. \tag{3}$$

A *p-confidence interval* for $0 \leq p \leq 1$, given by

$$\left[\bar{t} - z_p \cdot S/\sqrt{n}, \quad \bar{t} + z_p \cdot S/\sqrt{n}\right], \tag{4}$$

where z_p is the $(p+1)/2$ quantile of the normal distribution with expectation zero and variance one, is a possible measure of quality of estimating θ as $\bar{t}$. The value of p is usually expressed as a percentage, and $z_p = 1.96$ for the commonly used $p = 95\%$. Equation (4) is derived from two observations [9]. First, by the *central limit theorem*, random variable $(\hat{\theta}_1 + \cdots + \hat{\theta}_n - \mathbb{E}[\hat{\theta}])/(\sqrt{n} \cdot \mathrm{Var}[\hat{\theta}])$ converges in distribution to the standard distribution with expectation zero and variance one. Second, for large n, the sample average $\bar{t}$ and sample variance S^2 converge to $\mathbb{E}[\hat{\theta}]$ and $\mathrm{Var}[\hat{\theta}]$, respectively. It is difficult to choose n without knowing the distribution of $\hat{\theta}$, but $n \geq 30$ is frequently used in practice. Equation (4) can be intuitively understood as follows: if we repeatedly take n samples of $\hat{\theta}$ and compute each time the p confidence interval, then, for large n, we can expect the confidence interval to contain θ in roughly p percent of cases. Lindberg's version of the central limit theorem can be used to show that Equation (4) provides a p confidence interval even if each sample t_i is obtained using a possibly different estimator $\hat{\theta}_i$, provided that all $\hat{\theta}_i$ have the same expectation and that $\mathrm{Var}[\hat{\theta}_i] \leq V$ for each $i \geq 1$ and some finite V.[2]

2.3 Problem Statement

In this article, we present several estimators of $|\mathsf{ans}_I(Q)|$ for a database instance I and query Q. We do not construct any synopses or make any ad hoc assumptions about the data distribution, and we aim to use significantly less work than to compute $|\mathsf{ans}_I(Q)|$ exactly. We present each estimator as a randomised algorithm that realises a random variable $\hat{\theta}$. Thus, each outcome $\omega \in \Omega$ is a "record" of all random choices that an algorithm can make; $\mathrm{P}(\omega)$ is the probability of the algorithm making such choices; and $\hat{\theta}(\omega)$ is the value that the algorithm computes (deterministically) from ω.

[2]We thank Ke Yi from Hong Kong University of Science and Technology for a discussion of Lindberg's CLT.

To obtain accurate estimates, we shall run our estimators several times and use Equation (3) to compute the sample average and variance. In all cases, this will produce a strongly consistent estimator of $|\text{ans}_I(Q)|$—that is, the sample average is guaranteed to converge to $|\text{ans}_I(Q)|$. Moreover, for queries without DISTINCT, individual estimates will be unbiased. We will also use Equation (4) to compute the 95% confidence intervals of the final estimate.

Following the established practice in the literature, we use the *q-error* to measure the accuracy of cardinality estimation algorithms. In particular, if q and $\hat{q}$ are the real and estimated cardinalities, respectively, then the q-error is defined as

$$q\text{-}err(q,\hat{q}) = \begin{cases} \max(\frac{q}{\hat{q}}, \frac{\hat{q}}{q}) & \text{if } q \neq 0 \text{ and } \hat{q} \neq 0 \\ 1 & \text{if } q = 0 \text{ and } \hat{q} = 0, \text{ and} \\ \infty & \text{otherwise.} \end{cases}$$

3 Related Approaches to Query Cardinality Estimation

The problem of query cardinality estimation has been extensively studied in the literature and the space of proposed solutions is vast, so we cannot exhaustively survey the state of the art. We mentioned some of the more prominent approaches in Section 1, and in this section, we discuss the works that are more closely relevant to ours. In particular, in Section 3.1, we show that sampling-based methods can often be seen as instances of the very general framework by Lipton and Naughton [48]; in Section 3.2, we discuss in detail the WanderJoin algorithm [47]; and in Section 3.3, we discuss the cardinality estimation methods used in the G-CARE framework.

3.1 Principles of Sampling-based Cardinality Estimation

Numerous sampling-based cardinality estimation approaches have been proposed in the literature. Although seemingly different, many of them can be seen as instances of a general framework by Lipton and Naughton [48]. Let I be a database instance, let Q be a query, and let $\mathcal{A}$ be the set of answers of Q on I. Now assume that we have an effective way of partitioning $\mathcal{A}$ into disjoint subsets $\mathcal{A}_1, \ldots, \mathcal{A}_n$ so that $|\mathcal{A}_i|$ can be computed efficiently for each $1 \leq i \leq n$; we shall discuss shortly how this can be achieved in practice. We can then estimate the cardinality of Q on I as follows: we choose $i \in \{1, \ldots, n\}$ uniformly at random, we compute $|\mathcal{A}_i|$, and we return the estimate $n \cdot |\mathcal{A}_i|$. The expected estimate is $(\sum_{i=1}^{n} n \cdot |\mathcal{A}_i|)/n = \sum_{i=1}^{n} |\mathcal{A}_i| = |\mathcal{A}|$, where the last equality holds because $\mathcal{A}_1, \ldots, \mathcal{A}_n$ are disjoint; hence, our estimate is unbiased. In fact, we have a Horvitz–Thompson estimator [36] where $|\mathcal{A}_i|$ is the quantity corresponding to each outcome $i \in \{1, \ldots, n\}$.

This approach is applicable to any query that satisfies the assumption on answer partitioning, including recursive path or Datalog queries. Moreover, estimation accuracy can be improved by computing the average of several samples, and a key question is how many samples should be taken. For certain classes of queries, it is possible to precompute the number of samples so the resulting estimate is within desired bounds [48]. Alternatively, one can keep taking samples until the estimate falls within a confidence interval [29] computed as shown in Section 2.

Cardinality estimation should be orders of magnitude more efficient than query evaluation, so the set of answers $\mathcal{A}$ should be partitioned indirectly—that is, without computing it fully. This is usually achieved by partitioning the database I in a way that induces partitioning of $\mathcal{A}$. We next outline several ways to achieve this using the example query $Q = \text{AND}(R(x,y), S(y,z), T(z,x))$.

The CS2 approach [73] effectively partitions one relation in a query. For example, splitting $I(R)$ into disjoint subsets $I(R)_1, \ldots, I(R)_n$ induces a partition of the answers to Q where $\mathcal{A}_i$ is as the answer of Q on $I(R)_i \cup I(S) \cup I(T)$. The CS2 approach takes as input a *join schema* that identifies all supported joins, and it uses the join schema to construct a synopsis of the database: one relation

of the join schema is sampled, and all tuples from all other relations that join (possibly indirectly) with the sampled tuples are included into the synopsis. The cardinality of any query whose joins are contained within the join schema can be estimated by evaluating the query over the synopsis and scaling the result as shown by Lipton and Naughton [48].

Haas et al. [28] discuss theoretical and practical properties of estimators obtained by partitioning several relations of a query. On our example, such estimators split all of $I(R)$, $I(S)$, and $I(T)$ into disjoint subsets, and they take each answer partition $\mathcal{A}_{j,k,\ell}$ to be the answers of the query Q evaluated over partitions $I(R)_j$, $I(S)_k$, and $I(T)_\ell$.

Online aggregation algorithms [33] use similar principles to approximate answers of aggregation queries. The approach by Haas [26] can be seen as partitioning each relation into individual facts. On our example query Q, the algorithm randomly selects facts $R(a, b)$, $S(c, d)$, and $T(e, f)$; if the facts join (i.e., if $b = c$, $d = e$, and $f = a$), then the cardinality is estimated as $|I(R)| \cdot |I(S)| \cdot |I(T)|$, and otherwise it is estimated as zero. The algorithm can also handle `DISTINCT` queries by estimating the cardinality as zero whenever a repeated combination of values is encountered, but the resulting estimator is unbiased only if sampling is performed without replacement. The *ripple join* algorithm [27] improves this idea to ensure faster convergence: all selected facts are kept in memory, and, whenever a new fact is added to the selected set of facts, it is joined with all previously selected facts to update the running estimation of query cardinality. WanderJoin [47] is a recent proposal that was empirically shown to outperform ripple join. WanderJoin provides the foundation for our work so discuss in detail the variant from the G-CARE framework in Section 3.2.

Charikar et al. [15] proved that estimating the cardinality of `DISTINCT` queries over a single relation is inherently difficult: no estimator can guarantee small error across all databases and queries unless it examines a large fraction of the database. They also presented a provably optional, but not unbiased estimator, as well as several heuristic estimators optimised for typical inputs.

3.2 The WanderJoin Algorithm

We describe the idea behind WanderJoin using the following example:

Example 3.1. Let $Q = \mathtt{AND}(R(x, y), S(y, z), T(z, x))$ be the example query from Section 3.1, and let I be the following database instance.

$R(a, b_1)$	$S(b_1, c_1)$	$T(c_1, d_1)$
$R(a, b_2)$	$S(b_1, c_2)$	$T(c_1, a)$
	$S(b_1, c_3)$	$T(c_4, d_2)$
	$S(b_2, c_4)$	
	$S(b_2, c_5)$	

The WanderJoin algorithm randomly selects one fact of I per query atom, but the choices are not independent. A fact for $R(x, y)$ is chosen from the set $I_1 = I(R)$ of all facts for relation R. Assume that $R(a, b_1)$ is selected. Next, $S(y, z)$ is matched to the set $I_2 = \{S(b_1, c_1), S(b_1, c_2), S(b_1, c_3)\}$ of facts with b_1 in the first position: no fact in $I(S) \setminus I_2$ joins with $R(a, b_1)$ so one can disregard $I(S) \setminus I_2$, and this increases the chance of obtaining a valid answer. One can now proceed in two ways.

First, assume that $S(y, z)$ is matched to $S(b_1, c_1)$. All variables of Q have been fixed at this point; hence, $I_3^1 = \{T(c_1, a)\}$ is the set of candidates for atom $T(z, x)$ and so one can choose $T(c_1, a)$ deterministically. Facts $R(a, b_1)$, $S(b_1, c_1)$, and $T(c_1, a)$ provide one answer to Q, and they are chosen with probability $\mathrm{P}_1 = 1/|(I_1| \cdot |I_2| \cdot |I_3^1|) = 1/(2 \cdot 3 \cdot 1)$; therefore, $1/\mathrm{P}_1 = 6$ is the Horvitz–Thompson (and thus unbiased) estimate of the cardinality of Q on I.

Second, assume that $S(y, z)$ is matched to $S(b_1, c_2)$. The set of candidates for atom $T(z, x)$ is now $I_3^2 = \emptyset$—that is, there is no way to match $T(z, x)$ and obtain an answer to Q. The probability of

choosing $R(a, b_1)$ and $S(b_1, c_2)$ is $\mathrm{P}_2 = 1/(|I_1| \cdot |I_2|) = 1/6$, but this choice does not provide a query answer so the Horvitz–Thompson estimate is $0/\mathrm{P}_2 = 0$. When sampling is repeated to compute the average, such zero estimates must be included into the average.

This process can be seen as random walk on the graph whose vertices correspond to the facts of I, and where two facts are connected if they join according to Q. For example, facts $R(a, b_1)$ and $S(b_1, c_2)$ are considered connected, since they satisfy the join of $R(x, y)$ and $S(y, z)$. Note that this graph is not constructed explicitly, but is used only to understand the sampling process. ◁

The order in which atoms are sampled critically determines the accuracy of the estimates produced by WanderJoin.

Example 3.2. Let $Q' = \mathtt{AND}(T(z, w), S(y, z), R(x, y))$ be a reordering of the query Q from Example 3.1; the answers to both queries clearly coincide. All relations in I satisfy a functional dependency from the second to the first argument so, once we match $T(z, w)$, all other atoms can be matched deterministically. This increases the likelihood of obtaining a valid query answer, which ultimately improves estimate precision. Indeed, if we match $T(z, w)$ to $T(c_1, a)$, then we can deterministically choose $S(b_1, c_1)$ and $R(a, b_1)$ and return the estimate $3 \cdot 1 \cdot 1 = 3$. ◁

Using a "good" order of atoms is thus key to obtaining precise estimates, but selecting such an order *a priori* is difficult. Li et al. [47] suggest to first conduct a fixed number of trial runs using all reasonable orders; after all trial runs have completed, one should compute the variance and the cost for each order, select the order with the least cost, and use this order in all remaining runs until either a time budget is exhausted or the estimate falls within a desired confidence interval.

Park et al. [55] included a variant of this approach into the G-CARE framework; we denote this variant by WJ. To estimate the cardinality of a conjunction of atoms, WJ conducts 30 estimation attempt; each attempt reports a single estimate, and the average of these 30 results is reported as the final estimate. Each estimation attempt consists of the following steps. First, all reasonable orders are enumerated. Second, the total number of runs to make is computed by

$$NR = \left\lfloor 0.03 \cdot \frac{|I(R_1)| + \cdots + |I(R_n)|}{n} \right\rfloor, \tag{5}$$

where $R_1, \ldots, R_n$ are the relations of all query atoms. Third, a series of trial runs is performed. In each run, an order is selected in a round-robin fashion, and a cardinality estimate is produced as in Example 3.1. The trial run phase finishes either after NR runs or when one order accumulates 100 valid samples. In the latter case, all orders with at least 50 valid samples are identified, the order with least estimate variance among these is selected as the "best" one, and all remaining (i.e., up to NR) runs are performed using this "best" order. Finally, the result of the estimation attempt is computed as the average of the estimates of all (i.e., trial and "best"-order) runs.

The WJ variant by Park et al. [55] can handle only queries of the form $Q = \mathtt{AND}(A_1, \ldots, A_n)$. The variant by Li et al. [47] can also handle aggregate queries with grouping, for which it provides an aggregation estimate for each group; however, this does not estimate *the number of groups* and thus does not provide a solution for `DISTINCT` queries. Moreover, none of the versions of WanderJoin known to us can handle queries involving arbitrary nesting of query operators.

A key advantage of WanderJoin is that it does not require schema-level synopses whose construction requires anticipating the query workload. The latter is often possible in relational databases; for example, "Line Items" are likely to be joined with "Orders" but not with "Employees," and self-joins of two instances of "Orders" are unlikely. In contrast, queries in graph databases often explore patterns that are difficult to predict, and self-joins are frequent (e.g., "friends of my friends").

3.3 Cardinality Estimation Methods Used in the G-CARE Framework

We now briefly summarise the cardinality estimation algorithms in the G-CARE framework [55]. Park et al. have classified their approaches into the synopsis-based and sampling-based.

Synopsis-based methods follow the principles outlined in Section 1: a graph is summarised using a synopsis data structure, which is used to estimate the cardinality of a class of queries. Park et al. [55] actually call such methods *summary-based*; however, we prefer *synopsis-based* because the term "summary" often has a more specific meaning in the graph summarisation literature [49].

The *Bounded Sketch* (BS) method by Cai et al. [13] divides each relation into partitions, and, for each partition and each attribute, records the number of constants and the maximum degree. To estimate the cardinality of a query, each partition is processed using bounding formulas by Khamis et al. [42]. These formulas are based on deep insights about how query structure limits the maximum number of answers that a query can produce on a family of databases.

The *Characteristic Sets* (C-SET) method by Neumann and Moerkotte [52] was developed in the context of the highly influential RDF-3X system [54]. It uses a synopsis that enumerates all types of star-shaped structures in the database with their respective counts. The cardinality of a query is estimated by decomposing the query into star-shaped subqueries, estimating the cardinality of each subquery using the synopsis, and combining the estimates using the independence assumption.

The SUMRDF method by Stefanoni et al. [63] uses a synopsis obtained via graph summarisation—the process of merging graph vertices until the graph size falls within a given budget. The synopsis is then interpreted using the *possible worlds* semantics: any graph that produces the same summary is possible. The cardinality of a conjunctive query is estimated as the average number of answers over all possible worlds. The method can also provide certainty bounds on the estimate.

Sampling-based methods in the G-CARE framework follow the ideas from Section 3.1. We have described *WanderJoin (WJ)* in Section 3.2, so we next focus on the remaining three methods.

The JSUB method is derived from the work by Zhao et al. [76]: It selects uniformly at random a fact matching the first query atom, and then it evaluates the remaining atoms with the first atom bound to the selected fact. Thus, JSUB is similar to CS2 [73] in that it samples just one atom, but sampling is performed for each estimate; in contrast, CS2 uses sampling to create a synopsis.

The *Correlated Sampling* (CS) method [70] in the G-CARE framework uses a similar approach, but, instead of sampling the data, it uses hashing to produce a database synopsis.

The IMPR method adapts the sampling-based technique by Chen and Lui [18] for estimating the number of k-node graphlets for $k \in \{2, 4, 5\}$. Roughly speaking, IMPR uses random walks to identify a *visible* subgraph of a given graph and then counts the number of answers on the visible subgraph to provide an estimate of the number of graphlets. Park et al. [55] adapted this technique to graph matching, as well as to work on directed labelled graphs.

4 Motivation

Two observations motivate the results presented in this article. The first one is that no method we mentioned in Section 1 or 3 can process complex queries with arbitrary operator nesting such as the one in Example 1.1. One might attempt to apply the framework by Lipton and Naughton [48] from Section 3.1, but Example 4.1 reveals several problems with such an approach.

Example 4.1. Consider queries Q_1 and Q_2 and a database instance I as follows.

$$Q_1 = \texttt{DISTINCT}(Q_2) \qquad Q_2 = \texttt{PROJECT}_{\{x,z\}}(R(x,y)\ \texttt{AND}\ S(y,z)) \tag{6}$$

$$I = \{R(a, b_i), S(b_i, c) \mid 1 \leq i \leq k\} \tag{7}$$

To apply the approach from Section 3.1 to Q_2, we could partition I into $I_i = \{R(a, b_i), S(b_i, c)\}$ for $1 \leq i \leq k$; clearly, $\mathsf{ans}_I(Q) = \bigcup_{i=1}^{k} \mathsf{ans}_{I_i}(Q_2)$, as required. To estimate the cardinality of Q_2 in I, we randomly choose $i \in \{1, \ldots, k\}$ and return $k \cdot |\mathsf{ans}_{I_i}(Q_2)|$ as the estimate. Since I_i is much smaller than I, computing $|\mathsf{ans}_{I_i}(Q_2)|$ is likely to be much faster than computing $|\mathsf{ans}_I(Q_2)|$.

Duplicate elimination reduces the number of answers in a way that can prevent effective partitioning. Indeed, $\mathsf{ans}_I(Q) \neq \bigcup_{i=1}^{k} \mathsf{ans}_{I_i}(Q_2)$, so the partitioning from the previous paragraph is not applicable. In fact, it is unclear how to partition I into $I_1, \ldots, I_n$ so $\mathsf{ans}_{Q_1}(I) = \bigcup_{i=1}^{n} \mathsf{ans}_{Q_1}(I_i)$ holds but computing $|\mathsf{ans}_{I_i}(Q_2)|$ is much faster than computing $|\mathsf{ans}_I(Q_2)|$.

The approach we present in Section 5 addresses these problems. In particular, it can process query Q_1 efficiently, and it is applicable even if Q_1 is conjoined with another query. ◁

Our second observation is that the WanderJoin variant by Park et al. [55] is indeed very accurate, but it can be slow on large datasets. To show this, we repeated the experiments by Park et al. [55] on an extended set of datasets. We next present our experimental setup and discuss our findings.

Datasets. Park et al. [55] tested the accuracy of cardinality estimation methods on the following four benchmarks:

- The AIDS Antiviral Screen dataset [60] describes chemical compounds and has been used for benchmarking various graph problems.
- The Human dataset [75] describes protein interactions using the Gene Ontology vocabulary.
- The Yago [64] knowledge graph was derived from WikiPedia and WordNet and has been used in applications such as entity linking, information extraction, and ontology construction.
- The LUBM [25] benchmark has been extensively used to test various aspects of RDF systems. It provides a generator of arbitrarily sized graphs, an OWL ontology that can be used to perform logical inference over the graphs, and 14 test queries.

Park et al. [55] generated an extensive set of test queries for the Yago, AIDS, and Human datasets and made them available in the G-CARE GitHub repository.[3]

All four datasets described above are fairly small: Yago was the largest dataset with 15.8M edges. (In the G-CARE study, a much larger DBpedia dataset with 225M edges was used in the query planning experiments, but not in the accuracy experiments.) To test the approaches from the G-CARE framework on much larger inputs, we extended the datasets as follows:

- We replaced the version of LUBM with the LUBM-01K-mat dataset, which includes a much larger base graph as well as facts logically implied by the LUBM ontology. We used the 14 standard test queries, as well as 23 queries handcrafted by Stefanoni et al. [63].
- We produced a large dataset using the WatDiv benchmark [4]. WatDiv provides a generator that produces graphs and accompanying queries. We find WatDiv interesting because it was designed to produce graphs with nonuniform data distribution, and the latter often causes problems for cardinality estimation. Most WatDiv queries contain one constant and thus produce a small number of answers so, to obtain queries that produce large answer sets, we additionally produced "free" queries by replacing all constants with variables.
- We used the DBLP benchmark by Zou et al. [77], where we extended the standard queries with further nine handcrafted queries.

We thus obtained six benchmarks shown in Table 2. The numbers of facts correspond to the result of transforming an RDF graph using vertical partitioning, cf. Section 2.1: unary and binary facts correspond to labelled vertices and edges, respectively. The new datasets are between one and three orders of magnitude larger than those considered in the G-CARE study.

[3] https://github.com/yspark-dblab/gcare

Table 2. Summary of the Used Benchmarks

	AIDS	Human	Yago	LUBM-01K-mat	WatDiv	DBLP
# unary facts	254,156	21,621	42,441,193	50,245,643	1,359,262	5,475,754
# binary facts	547,910	172,564	15,835,675	132,123,767	107,638,452	50,111,001
# queries	780	49	1,366	37	104	15
min. card.	1	1	1	0	0	1
max. card.	951,601	9,610	163,118,890	588,378,270	4,244,261	2,284,408
# card. $\leq 10,000$	379	49	939	23	91	9

Table 3. Summary of the Results on All Datasets

	BS	CS	C-SET	IMPR	JSUB	SUMRDF	WJ	BS	CS	C-SET	IMPR	JSUB	SUMRDF	WJ
	AIDS							LUBM						
# errors	0	0	0	500	0	0	0	0	0	0	9	0	0	0
# timeouts	0	0	0	0	0	287	0	0	2	0	0	0	0	0
# $q\text{-}err = \infty$	0	329	0	145	14	0	5	1	5	0	11	17	2	0
# $q\text{-}err < \infty$	780	451	780	135	766	493	775	36	30	37	17	20	35	37
	Human							WatDiv						
# errors	0	0	0	0	0	0	0	0	0	0	39	0	0	0
# timeouts	0	0	0	0	0	0	0	0	1	0	0	0	0	0
# $q\text{-}err = \infty$	0	15	0	22	3	0	0	17	28	3	11	54	8	8
# $q\text{-}err < \infty$	49	34	49	27	46	49	49	87	75	101	54	50	96	96
	Yago							DBLP						
# errors	0	0	0	1004	0	0	0	0	0	0	6	0	0	0
# timeouts	0	0	0	0	0	839	0	0	1	0	0	0	0	0
# $q\text{-}err = \infty$	0	688	0	189	308	0	175	0	6	0	0	9	0	4
# $q\text{-}err < \infty$	1,366	678	1,366	173	1,058	527	1,191	15	8	15	9	6	15	11

Test Setting. We compiled the G-CARE code from GitHub, ran it on the six datasets from Table 2, and recorded all estimates and estimation times. The framework imposes a five-minute timeout on each run of an algorithm. We used a server with an Intel Core i7-13700 CPU running at 2.1 GHz with eight performance and eight efficiency cores; the efficiency cores are extended to 16 logical cores via hyperthreading. The server has 64 GB of main memory and an NVidia GeForce RTX 4080 GPU with 16 GB of RAM, and it was running Ubuntu 22.04, kernel version 6.5.0-14-generic.

Results. Figure 2 shows the q-errors (left) and estimation times (right) produced by the G-CARE framework. Some approaches produce very small, yet nonzero estimates, which in turn yield very large q-errors that can distort the results. We thus follow the practice by Park et al. [55] and Stefanoni et al. [63] and round to one all nonzero cardinalities smaller than one. The number of queries is very large so we cannot show the per-query results in this article. Detailed results are available online [37], and here we summarise the result distribution using box plots, each showing the minimum, lower quartile, median, upper quartile, and maximum values for q-errors and running times. Values that cannot be plotted (e.g., infinite q-errors or timeouts) are shown using maximum whiskers with arrows. Finally, we show the average of all valid values as a dot. Table 3 shows, for each benchmark and estimation method, the numbers of queries that produced an unspecified runtime error, timeout, infinite q-error, and finite q-error.

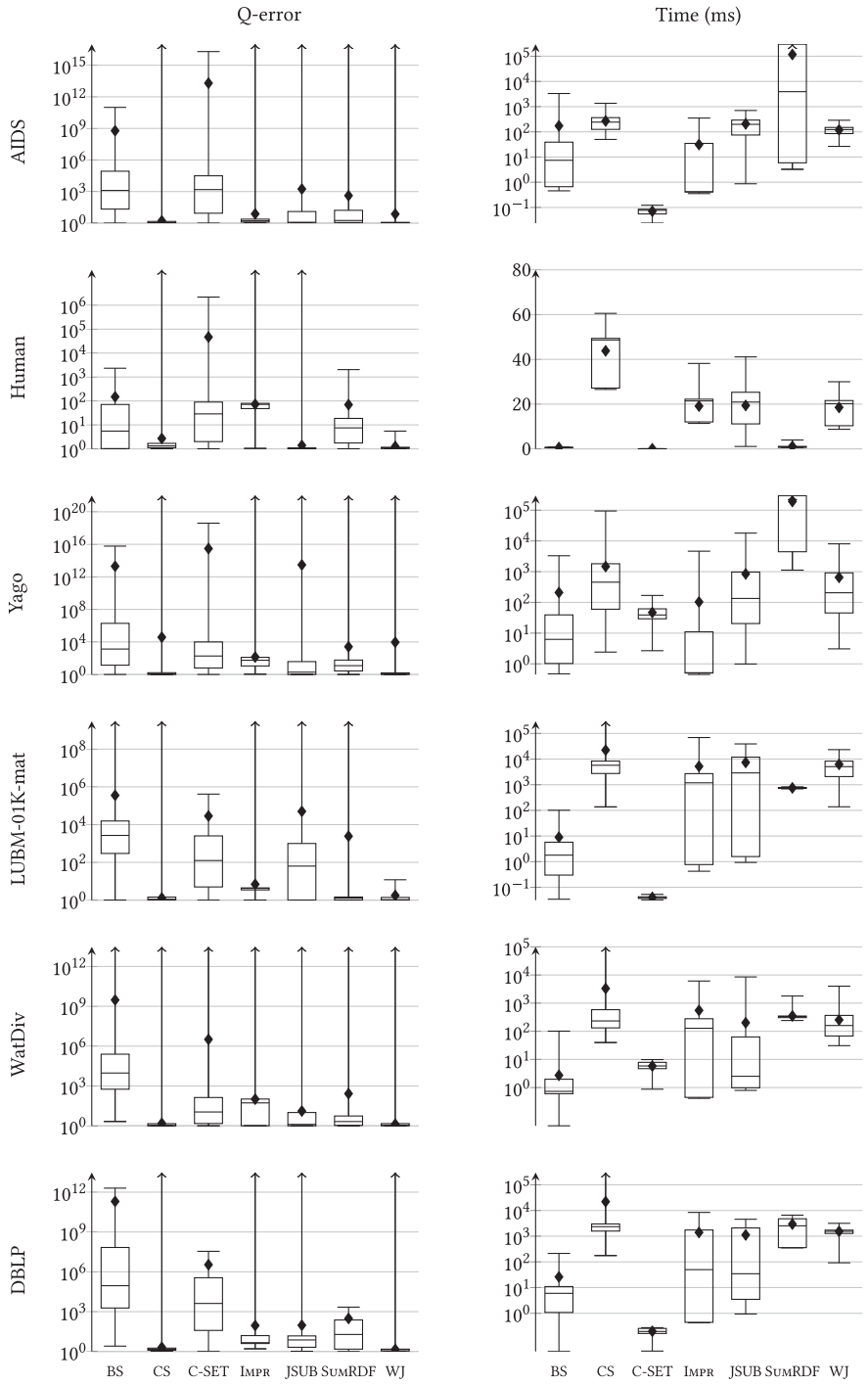


Fig. 2. Results of the G-CARE Framework on All Benchmarks

Discussion. Our results confirm the conclusions by Park et al. [55] about estimation accuracy: WJ outperforms all other methods. Although the q-errors of CS an WJ seem comparable, Table 3 shows that WJ can successfully estimate a much larger number of queries. In fact, WJ performs even better on the new datasets: larger data sizes typically increase the maximum q-error for most techniques, while the maximum q-error of WJ seems largely unchanged.

Our results also agree with the observations by Park et al. [55] about estimation times on the original datasets: the performance of WJ is roughly in line with the other methods. However, a slightly different picture emerges on the larger datasets: C-SET and BS are fastest, and the remaining techniques exhibit roughly the same maximum running times; however, the minimum running times of WJ are several orders of magnitude larger than of most other techniques. Moreover, the average running times of all techniques, but WJ in particular, are quite large: the averages for LUBM-01K-mat, WatDiv, and DBLP are around 6.2 s, 0.25 s, and 1.5 s, respectively. A cardinality estimation routine is often called hundreds or even thousands of time during query planning, so it is essential that estimates are computed quickly. The WJ variant clearly does not satisfy this requirement. We identified three plausible explanations for this.

First, as we discussed in Example 3.2, the order of query atoms critically determines the accuracy of WJ. Since it is unclear how to identify a "good" order in advance, the WJ variant considers all possible orders. While this benefits accuracy, it inevitably increases the running times, particularly on large queries with many possible orders.

Second, the number of samples to take is determined using Equation (5), which, in most cases, depends linearly on the data size. It intuitively makes sense to take more samples on larger inputs to explore larger portions of the sample space, but such reasoning is actually misleading. For example, queries containing a constant are localised to a subset of the input graph around that constant; this subset typically does not change even if the input grows in size so taking more samples is unjustified. Moreover, even for queries that return more answers on larger graphs, estimation times that scale linearly with the input size are not adequate for query planning.

Third, motivated by the folklore belief that 30 samples are generally sufficient, WJ repeats each estimation attempt 30 times and reports the average. Combined with the first two observations, this further increases the number of samples, particularly on large graphs.

To summarise, the number of samples tends to scale with the size of the input dataset and the query. Since the average of unbiased estimates converges to the actual cardinality, the accuracy of WJ is unsurprising. However, such an approach can easily become more costly than answering the query exactly. The G-CARE framework does not provide code for computing the exact number of answers, and Park et al. [55] do not report exact query answering times. We show in Section 7 that exact cardinalities can be computed much faster than in Figure 2; for example, our implementation needs at most 1.7 s, 55 ms, and 405 ms to answer any LUBM, WatDiv, and DBLP query, respectively. Hence, further work is needed to turn WJ into an effective cardinality estimation approach.

5 Strongly Consistent Cardinality Estimator for Complex Queries

Towards presenting our cardinality estimation approach, in Section 5.1 we first present a query evaluation algorithm that provides the necessary context. Then, in Section 5.2 we discuss the intuitions; in Section 5.3 we present the basic algorithm and state its properties; and in Section 5.4 we discuss an important optimisation. Finally, in Section 5.5 we discuss several practical issues.

5.1 Query Evaluation via Sideways Information Passing

Standard query evaluation proceeds bottom-up as in Table 1; for example, $Q = Q_1$ AND Q_2 is evaluated by computing $\mathsf{ans}_I(Q_1)$ and $\mathsf{ans}_I(Q_2)$, and joining the two results. *Sideways information passing* techniques aim to optimise this process by allowing one operator in a query plan to identify a set

ALGORITHM 1: $\mathrm{eval}_I(Q, \sigma)$

Input: database instance I, query Q, and substitution σ with $\mathrm{dom}(\sigma) \subseteq \mathrm{v}(Q)$
Output: each substitution $\mu \in \mathrm{ans}_I(Q)$ such that $\sigma \sim \mu$
Local: set D of substitutions

```
1:  switch Q
2:  case Q = A
3:    for each matcher β of σ(A) to some F ∈ I do
4:      output σ ∪ β
5:  case Q = Q1 AND Q2
6:    for each σ1 ∈ eval_I(Q1, σ|_Q1) do
7:      for each σ2 ∈ eval_I(Q2, (σ ∪ σ1)|_Q2) do
8:        output σ ∪ σ1 ∪ σ2
9:  case Q = Q1 UNION Q2
10:   for each i ∈ {1, 2} do
11:     for each σi ∈ eval_I(Qi, σ) do
12:       output σi
13: case Q = Q1 MINUS Q2
14:   for each σ1 ∈ eval_I(Q1, σ) do
15:     if eval_I(Q2, σ1|_Q2) = ∅ then
16:       output σ1
17: case Q = Q1 FILTER E
18:   for each σ1 ∈ eval_I(Q1, σ) do
19:     if σ1(E) = true then
20:       output σ1
21: case Q = Q1 BIND x := E
22:   for each σ1 ∈ eval_I(Q1, σ|_Q1) do
23:     if σ1(E) ≠ ϵ and σ ~ {x ↦ σ1(E)} then
24:       output σ1 ∪ {x ↦ σ1(E)}
25: case Q = PROJECT_X(Q1)
26:   for each σ1 ∈ eval_I(Q1, σ) do
27:     output σ1|_X
28: case Q = DISTINCT(Q1)
29:   D := ∅
30:   for each σ1 ∈ eval_I(Q1, σ) do
31:     if σ1 ∉ D then
32:       D := D ∪ {σ1}
33:       output σ1
```

of possible bindings for query variables and pass these to other operators in the plan to eagerly eliminate tuples that do not match these bindings. Such techniques have been applied to relational [40, 61], RDF [53, 74], and recursive [8] queries, and in visual query processing [41].

Procedure $\mathrm{eval}_I(Q, \sigma)$ in Algorithm 1 uses a variant of sideways information passing to evaluate a query Q over a database instance I. This algorithm can be practical, but our point is mainly conceptual: our cardinality estimation algorithm can be seen as "sampling the loops" of Algorithm 1. Intuitively, the algorithm can be seen as a variant of the magic sets transformation [8] adapted to complex queries with nesting. For example, it evaluates $Q = Q_1$ AND Q_2 by enumerating all answers of Q_1 and evaluating Q_2 in the context of each answer. The algorithm realises sideways information passing via the substitution σ, which must satisfy $\mathrm{dom}(\sigma) \subseteq \mathrm{v}(Q)$, and it provides the *context* produced by subqueries evaluated prior to Q and is used to constrain the evaluation of Q. The procedure outputs each substitution $\mu \in \mathrm{ans}_I(Q)$ such that $\sigma \sim \mu$; that is, only answers compatible with the context substitution are produced. We present Algorithm 1 in form of a *generator*: each invocation of $\mathrm{eval}_I(Q, \sigma)$ should be understood as providing an iterator, and the **output** keyword adds one substitution to the iterator result. The algorithm can be easily turned into a form that uses standard iterators and evaluates all queries apart from DISTINCT(Q) in a pipelined fashion.

Procedure $\mathrm{eval}_I(Q, \sigma)$ considers all possible forms of Q (line 1). When Q is an atom A (line 2), the algorithm identifies all ways in which $\sigma(A)$ can be matched in I (lines 3–4). For $Q = Q_1$ AND Q_1, the algorithm uses a nested loop join: it evaluates Q_1 in the context of σ (line 6) and, for each resulting σ_1, it evaluates Q_2 in the context of $\sigma \cup \sigma_1$ (line 7) and outputs the result. To ensure that the context substitutions are defined over the variables of the respective subquery, σ and $\sigma \cup \sigma_1$ are projected to $\mathrm{v}(Q_1)$ and $\mathrm{v}(Q_2)$ in lines 6 and 7, respectively. For $Q = Q_1$ UNION Q_2, the algorithm evaluates Q_1 and Q_2 in the context of σ independently. For $Q = Q_1$ MINUS Q_2, the algorithm evaluates Q_1 in the context of σ (line 14), and it filters out each answer σ_1 that can be extended to an answer of Q_2 (line 15). Again, σ_1 is restricted to $\mathrm{v}(Q_2)$ to obtain a valid context substitution. Moreover, the evaluation of $\mathrm{eval}_I(Q_2, \sigma_1|_{Q_2})$ in line 15 can stop as soon as one answer substitution is identified.

The case of $Q = Q_1$ FILTER E is analogous. For $Q = Q_1$ BIND $x := E$, each σ_1 obtained by evaluating Q_1 is extended by mapping x to $\sigma_1(E)$, and the result is output only if it is compatible with σ. For $Q = \mathtt{PROJECT}_X(Q_1)$, the algorithm simply removes the bindings for variables outside X in each answer obtained by evaluating Q_1 in the context of σ. Finally, for $Q = \mathtt{DISTINCT}(Q_1)$, a substitution produced by the evaluation of Q_1 is returned only the first time it is encountered. Note that the set D used to remove duplicate substitutions is local to each invocation of $\mathsf{eval}_I(Q, \sigma)$.

Theorem 5.1 captures formally the relevant properties of Algorithm 1. A straightforward consequence is that $\mathsf{ans}_I(Q) = \mathsf{eval}_I(Q, \emptyset)$ for each database instance I and query Q.

Theorem 5.1. *For each database instance I, query Q, and substitution σ such that* $\mathsf{dom}(\sigma) \subseteq \mathsf{v}(Q)$,

$$\mathsf{eval}_I(Q, \sigma) = \{\!\{\mu \in \mathsf{ans}_I(Q) \mid \sigma \sim \mu\}\!\}. \tag{8}$$

Proof Sketch. The claim can be proved by a straightforward induction on the structure of Q. The induction base holds immediately from the definition of a matcher of $\sigma(A)$ to a fact $F \in I$. For the induction step, we consider different forms of Q and an arbitrary σ such that $\mathsf{dom}(\sigma) \subseteq \mathsf{v}(Q)$. For $Q = Q_1$ AND Q_2, property (8) holds for Q_1 and Q_2. Thus, each substitution σ_1 in Algorithm 1 satisfies $\sigma \sim \sigma_1$, and each substitution σ_2 in Algorithm 1 satisfies $\sigma \cup \sigma_1 \sim \sigma_2$; hence, $\sigma \sim \sigma \cup \sigma_1 \cup \sigma_2$, as required. Moreover, $\sigma \cup \sigma_1 \cup \sigma_2 \in \mathsf{eval}_I(Q, \sigma)$ clearly holds, and it should be obvious that the multiset $\mathsf{eval}_I(Q, \sigma)$ contains all required substitutions with the corresponding multiplicities. Cases for $Q = Q_1$ UNION Q_2 and $Q = \mathtt{PROJECT}_X(Q_1)$ are analogous. For $Q = \mathtt{DISTINCT}(Q_1)$, the induction assumption ensures that (8) holds for Q_1, which in turn ensures

$$\{\!\{\mu \in \mathsf{ans}_I(Q) \mid \sigma \sim \mu\}\!\} = \{\mu \in \mathsf{ans}_I(Q_1) \mid \sigma \sim \mu\} = \{\mu \mid \mu \in \mathsf{eval}_I(Q_1, \sigma)\} = \mathsf{eval}_I(Q, \sigma).$$

The last equality is due to how the set D is used in Algorithm 1 to eliminate duplicates. The cases for $Q = Q_1$ FILTER E and $Q = Q_1$ BIND $x := E$ are straightforward. Finally, for $Q = Q_1$ MINUS Q_2, the induction assumption ensures that (8) holds for Q_1 and Q_2, which in turn ensures

$$\begin{aligned}\{\!\{\mu_1 \in \mathsf{ans}_I(Q) \mid \sigma \sim \mu_1\}\!\} &= \{\!\{\mu_1 \in \mathsf{ans}_I(Q_1) \mid \sigma \sim \mu_1 \text{ and } \nexists \mu_2 \in \mathsf{ans}_I(Q_2) \text{ such that } \mu_1 \sim \mu_2\}\!\} = \\ &= \{\!\{\mu_1 \in \mathsf{eval}_I(Q_1, \sigma) \mid \mathsf{eval}_I(Q_2, \mu_1|_{Q_2}) = \emptyset\}\!\} = \mathsf{eval}_I(Q, \sigma).\end{aligned}$$

Again, the last equality is ensured by the structure of Algorithm 1. □

If matchers in line 3 can be computed using indexes, then the evaluation of $Q = \mathtt{AND}(A_1, \ldots, A_n)$ amounts to index nested loop joins, which are widely used in practice. Our algorithm processes one tuple at a time in lines 6–8, which can incur a cost due to random access. However, in RAM-based databases, this cost is often compensated by sideways information passing, which can significantly reduce the overall number of processed tuples. Algorithm 1 can thus be practical in certain cases, and it is used by the prototype implementation from Section 7 to evaluate queries exactly.

5.2 Principles for Estimating the Cardinality of Complex Queries

The inspiration for our work comes from the WanderJoin algorithm (see Section 3.2), which can be seen as "sampling the loops" of Algorithm 1. Given $Q = A_1$ AND A_2, Algorithm 1 enumerates each matcher σ_1 of $\sigma(A_1)$ to a fact in I, and for each σ_1 it enumerates each matcher σ_2 of $(\sigma \cup \sigma_1)(A_2)$ to a fact in I. In contrast, WanderJoin guesses just one such pair of σ_1 and σ_2, and this process can be seen as using sideways information passing to compute just one answer to Q. We next present several examples that illustrate how to extend these principles to other query types.

Example 5.2. Let Q and I be the following query and database instance, respectively, so $\mathsf{ans}_I(Q)$ contains substitutions of the form $\{x \mapsto a_i, y \mapsto b, z \mapsto c_j\}$ for $1 \le i \le 4$ and $1 \le j \le 2$.

$$Q = Q_1 \text{ AND } T(y, z) \qquad Q_1 = R(x, y) \text{ UNION } S(x, y)$$
$$I = \{R(a_1, b), R(a_2, b), R(a_3, b), S(a_4, b), T(b, c_1), T(b, c_2), T(d, e)\}$$

The UNION operator does not eliminate duplicates (cf. Section 2.1), so one might intuitively estimate its cardinality as the sum of the cardinality of its disjuncts. However, to handle the conjunction in Q, we would need to pass cardinality estimates for $R(x, y)$ and $S(x, y)$ sideways to $T(y, z)$, and it is unclear how to combine them into an unbiased estimate of the cardinality of Q.

Our solution is to "sample the loops" of Algorithm 1. Instead of considering both $i = 1$ and $i = 2$ in line 10, we randomly select just one disjunct, we randomly produce one answer for the selected disjunct, and we pass it sideways to $T(y, z)$. For example, we could select $i = 1$ and then randomly select one matcher of $R(x, y)$ to a fact in I; for example, $\sigma_1 = \{x \mapsto a_1, y \mapsto b\}$. There are three candidate matchers, so we estimate the cardinality of $R(x, y)$ as 3. To account for the two possible choices for i in line 10, we estimate the cardinality of Q_1 as $3 \cdot 2 = 6$. We thus obtain a single answer and estimate for Q_1, which we pass sideways to $T(y, z)$ as in WanderJoin: we use sampling to find one answer to $\sigma_1(T(y, z)) = T(b, z)$. For example, we can randomly select $\sigma_2 = \{z \mapsto c_2\}$; there are two candidates, so we estimate the cardinality of $T(b, z)$ as 2, and we return the answer $\sigma_1 \cup \sigma_2$ and the cardinality estimate $6 \cdot 2 = 12$. As in WanderJoin, we can ignore $T(d, e)$ while processing $T(b, z)$ (provided adequate indexes are available), which increases the likelihood of a match.

We overestimated the cardinality of Q, since we explored just the first disjunct of Q_1. Choosing $i = 2$ leads to an underestimation of $1 \cdot 2 \cdot 2 = 4$. However, the expected value is 8, which is the correct cardinality—that is, our estimator is unbiased.

The cardinality estimate of Q is thus produced from the data that Algorithm 1 uses to produce just one answer, so individual estimates can vary considerably. However, by running the algorithm several times, we explore a larger portion of such answers. As the number of runs increases, the estimate average converges to the exact cardinality, and the variance converges to zero. We discuss how to select the number of runs in Section 5.5. ◁

Example 5.3. Let Q and I be the following query and database instance, respectively, so $\mathsf{ans}_I(Q)$ contains substitutions $\mu_1 = \{x \mapsto a\}$ and $\mu_2 = \{x \mapsto b\}$.

$$Q = A(x)\ \texttt{MINUS}\ R(x, y) \qquad\qquad I = \{A(a), A(b), A(c), R(c, d_1), R(c, d_2)\}$$

To estimate $|\mathsf{ans}_I(Q)|$, we again "sample the loops" of Algorithm 1: we randomly select one substitution σ_1 from line 14, and we check whether $\sigma_1(R(x, y))$ has any matches in I. The latter check uses sideways information passing, but it must be exact, since we must produce only valid answers. As in Algorithm 1, we can stop answering $\sigma_1(R(x, y))$ as soon as we find the first answer.

In our example, there are three ways to match $A(x)$. Thus, if we select μ_1 or μ_2, then the estimate is 3; and if we select $\mu_3 = \{x \mapsto c\}$, then the estimate is 0. Again, our estimator is unbiased. ◁

Queries of the form $Q = Q_1\ \texttt{FILTER}\ E$ can be estimated analogously to $Q = Q_1\ \texttt{MINUS}\ Q_2$: we filter the answers of Q_1 using E. Queries of the form $Q = Q_1\ \texttt{BIND}\ x := E$ can be handled in a similar way, so we next focus on the much more challenging DISTINCT operator.

Example 5.4. Let Q and I be the following query and database instance, respectively, so $\mathsf{ans}_I(Q)$ contains substitutions $\mu_1 = \{x \mapsto a\}$ and $\mu_2 = \{x \mapsto c\}$.

$$Q = \texttt{DISTINCT}(Q_1) \quad Q_1 = \texttt{PROJECT}_{\{x\}}(R(x, y))$$
$$I = \{R(a, b_1), \ldots, R(a, b_k), R(c, d)\}$$

Assume for the moment that we can "magically" associate μ_1 and μ_2 with the representative facts that produce these substitutions; for example, we can associate μ_1 with $R(a, b_1)$ and μ_2 with $R(c, d)$. We can then "sample the loops" of Algorithm 1 as follows: we guess a matcher σ_1 for $R(x, y)$ from $k + 1$ candidates; however, the guess is successful and we return the estimate of $k + 1$ only if we choose a representative, and we return 0 otherwise. The expectation is $(k + 1)\frac{2}{k+1} = 2$, so the estimator is unbiased. Several difficulties need to be addressed to make this idea practical.

First, the projection operator of Q_1 "erases" an association between selected facts and the resulting substitutions; for example, when sampling Q_1 returns substitution μ_1, we do not know whether μ_1 was obtained from $R(a, b_1)$ or $R(a, b_2)$. Analogously, in $\texttt{DISTINCT}(Q_1\ \texttt{UNION}\ Q_2)$, both Q_1 and Q_2 can produce the same substitution, but only one should count as a "success." To address this problem, our estimation algorithm returns an answer substitution for Q, a cardinality estimate, and an *outcome*—an object that uniquely encodes the choices that were used to obtain the answer. In our example, the outcome is simply the fact chosen to satisfy $R(x, y)$.

Second, the variance of the estimator can be large. A single run of this approach on Q and I can be modelled as an estimator $\hat{\theta} = (k + 1) \cdot X$, where X is a Bernoulli random variable with parameter $p = 2/(k + 1)$. It is known that $\mathbb{E}[X] = p$ and $\mathrm{Var}[X] = p(1 - p)$, so $\mathbb{E}[\hat{\theta}_1] = (k + 1) \cdot \mathbb{E}[X] = 2$ and $\mathrm{Var}[\hat{\theta}_1] = (k + 1)^2 \cdot \mathrm{Var}[X] = 2k - 2$. Thus, $\hat{\theta}_1$ is unbiased, but its variance grows with k. We can reduce the variance taking the average of n runs, which realises an estimator $\hat{\theta}_2 = \frac{k+1}{n} \cdot Y$, where Y is a binomial random variable with parameters n and the same p. It is known that $\mathbb{E}[Y] = np$ and $\mathrm{Var}[Y] = np(1 - p)$, so $\mathbb{E}[\hat{\theta}_2] = 2$ and $\mathrm{Var}[\hat{\theta}_2] = (2k - 2)/n$. The 95% confidence interval is thus $2 \pm 1.96\frac{\sqrt{2k-2}}{n}$, so we need least $1.4\sqrt{k - 1}$ runs to obtain the confidence interval that is at most as wide as the estimate itself. This observation echoes the formal result by Charikar et al. [15], who have proved that no estimator for DISTINCT queries can guarantee low error on all inputs unless it examines a large fraction of the input data. We show empirically in Section 7.3 that our approach is both accurate and efficient in practice.

Third, it is unclear how to associate μ_1 and μ_2 with representative facts without actually evaluating Q. We address this problem by maintaining a global mapping D^Q of the answers to Q_1 to unique outcomes. This mapping is initially empty, and it is updated in successive runs. In our example, if the first run produces substitution μ_1 due to the outcome $\omega = R(a, b_1)$, then $D^Q[\mu_1]$ is defined as ω. Thus, whenever μ_1 is produced in subsequent runs, our algorithm can determine whether this was achieved by $D^Q[\mu_1]$ or in some other way. Due to this change, individual estimator runs are no longer unbiased. For example, D^Q is initially empty so the first call of our estimator always returns $k + 1$; hence, the expectation of the first call is $k + 1$, rather than 2. Nevertheless, we prove that the sequence of averages of repeated calls is a *strongly consistent* estimator of the true cardinality. Hence, we can use our approach just like any unbiased estimator: as the number of runs increases, the estimate average converges to the query cardinality, and the variance converges to zero. ◁

5.3 The Basic Cardinality Estimation Approach

Algorithm 2 presents our cardinality estimation approach formally. Just like Algorithm 1, it takes as input a database instance I, a query Q, and a context substitution σ. The algorithm returns a triple $[\omega, \beta, c]$ that can have two forms. If sampling succeeds, then the triple is structured as follows.

- Component ω is an *outcome* object that describes all random choices made by all recursive calls, as motivated by Example 5.4.
- Component β is a substitution satisfying $\beta \in \mathsf{eval}_I(Q, \sigma)$.
- Component c is an estimate of $|\mathsf{eval}_I(Q, \sigma)|$.

Furthermore, the algorithm can indicate that it failed to identify an answer to Q by returning $[\bot, \emptyset, 0]$ where $\bot$ is a distinct *failure* outcome. If Q is a DISTINCT query, then the algorithm uses an initially empty *global* mapping D^Q of substitutions to outcomes. The structure of Algorithm 2 is similar to Algorithm 1, and it realises the idea of "loop sampling" from Section 5.2.

If Q is an atom A, then the algorithm randomly selects a matcher β of A to a fact in I. To capture different ways to achieve this, the algorithm is parameterised by a function sspace determining the *sample space*. Specifically, $\mathsf{sspace}_I(\sigma(A))$ should contain at least all facts of I that can be matched to $\sigma(A)$, but it is allowed to contain other facts as well. In practice, the sample space will

ALGORITHM 2: estimate$_I(Q, \sigma)$

Input: database instance I, query Q, and substitution σ with $\mathrm{dom}(\sigma) \subseteq \mathrm{v}(Q)$
Output: a triple $[\omega, \beta, c]$ where either $\omega = \bot$, $\beta = \emptyset$, and $c = 0$, or ω is an outcome, $\beta \in \mathrm{eval}_I(Q, \sigma)$, and c is an unbiased estimate of $|\mathrm{eval}_I(Q, \sigma)|$
Global: mapping D^Q of substitutions to outcomes unique for Q (initially empty)

1: **switch** Q
2: **case** $Q = A$
3: **if** $\mathrm{sspace}_I(\sigma(A)) \neq \emptyset$ **then**
4: Choose $F \in \mathrm{sspace}_I(\sigma(A))$ with prob. $\mathrm{P}(F)$
5: **if** a matcher β of $\sigma(A)$ to F exists **then**
6: **return** $[F, \sigma \cup \beta, 1/\mathrm{P}(F)]$
7: **case** $Q = Q_1$ AND Q_2
8: $[\omega_1, \sigma_1, c_1] := \mathrm{estimate}_I(Q_1, \sigma|_{Q_1})$
9: **if** $\omega_1 \neq \bot$ **then**
10: $[\omega_2, \sigma_2, c_2] := \mathrm{estimate}_I(Q_2, (\sigma \cup \sigma_1)|_{Q_2})$
11: **if** $\omega_2 \neq \bot$ **then**
12: **return** $[\langle \omega_1, \omega_2 \rangle, \sigma \cup \sigma_1 \cup \sigma_2, c_1 \cdot c_2]$
13: **case** $Q = Q_1$ UNION Q_2
14: Choose $i \in \{1, 2\}$ with prob. p_1 and p_2
15: $[\omega_i, \sigma_i, c_i] := \mathrm{estimate}_I(Q_i, \sigma)$
16: **if** $\omega_i \neq \bot$ **then**
17: **return** $[\langle i, \omega_i \rangle, \sigma_i, c_i/p_i]$
18: **case** $Q = Q_1$ MINUS Q_2
19: $[\omega_1, \sigma_1, c_1] := \mathrm{estimate}_I(Q_1, \sigma)$
20: **if** $\omega_1 \neq \bot$ and $\mathrm{eval}_I(Q_2, \sigma_1|_{Q_2}) = \emptyset$ **then**
21: **return** $[\omega_1, \sigma_1, c_1]$
22: **case** $Q = Q_1$ FILTER E
23: $[\omega_1, \sigma_1, c_1] := \mathrm{estimate}_I(Q_1, \sigma)$
24: **if** $\omega_1 \neq \bot$ and $\sigma_1(E) = \mathrm{true}$ **then**
25: **return** $[\omega_1, \sigma_1, c_1]$
26: **case** $Q = Q_1$ BIND $x := E$
27: $[\omega_1, \sigma_1, c_1] := \mathrm{estimate}_I(Q_1, \sigma|_{Q_1})$
28: **if** $\omega_1 \neq \bot$, $\sigma_1(E) \neq \epsilon$, and $\sigma \sim \{x \mapsto \sigma_1(E)\}$ **then**
29: **return** $[\omega_1, \sigma_1 \cup \{x \mapsto \sigma_1(E)\}, c_1]$
30: **case** $Q = \mathrm{PROJECT}_X(Q_1)$
31: $[\omega_1, \sigma_1, c_1] := \mathrm{estimate}_I(Q_1, \sigma)$
32: **if** $\omega_1 \neq \bot$ **then**
33: **return** $[\omega_1, \sigma_1|_X, c_1]$
34: **case** $Q = \mathrm{DISTINCT}(Q_1)$
35: $[\omega_1, \sigma_1, c_1] := \mathrm{estimate}_I(Q_1, \sigma)$
36: **if** $\omega_1 \neq \bot$ **then**
37: **if** $D^Q[\sigma_1]$ is undefined **then**
38: $D^Q[\sigma_1] := \omega_1$
39: **if** $D^Q[\sigma_1] = \omega_1$ **then**
40: **return** $[\omega_1, \sigma_1, c_1]$
41: **return** $[\bot, \emptyset, 0]$

be determined by the available indexes. For example, if $\sigma(A) = R(c, x)$ and the facts of relation R are indexed on the first position, then we can take $\mathrm{sspace}_I(\sigma(A))$ as all facts obtained by the index lookup for c. If, however, a precise index is unavailable, then $\mathrm{sspace}_I(\sigma(A))$ can be any suitable overestimate. For example, if no index can match $\sigma(A) = R(x, x)$ directly, then we can take $\mathrm{sspace}_I(\sigma(A)) = I(R)$. If the sample space is empty, then substitution β cannot be selected so the algorithm fails (line 3). Otherwise, the algorithm randomly selects a fact $F \in \mathrm{sspace}_I(\sigma(A))$ (line 4). Facts can be chosen according to an arbitrary but fixed probability distribution P on the sample space, which provides possible avenues for optimisation (e.g., selecting facts with frequently occurring constants more eagerly). Our implementation, however, chooses facts uniformly at random, so $\mathrm{P}(F) = 1/|\mathrm{sspace}_I(\sigma(A))|$. Once F is selected, the algorithm checks whether the selected fact F indeed matches $\sigma(A)$ via substitution β (line 5). If not, the algorithm fails, which is analogous to the final check in WanderJoin (see Example 3.1); otherwise, the algorithm returns substitution $\sigma \cup \beta$, unbiased estimate $1/\mathrm{P}(F)$, and outcome F indicating that $\sigma \cup \beta$ was obtained by selecting F.

The remaining operators are handled as outlined in Section 5.2: conjunctions use sideways information passing in a way that mimics WanderJoin, and all operators can be seen as "sampling the loops" of Algorithm 1. For the sake of generality, the two disjuncts of $Q = Q_1$ UNION Q_2 are explored with arbitrary probabilities p_1 and p_2; however, $p_1 = p_2 = 0.5$ is likely to be sufficient for practice. Sampling a subquery can produce a substitution that does not satisfy the relevant conditions. For example, for $Q = Q_1$ FILTER E, line 19 can produce an answer σ_1 of Q_1 that does not satisfy E. In all such cases, the algorithm indicates failure by returning in line 41, which prevents

any further sideways information passing. Just like in WanderJoin, when calling the algorithm repeatedly, failures must be counted as estimates of zero cardinality.

Theorem 5.5 captures the formal properties of Algorithm 2, and it is proved in Appendix A. Since $\mathsf{ans}_I(Q) = \mathsf{eval}_I(Q, \emptyset)$, we can estimate the cardinality of Q by using an empty context substitution.

THEOREM 5.5. *Let $\hat{\theta}_1, \hat{\theta}_2, \ldots$ be the sequence of random variables representing the third component of the results of successive calls to* $\mathsf{estimate}_I(Q, \sigma)$ *for some I, Q, and σ with* $\mathsf{dom}(\sigma) \subseteq \mathsf{v}(Q)$.

— *The sequence of averages $\frac{1}{n} \cdot \sum_{i=1}^{n} \hat{\theta}_i$ is a strongly consistent estimator of $|\mathsf{eval}_I(Q, \sigma)|$.*

— *If Q does not contain* `DISTINCT`*, then each $\hat{\theta}_i$ is an unbiased estimator of $|\mathsf{eval}_I(Q, \sigma)|$.*

To prove Theorem 5.5, we first show that, if mappings D^Q used to handle `DISTINCT` queries are preinitialised so the check in line 37 is never satisfied (i.e., $D^Q[\sigma_1]$ is always defined), then all $\hat{\theta}_i$ are unbiased. We prove the latter claim inductively, but conjunctions pose a problem that we discuss next. Assume that Algorithm 2 is called for $Q = Q_1$ `AND` Q_2 and some σ, and that the recursive call for each Q_i with $i \in \{1, 2\}$ produces σ_i and an unbiased estimate $\hat{C}_i(\sigma_i)$ with probability $\mathrm{P}_i(\sigma_i)$. The expectation of the estimate $\hat{C}(\sigma)$ of $|\mathsf{eval}_I(Q, \sigma)|$ can then be computed as follows, where σ_1 and σ_2 range over $\mathsf{eval}_I(Q_1, \sigma|_{Q_1})$ and $\mathsf{eval}_I(Q_2, (\sigma \cup \sigma_1)|_{Q_2})$, respectively.

$$\mathbb{E}[\hat{C}(\sigma)] = \sum_{\sigma_1} \sum_{\sigma_2} \mathrm{P}_1(\sigma_1) \cdot \mathrm{P}_2(\sigma_2) \cdot \hat{C}_1(\sigma_1) \cdot \hat{C}_2(\sigma_2) = \tag{9}$$

$$= \sum_{\sigma_1} \mathrm{P}_1(\sigma_1) \cdot \hat{C}_1(\sigma_1) \cdot \sum_{\sigma_2} \mathrm{P}_2(\sigma_2) \cdot \hat{C}_2(\sigma_2) = \tag{10}$$

$$= \sum_{\sigma_1} \mathrm{P}_1(\sigma_1) \cdot \hat{C}_1(\sigma_1) \cdot \mathbb{E}[\hat{C}_2((\sigma \cup \sigma_1)|_{Q_2})] = \tag{11}$$

$$= \sum_{\sigma_1} \mathrm{P}_1(\sigma_1) \cdot \hat{C}_1(\sigma_1) \cdot |\mathsf{eval}_I(Q_2, (\sigma \cup \sigma_1)|_{Q_2})| \tag{12}$$

Equality of Equations (10) and (11) follows from the definition of the expectation, and the equality of Equations (11) and (12) follows from the inductive assumption that estimates for Q_2 and $(\sigma \cup \sigma_1)|_{Q_2}$ are unbiased. However, $|\mathsf{eval}_I(Q_2, (\sigma \cup \sigma_1)|_{Q_2})|$ depends on σ_1, so we cannot apply analogous reasoning to Q_1. We address this problem by showing that Algorithm 2 in fact realises a Horvitz–Thompson estimator: its estimates are not only unbiased, but they also satisfy $\hat{C}(\sigma) = 1/\mathrm{P}(\sigma)$. Thus, terms $\mathrm{P}_1(\sigma_1)$ and $\hat{C}_1(\sigma_1)$ in Equation (12) cancel out, so we can continue the calculation as follows.

$$\mathbb{E}[\hat{C}(\sigma)] = \sum_{\sigma_1} |\mathsf{eval}_I(Q_2, (\sigma \cup \sigma_1)|_{Q_2})| = |\mathsf{eval}_I(Q, \sigma)| \tag{13}$$

For the general case when mappings D^Q are not preinitialised, we first show that, with probability one, successive invocations populate each D^Q so $D^Q[\sigma_1]$ is defined for all relevant σ_1. Thus, after sufficiently many "warm-up" runs, our estimator starts producing unbiased estimates as argued in the previous paragraph, and so the average of all estimates obtained from this point onwards converges to the actual cardinality with probability one. Moreover, the number of "warm-up" runs is finite, so the bias introduced by these runs converges to zero as the number of runs increases.

We finish this section by a brief discussion of how to extend Algorithm 2 to features of graph query languages not included in our definition from Section 2.1. Example 5.6 shows that certain forms of aggregation queries can be challenging.

Example 5.6. Consider a SPARQL query of the following form, where the result of aggregation is joined with another subquery `<Q2>`. Whether the cardinality of this query can be estimated using Algorithm 2 depends on whether variable `?Z` occurs in `<Q2>`.

```
SELECT * WHERE { { SELECT ?X (SUM(?Y) AS ?Z) WHERE { ?X :hasTemp ?Y } GROUP BY ?X } . <Q2> }
```

If ?Z does not occur in <Q2>, then this query returns the same number of answers as the following query. Hence, the exact result of aggregation is irrelevant, so we can transform the query into one that Algorithm 2 can handle.

```
SELECT * WHERE { { SELECT DISTINCT ?X WHERE { ?X :hasTemp ?Y } } . <Q2> }
```

In contrast, if ?Z occurs in <Q2>, then the value of ?Z must be computed *exactly* if it is to join with <Q2>. This is analogous to Q_1 MINUS Q_2, where $\text{eval}_I(Q_2, \sigma_1|_{Q_2}) = \emptyset$ in line 20 of Algorithm 2 must be checked exactly. Our algorithm can still be used: we can fix the value of ?X by guessing a match for atom ?X :hasTemp ?Y to some fact and then compute the corresponding value of ?Z by evaluating the aggregation exactly for the fixed value of ?X. Depending on the size of the group for ?X, this may or may not introduce unacceptable overheads.

Moreover, if the aggregation function is changed to MIN, then we can avoid evaluating one group in its entirety: we randomly select a fact matching ?X :hasTemp ?Y, and we check whether the value for ?Y is indeed minimal for ?X; if the database instance is indexed appropriately, then this can be more efficient than evaluating the aggregate subquery in full. The main risk is that the rate of failure (i.e., guesses that do not lead to a solution) of such an approach can be high.

To summarise, cardinality estimation for queries with nested aggregation can be hard, and it remains to be seen whether any of the approaches outlined above are practical. We also point out that the WanderJoin algorithm by Li et al. [47] can handle GROUP BY queries, but, instead of estimating the number of groups, its objective is to estimate the aggregation value *for each group*. ◁

Graph query languages often support *conjunctive regular path queries* (CRPQs) [7], where atoms can have the form $re(s, t)$ for re a regular expression over binary relations. A substitution σ is an answer to $re(s, t)$ on I if there exists a word $R_1 \dots R_n$ in the regular language of re and facts $\{R_1(c_0, c_1), \dots, R_n(c_{n-1}, c_n)\} \subseteq I$ such that $\sigma(s) = c_0$ and $\sigma(t) = c_n$. Atom $re(s, t)$ is semantically equivalent to $Q = \text{DISTINCT}(\text{UNION}(w_1(s, t), w_2(s, t), \dots))$, where $w_1, w_2, \dots$ are all words of the language of re. Now, even if this union is infinite, Algorithm 2 can be applied to Q provided that disjuncts are selected using probabilities that add up to one. Hence, extending Algorithm 2 to CRPQs seems feasible in principle, and we shall develop this idea further in our future work.

Finally, graph query languages often support sorting, but this does not affect query cardinality. Sorting is sometimes combined with OFFSET/LIMIT operators to select a subset of query answers, and we do not see how to incorporate such queries into our framework.

5.4 Optimising the Basic Approach

A closer look at Algorithm 2 reveals that substitutions produced by *tail-recursive* calls are not used for sideways information passing. Moreover, the Horvitz–Thompson property is used to transform Equation (12) into (13), but *not* to transform Equation (11) into (12): $\hat{C}_2(\sigma_2)$ is only required to be unbiased. Consequently, we can optimise tail-recursive calls to return unbiased, but not necessarily Horvitz–Thompson, estimates. Examples 5.7 and 5.8 motivate such optimisations.

Example 5.7. Let $Q = R(x, y)$ AND $S(y, z)$ and $I = \{R(a_i, b_i) \mid 1 \le i \le k\} \cup \{S(b_1, c_1)\}$ for $k \ge 1$. When Algorithm 2 is applied n times to Q and I, each run is independent, so the probability of obtaining a nonzero estimate after n runs is $p_1 = 1 - (1 - 1/k)^n$—that is, the complement of the probability of not selecting $R(a_1, b_1)$ in any of the n runs.

We can improve this by sampling $I(R)$ without replacement and thus exploring a larger portion of the sample space. For example, we can partition $I(R)$ into n blocks of k/n facts, sample each block independently, and sum the resulting estimates. The probability of a nonzero estimate is

ALGORITHM 3: $\text{estimate}_I^{opt}(Q, \sigma)$

Input: database instance I, query Q, and substitution σ with $\text{dom}(\sigma) \subseteq \text{v}(Q)$
Output: an unbiased estimate of $|\text{eval}_I(Q, \sigma)|$

```
1:  C := 0
2:  switch Q
3:  case Q = Q1 AND Q2 where Q1 is an atom A1
4:    Partition sspace_I(σ(A1)) into nonempty subsets S1, ..., SN
5:    for 1 ≤ i ≤ N do
6:      Choose F ∈ Si with probability P(F)
7:      if a matcher β of σ(A1) to F exists then
8:        C := C + estimate_I^opt(Q2, (σ ∪ β)|_Q2)/P(F)
9:  case Q = Q1 AND Q2 where Q1 is not an atom
10:   [ω1, σ1, c1] := estimate_I(Q1, σ|_Q1)
11:   if ω1 ≠ ⊥ then
12:     C := c1 · estimate_I^opt(Q2, (σ ∪ σ1)|_Q2)
13: case Q = Q1 UNION Q2
14:   for 1 ≤ i ≤ 2 do
15:     C := C + estimate_I^opt(Qi, σ)
16: case Q = PROJECT_X(Q1)
17:   C := estimate_I^opt(Q1, σ)
18: otherwise
19:   [ω, σ′, c] := estimate_I(Q, σ)
20:   C := c
21: return C
```

then $p_2 = n/k$—that is, the probability of choosing $R(a_1, b_1)$ from k/n facts. One can verify that $p_2 \geq p_1$ for all n and k, and that the difference between p_1 and p_2 is larger when k and n are of similar orders of magnitude. ◁

Example 5.8. Given $Q = Q_1$ UNION Q_2, Algorithm 2 explores either Q_1 or Q_2, but never both. Now assume that the algorithm can estimate the cardinality of Q_1 and Q_2 correctly. The space of possible estimates after n runs is $\frac{n_1}{n}|\text{ans}_I(Q_1)| + (1 - \frac{n_1}{n})|\text{ans}_I(Q_2)|$ for each n_1 between 0 and n. However, $|\text{ans}_I(Q)| = |\text{ans}_I(Q_1)| + |\text{ans}_I(Q_2)|$, and the sum of unbiased estimators is an unbiased estimator of the sum; hence, we can estimate Q correctly independently of n by just adding the estimates of Q_1 and Q_2. Intuitively, eliminating the choice in line 14 of Algorithm 2 reduces randomness and thus decreases the estimator's variance. ◁

Function $\text{estimate}_I^{opt}(Q, \sigma)$ shown in Algorithm 3 uses this idea. Unlike Algorithm 2, it returns a cardinality estimate, but not an outcome or a substitution. For $Q = A_1$ AND Q_2 where A_1 is an atom, the algorithm partitions the sampling space of $\sigma(A_1)$ into nonempty disjoint subsets $\mathcal{S}_1, \ldots, \mathcal{S}_N$ and samples each $\mathcal{S}_i$ independently. The answers to Q where A_1 is matched in $\mathcal{S}_i$ and $\mathcal{S}_j$ are disjoint for all $i \neq j$, so an unbiased cardinality estimate can be obtained by summing the estimates of Q over all partitions (line 8). For $Q = Q_1$ AND Q_2 where Q_1 is not an atom, the algorithm estimates Q_2 by calling itself (line 12); in contrast, Q_1 is estimated using the unoptimised algorithm (line 10) to produce a substitution σ_1 that can be passed sideways to Q_2. For $Q = Q_1$ UNION Q_2, the algorithm adds the optimised estimates of Q_1 and Q_2 (line 15). For $Q = \text{PROJECT}_X(Q_1)$, the algorithm simply returns the optimised estimate of Q_1 (line 17). Finally, if Q of any other type, then the algorithm falls back to the original algorithm (line 19). For example, for $Q = Q_1$ FILTER E, subquery Q_1 must produce a single substitution where expression E can be evaluated; for $Q = Q_1$ MINUS Q_2, subquery Q_1 must produce a substitution that can be passed sideways to Q_2, and so on.

Theorem 5.9 summarises the formal properties of Algorithm 3, and its proof is provided in full in Appendix B.

THEOREM 5.9. *Let $\hat{\theta}_1, \hat{\theta}_2, \ldots$ be the sequence of random variables representing the results of successive calls to* $\text{estimate}_I^{opt}(Q, \sigma)$ *for some I, Q, and σ with* $\text{dom}(\sigma) \subseteq \text{v}(Q)$.

— *The sequence of averages $\frac{1}{n} \cdot \sum_{i=1}^{n} \hat{\theta}_i$ is a strongly consistent estimator of $|\text{eval}_I(Q, \sigma)|$.*
— *If Q does not contain* DISTINCT, *then each $\hat{\theta}_i$ is an unbiased estimator of $|\text{eval}_I(Q, \sigma)|$.*

5.5 Practical Considerations

We next discuss several issues that must be addressed to make Algorithms 2 and 3 practical.

Enumeration of the Relevant Orders. As we explained in Section 3.2, the order of atoms in a conjunction profoundly affects the estimate variance, which determines estimation accuracy. However, identifying an optimal order in advance can be challenging. Given $Q = \mathtt{AND}(A_1, \ldots, A_n)$, the WanderJoin variant from the G-CARE framework takes NR independent estimates using all permutations of the atoms of Q. Example 5.10 shows that this can be inefficient.

Example 5.10. Let $Q = \mathtt{AND}(R_1(x, y_1), \ldots, R_n(x, y_n))$; such Q is commonly called a *star* query, since the atoms of Q connect variables y_i to the central variable x in a star-like fashion. There are $n!$ different permutations of the atoms of Q, each of which is reasonable in the sense that it does not introduce a cross-product into the join. The Yago benchmark from the G-CARE framework contains 80 such queries where $n = 12$, each giving rise to more than 479 million permutations.

However, only the choice of the first atom determines the variance of the resulting estimator. Assume that Q is ordered as shown in the previous paragraph. Once we select a fact matching $R_1(x, y_1)$, this determines the value of variable x in all remaining atoms; moreover, the remaining atoms do not share other variables, so choosing a fact for atom $R_i(x, y_i)$ with $i \geq 2$ does not impact the choices for any atom $R_j(x, y_j)$ with $j > i$. Thus, it only makes sense to consider n orderings of Q, each starting with a distinct atom of Q, and to order the remaining atoms arbitrarily. ◁

We can apply this idea to an arbitrary conjunction as follows. When considering an order $Q = \mathtt{AND}(Q_1, \ldots, Q_n)$, we annotate each Q_i with the set of variables that will be bound when Q_1 is called. Moreover, whenever an order enumeration procedure produces orders Q' and Q'' where all corresponding conjuncts are annotated with the same sets of variables, we keep either Q' or Q''.

Selecting a Single Order. We next describe a simple way to order the atoms of a conjunction that is likely to reduce the estimator's variance. Our idea is based on an observation that the variance is usually related to the sample space size in line 3 of Algorithm 2: a larger sample space provides more ways to constrain the rest of the query, so, unless the data distribution is symmetric, choosing different facts usually leads to different cardinality estimates. One can thus expect to reduce the estimator's variance by minimising the number of choices available at each step.

To estimate the size of the sample spaces, our algorithm relies on very simple statistics about the data. In particular, for each n-ary relation R and each subset $S \subseteq \{1, \ldots, n\}$, we precompute

$$R^S = \frac{|\mathsf{ans}_I(R(x_1, \ldots, x_n))|}{|\mathsf{ans}_I(\mathtt{DISTINCT}(\mathtt{PROJECT}_{\{x_i \mid i \in S\}}(R(x_1, \ldots, x_n))))|}. \tag{14}$$

In other words, R^S is the average number of facts of $I(R)$ when the values for the arguments with indexes in S are fixed. Algorithm 4 uses this information to order a set of atoms. We assume that the atoms are connected; otherwise, we can apply the algorithm to each connected component separately. The algorithm uses a simple greedy strategy. In lines 2–7, the algorithm considers each A_i as a possible first atom. Set V is used to keep track of bound variables and is initialised to $\mathsf{v}(A_i)$ in line 3. Next, the algorithm extends the candidate order in lines 4–6. At each step, the algorithm selects an unprocessed atom A_j that does not introduce a cross-product. If there are several such atoms, then A_j is greedily selected to minimise the average number of matches for the variables in V. The cost of each candidate order is the product of the costs of all atoms. Finally, line 7 ensures that the order for the starting atom with the least overall cost is returned.

Algorithm 4 is reminiscent of greedy join ordering algorithms, and the used cost can be seen as a cardinality estimation obtained by ad hoc assumptions from Section 1. However, unlike the existing approaches that require complex statistics about the database instance (e.g., for the approach by

ALGORITHM 4: order–by–fanout$_I(A_1, \ldots, A_n)$

Input: database instance I and a conjunction of atoms $A_1, \ldots, A_n$
Output: a reordered conjunction computed using the fanout heuristic

1: *BestOrder* := *nil*, *BestCost* := ∞
2: **for each** $1 \leq i \leq n$ **do**
3: *Order* := $[A_i]$, *Cost* := cost–fanout$(A_i, \emptyset)$, $V := \mathsf{v}(A_i)$
4: **while** $|Order| \neq n$ **do**
5: Choose A_j in $\{A_k \mid A_k \notin Order \text{ and } \mathsf{v}(A_k) \cap V \neq \emptyset\}$ with least cost–fanout(A_j, V)
6: Extend *Order* with A_j, *Cost* := *Cost* $\cdot$ cost–fanout(A_j, V), and $V := V \cup \mathsf{v}(A_j)$
7: **if** *OrderCost* < *BestCost* **then** *BestOrder* := *Order* and *BestCost* := *OrderCost*
8: **return** *BestOrder*

9: **function** cost–fanout$(R(t_1, \ldots, t_n), V)$
10: **return** R^S where $S := \{i \mid 1 \leq i \leq n \text{ and } t_i \text{ is a constant or } t_i \in V\}$

Chen et al. [16], the cardinality of joins of pairs of relations must be known), Algorithm 4 requires only limited information about each relation. The resulting cost can thus be vastly different from the actual query cardinality, and the resulting orders can be suboptimal. This, however, is compensated by Algorithms 2 and 3 that produce much more accurate cardinality estimates, as well as the query planning algorithm we present in Section 6. We show empirically in Section 7 that the resulting query plans can sometimes be significantly more efficient, particularly on complex queries, but without incurring a substantial overhead for query planning on simpler queries.

Dynamic Stopping Condition. In Section 4, we argued that the number of runs of an estimation algorithms should ideally not depend on the input size. Instead, we determine the number of runs dynamically similarly to online aggregation algorithms. In particular, we fix the target q-error ($q\text{-}err_t$) and the minimum (N_{min}) and maximum (N_{max}) numbers of runs. After each run, we compute the mean $\bar{t}$ and the variance S of the n estimates collected thus far. We stop the process if $n = N_{max}$ (which ensures termination on queries with zero cardinality), or if $n \geq N_{min}$, $\bar{t} > 0$ (i.e., at least one run produced a nonzero estimate), and $\bar{t} + 1.96 \cdot S/\sqrt{n} \leq \bar{t} \cdot q\text{-}err_t$ (i.e., the upper end of the 95% confidence interval falls within the target q-error range).

Partitioning the Sample Space in Line 4 of Algorithm 3. We partition the sample space into blocks of fixed *partition size* p. The number of partitions thus depends on the size of the sample space, so the number of samples taken can, in some cases, depend on the input size.

Combining Algorithms 2 and 3. As we discuss in Section 7, Algorithm 2 sometimes returns zero estimates on more complex queries. Partitioning in line 4 of Algorithm 3 can improve the likelihood of finding a nonzero estimate, but it can also increase the running time. These observations motivate the following combined approach. We first try to obtain a nonzero estimate using the basic algorithm and the dynamic stopping condition for some N^b_{min}, N^b_{max}, and $q\text{-}err_t$. If this produces a zero estimate, we disregard all collected samples and we repeat the process using the optimised algorithm for some N^o_{min} and N^o_{max} and the same $q\text{-}err_t$. The estimation time thus depends on the input graph size for complex queries only, which are hopefully rare. Moreover, since the optimised algorithm examines substantially more facts, we use N^o_{min} and N^o_{max} different from N^b_{min} and N^b_{max} to limit the overall amount of work.

Dependency-Directed Backtracking. On query $Q = \mathtt{AND}(R_1(x, y_1), R_2(x, y_2), R_3(x, y_3))$ and database instance $I = \{R_1(a, b), R_2(a, c_1), \ldots, R_2(a, c_n), R_3(d, e)\}$, Algorithms 1 and 3 match $R_1(x, y_1)$ to $R_1(a, b)$, and then match $R_2(x, y_2)$ to each $R_2(a, c_i)$ with $1 \leq i \leq n$, only to find that $R_3(x, y_3)$ cannot

ALGORITHM 5: order−DP$_I(A_1, \ldots, A_n)$

Input: database instance I and a conjunction of atoms $A_1, \ldots, A_n$
Output: a reordered conjunction where cost is computed using Algorithms 2 and 3

1: $P := \emptyset$
2: **for each** $1 \leq i \leq n$ **do**
3: $\quad P[\{A_i\}] = \langle A_i, \text{estimate}_I(A_i, \emptyset)\rangle$
4: **for each** $2 \leq \ell \leq n$ **do**
5: $\quad P' := \emptyset$
6: $\quad$ **for each** $\langle Atoms, \langle Order, Cost\rangle\rangle \in P$ **do**
7: $\quad\quad$ **for each** $1 \leq j \leq n$ such that $A_j \notin Atoms$ and $\mathsf{v}(A_j) \cap \mathsf{v}(Atoms) \neq \emptyset$ **do**
8: $\quad\quad\quad Atoms' := Atoms \cup \{A_j\}$
9: $\quad\quad\quad Order'$:= $Order$ extended with A_j
10: $\quad\quad\quad Cost' := Cost + \text{cost}_I(Atoms')$
11: $\quad\quad\quad$ **if** $P'[Atoms']$ is undefined or $Cost' < P'[Atoms'].Cost$ **then**
12: $\quad\quad\quad\quad P'[Atoms'] := \langle Order', Cost'\rangle$
13: $\quad P := P'$
14: $\quad$ Remove from P all but k orders with the least cost
15: **return** the order in P with the least cost

16: **function** cost$_I(Atoms)$
17: $\quad Q :=$ AND(order−by−fanout$_I(Atoms)$)
18: $\quad$ **return** the estimate of $|\text{ans}_I(Q)|$ produced using the combination of Algorithms 2 and 3 from Section 5.5 parameterised by N^b_{min}, N^b_{max}, N^o_{min}, N^o_{max}, and $q\text{-}err_t$

be matched. However, exploring all $R_2(a, c_i)$ is superfluous: The value of variable x in $R_3(x, y_3)$ is determined by $R_1(x, y_1)$ and is independent from the match to $R_2(x, y_2)$. Thus, when matching $R_3(a, y_3)$ fails, we can backtrack to $R_1(x, y_1)$ and attempt to match this atom differently.

More generally, when atom $\sigma(A)$ has no matches in line 3 of Algorithm 1, we can backtrack to the most recent atom in the conjunction that provided a binding for $\sigma(A)$; the case when $\text{sspace}_I(\sigma(A_1))$ in line 4 of Algorithm 3 is empty is analogous. Similar techniques are widely used to solve hard combinatorial problems such as propositional satisfiability.

6 Integrating Cardinality Estimation into Query Planning

An important question is whether, by providing accurate cardinality estimates, our algorithms can improve query plans in ways that significantly reduce end-to-end query evaluation times. Most query planners are based on variants of *dynamic programming (DP)*. Thus, in Algorithm 5, we present a simple DP-based planner for conjunctive queries whose plans can be evaluated using the query evaluation approach from Algorithm 1. This algorithm follows closely the general principles for DP-based planners, and we present it mainly to clarify all relevant details. In Section 7.4, we then show empirically that such an approach can indeed significantly benefit end-to-end query evaluation, particularly when queries are complex.

The algorithm takes a connected set of atoms, and it returns an ordering optimised for evaluation using Algorithm 1 from Section 5.1. The algorithm follows a standard dynamic programming approach. In particular, it maintains mappings P and P' of sets of atoms to pairs of an order and the corresponding cost. Mapping P is initialised in lines 2 and 3 to all orders consisting of a single atom. Then, the loop in lines 4–14 iteratively extends each *Order* in P with one additional atom. Condition $\mathsf{v}(A_j) \cap \mathsf{v}(Atoms) \neq \emptyset$ in line 7 ensures that extending *Order* with A_j does not result in a cross-product. After extending *Order* with A_j in line 9, the cost of the new order is computed

in line 10 and, if the resulting combination of atoms has not been seen before or the new cost is smaller (line 11), then the new order is recorded in P' (line 12). To further optimise the process, only the best k orders are kept after each iteration (line 14). Finally, the best order is returned in line 15.

Minimising the number of substitutions in lines 6 and 7 seems like an obvious way to optimise the evaluation of conjunctions using Algorithm 1, so we define the cost of an order $A_1, \ldots, A_m$ as

$$\sum_{i=1}^{m} |\mathsf{ans}_I(\mathsf{AND}(A_1, \ldots, A_i))|. \tag{15}$$

This is reflected in line 10 of Algorithm 5: the cost of *Order*′ is the sum of the cost of *Order* and the estimate of the cardinality of *Order*′. The latter is computed by ordering the plan's atoms using Algorithm 4 and then estimating the cardinality using the combined approach from Section 5.5.

While reordering in line 17 can be seen as "query planning for query planning," we found it essential to obtaining accurate, nonzero cardinality estimates on the benchmarks from Section 7. The results of our experiments show that Algorithm 5 incurs modest overheads on most queries, and that, particularly on complex queries, it can produce plans that can be much more efficient than the ones obtained by the simple reordering approach.

Finally, the number of query answers does not depend on the atom order, so, in the last iteration (i.e., when $\ell = n$ in line 4), the cardinality of all *Order*′ in line 10 should be the same. Consequently, without calling the estimation algorithm on the full query, we can identify the best plan after $n - 1$ iterations and simply extend it to the full plan with one missing atom.

7 Experimental Evaluation

We now present the results of our empirical evaluation. In Section 7.1 we describe our test setting; in Sections 7.2 and 7.3, we evaluate the accuracy and efficiency of our algorithms on conjunctive and complex queries, respectively; in Section 7.4 we evaluate the algorithm from Section 6 end-to-end by analysing total times that include both query planning and query evaluation; and in Section 7.5 we compare our work to NeuroCard [71], an influential cardinality estimation approach based on deep learning. All code, datasets, and experimental results are available online [37].

7.1 Test Setting

We used the six datasets from Section 4, which we extended with the IMDB dataset from the NeuroCard study. IMDB consists of 74.2 M facts distributed over 21 relations of arity between two and 12, as well as 70 `job-light` and 1,000 `job-light-ranges` queries consisting of conjunctions of atoms and `FILTER` conditions. The minimum and maximum cardinalities are 1 and 233,657,819,759, respectively, and the cardinality of 462 queries is less or equal to 10,000.

We developed a prototype system that can load a database instance into RAM, evaluate a query exactly using Algorithm 1, or estimate the query cardinality using one of the following variants.

- The Basic variant uses Algorithm 2 with the dynamic stopping condition from Section 5.5. Conjunctive queries are reordered using Algorithm 4, while complex queries are processed exactly as given in the input.
- The Opt variant uses Algorithm 3 with the same stopping condition and reordering. Partition size is fixed to $p = 32$.
- The Comb variants implements the combined approach from Section 5.5.
- The Ord-Fix variant optimises the wj algorithm in the G-CARE framework: it enumerates all orders as described in Section 5.5, computes *NR* using Equation (5), and runs Algorithm 2

exactly *NR* times. All orders are considered in a round-robin fashion until one order produces 100 nonzero estimates, and the remaining runs are done with an order having the least variance among orders that accumulated at least 50 nonzero estimates. Since the number of runs is generally quite high, we do not repeat this process 30 times.

— Ord-Var variant is analogous to Ord-Fix, but, instead of taking a fixed number of samples, it uses the dynamic stopping condition from Section 5.5.

The dynamic stopping condition uses $q\text{-}err_t = 10$, $N_{min} = 30$, and $N_{max} = 10,000$ in most cases. The only exception is Opt on conjunctive queries: partitioning the relation matching the first atom can be seen as introducing additional runs, so, in this case, we use $N_{min} = 1$ and $N_{max} = 100$.

Our system was developed in C++20. It can read data from RDF Turtle files or from CSVs; in the former case, RDF triples are transformed into a relational form using vertical partitioning (see Section 2.1). Unary and binary relations are indexed exhaustively after loading; for example, for a binary relation R, the system creates a hash table mapping each constant a to a vector of all facts of the form $R(a, b)$, an analogous hash table for the second argument, a hash table over both arguments of R, and a vector of all facts of the form $R(a, a)$. For relations of arity higher than two, only indexes needed to evaluate the benchmark queries are created. Our system can process SPARQL queries that can be translated into the algebra from Section 2.1. We extended the syntax of SPARQL with the ability to refer to n-ary atoms in queries.

We used the server described in Section 4. For each benchmark, we loaded the dataset and computed the exact and the estimated cardinalities of all queries using the relevant algorithms. We recorded the wall-clock time of each task, as well as the number of runs of Algorithm 2 or 3. Moreover, to compare the work performed by different algorithms, we also recorded the number of *matches*—that is, the number of times a fact is matched to an atom in line 3 of Algorithm 1, line 5 of Algorithm 2, or line 7 of Algorithm 3. For each query and algorithm, we computed the ratio of the number of matches for exact evaluation and for estimation. Thus, a ratio larger than one indicates that the estimation algorithm performs less work than the exact algorithm.

7.2 Cardinality Estimation of Conjunctive Queries

Figure 3 summarises our results for conjunctive queries with nonzero cardinality. For each benchmark, we report the number of queries and the total time for exact query evaluation. For each estimation algorithm, we report the total estimation time ("Total time"), the number of queries where estimation takes longer than exact evaluation ("# > exact"), the numbers of queries on which the Q-error is infinite ("# $q\text{-}err = \infty$") and larger than 10 ("# $q\text{-}err > 10$"), and the maximum q-error different from ∞ ("Max $q\text{-}err \neq \infty$"). Figure 4 shows the distributions of the q-errors, estimation times, and the match ratios. We discuss the 18 queries with zero cardinality separately.

Efficiency. In most cases, the total estimation time is considerably lower than the total exact evaluation time: the only exception is Ord-Fix on Human, but that dataset is very small so all queries are trivial. However, Ord-Fix is always slower than all other techniques, often by orders of magnitude, and this difference is not limited to just the most complex queries: estimation times are much higher for Ord-Fix even for the first quartile of queries of LUBM-01K-mat, DBLP, and IMDB. In fact, Ord-Fix seems to perform roughly like wj from Section 4 if we take into account that wj repeats the estimation process 30 times. The matches ratio for Ord-Fix shows that the technique performs more work than exact query evaluation for at least 25% of all queries on all benchmarks, and even up to half of the queries of Human, WatDiv, DBLP, and IMDB.

Techniques other than Ord-Fix all use the dynamic stopping condition from Section 5.5. In fact, Basic, Comb, and Ord-Var perform roughly the same amount of matches in roughly the same amount of time, which we attribute to the fact that N_{min} and N_{max} are the same in all cases. The Opt variant is generally between these three techniques and Ord-Fix, which is unsurprising: due

AIDS

# of queries with nonzero cardinality: 780					
Total time for exact evaluation: 22,252 ms					
	Basic	Opt	Comb	Ord-Var	Ord-Fix
Total time (ms)	29	221	43	70	485
# > exact	0	1	0	3	120
# $q\text{-}err = \infty$	11	3	1	26	27
# $q\text{-}err > 10$	62	36	54	193	52
Max $q\text{-}err \neq \infty$	225.2	3,946.5	238.7	40,982.1	773.1

Human

# of queries with nonzero cardinality: 49					
Total time for exact evaluation: 3 ms					
	Basic	Opt	Comb	Ord-Var	Ord-Fix
Total time (ms)	0	0	0	0	4
# > exact	0	0	0	0	4
# $q\text{-}err = \infty$	1	2	1	6	3
# $q\text{-}err > 10$	1	4	1	8	4
Max $q\text{-}err \neq \infty$	4.8	334.4	4.8	55.0	10.9

Yago

# of queries with nonzero cardinality: 1,365					
Total time for exact evaluation: 331,649 ms					
	Basic	Opt	Comb	Ord-Var	Ord-Fix
Total time (ms)	275	2,816	2,171	685	7,128
# > exact	18	64	68	105	449
# $q\text{-}err = \infty$	267	146	132	419	438
# $q\text{-}err > 10$	385	228	280	597	490
Max $q\text{-}err \neq \infty$	5,086.0	10,846.5	5,086.0	67,127.3	10,073.3

LUBM-01K-mat

# of queries with nonzero cardinality: 36					
Total time for exact evaluation: 10,857 ms					
	Basic	Opt	Comb	Ord-Var	Ord-Fix
Total time (ms)	5	782	5	8	2,049
# > exact	0	0	0	1	19
# $q\text{-}err = \infty$	0	1	0	0	0
# $q\text{-}err > 10$	1	1	1	0	1
Max $q\text{-}err \neq \infty$	42.8	8.9	42.8	8.0	12.0

WatDiv

# of queries with nonzero cardinality: 86					
Total time for exact evaluation: 301 ms					
	Basic	Opt	Comb	Ord-Var	Ord-Fix
Total time (ms)	7	96	76	10	166
# > exact	0	1	1	3	37
# $q\text{-}err = \infty$	1	3	1	6	6
# $q\text{-}err > 10$	4	4	4	16	7
Max $q\text{-}err \neq \infty$	43.8	14.0	43.8	37.5	12.4

DBLP

# of queries with nonzero cardinality: 15					
Total time for exact evaluation: 511 ms					
	Basic	Opt	Comb	Ord-Var	Ord-Fix
Total time (ms)	1	8	1	5	162
# > exact	0	0	0	1	11
# $q\text{-}err = \infty$	1	0	1	2	2
# $q\text{-}err > 10$	2	0	2	3	2
Max $q\text{-}err \neq \infty$	32.7	2.4	32.7	10.6	8.7

IMDB

# of queries with nonzero cardinality: 1,070					
Total time for exact evaluation: 4,189,495 ms					
	Basic	Opt	Comb	Ord-Var	Ord-Fix
Total time (ms)	69	713	116	87	59,790
# > exact	1	0	1	26	789
# $q\text{-}err = \infty$	27	141	21	38	17
# $q\text{-}err > 10$	150	312	145	272	17
Max $q\text{-}err \neq \infty$	126,739.8	182,178.1	126,739.8	73,763.6	4.7

Fig. 3. Summary of the results for conjunctive queries.

to partitioning, the work in Opt can depend on the size of the input graph. Nevertheless, all four techniques produce q-errors comparable to Ord-Fix, but with considerably less work.

Accuracy. All five variants can accurately estimate the cardinality of most queries. On all benchmarks apart from AIDS, Yago, and IMDB, the maximum finite q-error is below 43.8 for all variants. Moreover, Basic, Opt, and Comb produce a q-error of at most 32.7 on 90% of the queries on all benchmarks apart from Yago. These results echo the ones from Section 4 and show that WanderJoin-based algorithms seem to be much more accurate than other methods from the G-CARE framework, even the sampling-based ones. This is a direct consequence of sideways information passing: it reduces the sampling space for each atom and thus increases the likelihood of a valid match.

Interestingly, doing more work does not always improve the accuracy of Ord-Fix: the algorithm produced more zero estimates than Basic on AIDS, Human, Yago, WatDiv, and DBLP. Moreover, Ord-Var seems to be less precise than Basic and Comb despite doing about the same amount of work: the average and maximum q-errors of Ord-Var are larger in all cases apart from DBLP. As we discussed in Section 3.2, the variance of the estimates produced by different orders can vary significantly, which influences the rate of convergence of the estimate average. The simple ordering from Algorithm 4 seems to achieve its objective of minimising estimation variance. This seems particularly important on complex queries: Ord-Var and Ord-Fix were unable to produce nonzero estimates for 25% of the Yago queries, unlike the three variants that use the optimised order.

Zero Estimates. Our results show that, whenever an estimate is not zero, it is often accurate. However, all approaches sometimes incorrectly produce zero estimates, and queries with small

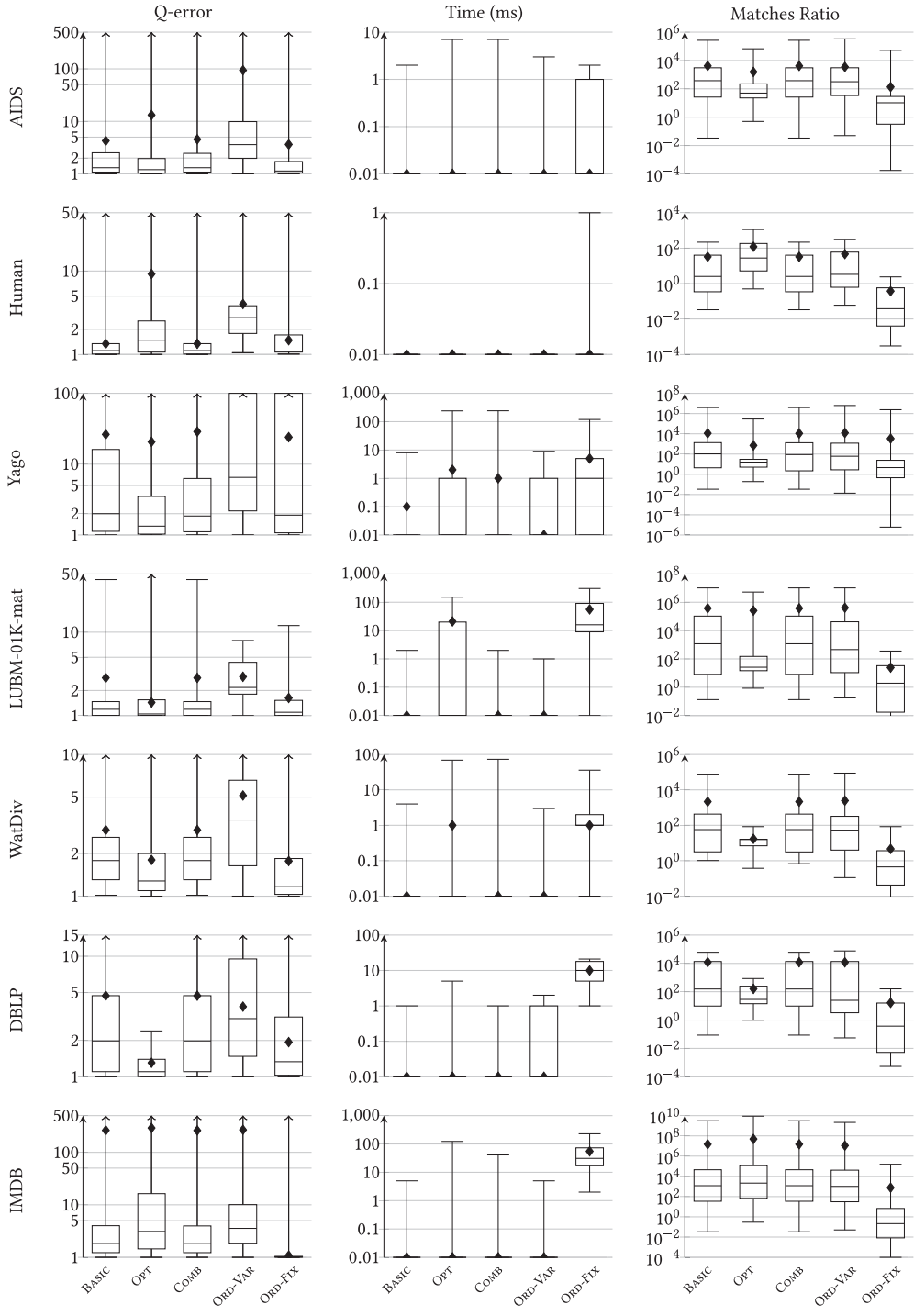


Fig. 4. Distribution of Q-Errors, times, and match ratios for conjunctive queries.

cardinalities seem most susceptible to this: on queries with at least 10,000 answers, Basic produces zero estimates only on 3 queries of AIDS, 42 queries of Yago, and 3 queries of IMDB. Sample space partitioning from Section 5.4 seems to alleviate this problem to some extent: the Opt variant produced a valid estimate of all but 146 queries of Yago, compared to 267, 419, and 438 queries for Basic, Ord-Var, and Ord-Fix. Furthermore, the Comb variant seems to mitigate some of the drawbacks of Opt: it produced the largest number of nonzero estimates on all benchmarks, but using the amount of work much closer to Basic than Opt. This is in fact the main motivation behind Comb, and we found it indispensable in our end-to-end experiments.

Empty Queries. On queries with no answers, all variants produce correct estimates, but the dynamic stopping condition always incurs the maximum number of runs. This is not a problem for Basic and Ord-Var, whose running time is independent of the database instance size; for example, Basic can process each empty query in under 1 ms. In contrast, the running time of Opt and Comb depends on the instance size due to relation partitioning, which, combined with the large number of runs, can be problematic. For example, the running time of Opt on the empty LUBM-01K-mat query is 703 ms, which is just under 782 ms required to process all other queries. The Ord-Var and Ord-Fix variants process this query in 2 ms and 18 ms, respectively, with match ratios of 204 and 17.7, respectively. On WatDiv, total estimation times for all 17 empty queries for Basic, Opt, Comb, Ord-Var, and Ord-Fix are 2 ms, 14 ms, 16 ms, 7 ms, and 11 ms, respectively.

Summary. The dynamic stopping condition seems very effective on nonempty queries, much more so than the fixed number of samples approach from the G-CARE framework. On empty queries, Basic and Ord-Var variants seem effective, whereas Opt and Comb can be slow. We discuss in Section 7.4 how this can affect query planning. Furthermore, the simple reordering algorithm decreases estimation variance, which seems particularly important for complex queries. However, if R^S used by Algorithm 4 are unavailable, then the Ord-Var variant can provide accurate and quick estimates in most cases. Finally, the Comb variant increases the likelihood of obtaining nonzero estimates, but without a considerable overhead on many queries.

7.3 Cardinality Estimation of Complex Queries

All benchmark queries are limited to simple conjunctions, and we are unaware of any publicly available repositories of real-word complex queries over our datasets. We thus used the following automated process to produce a collection of complex queries. For each nonempty query with at least four atoms, we used Algorithm 4 to produce an order of the form $\mathtt{AND}(A_1, \ldots, A_n)$. For $i = n/2$, we considered each atom A_j with $1 \leq j \leq i$ and tried to replace the relation of A_j with another relation, resulting in atom A'_j, such that query $\mathtt{AND}(A_1, \ldots, A_{j-1}, A'_j, A_{j+1}, \ldots, A_i)$ is not empty. If one such A'_j could be found, then we produced the following query of nonzero cardinality:

$$\mathtt{AND}(\mathtt{DISTINCT}(\mathtt{PROJECT}_S(\mathtt{AND}(A_1, \ldots, A_n)\ \mathtt{UNION}\ \mathtt{AND}(A_1, \ldots, A_{j-1}, A'_j, A_{j+1}, \ldots, A_i))), A_{i+1}, \ldots, A_n).$$

This transformation produced no query on the Human benchmark. On AIDS, Yago, LUBM-01K-mat, WatDiv, DBLP, and IMDB, we obtained 429, 820, 19, 41, 11, and 103 queries, respectively.

We then estimated the cardinality of these queries using the Basic, Opt, and Comb variants. We did not consider Ord-Var and Ord-Fix, because it is unclear how to enumerate all orders of complex queries. The results of our experiments are summarised in Figures 5 and 6 in the same way as in Section 7.2, and we next discuss our results.

Efficiency. As one might expect, complex queries are generally more difficult: exact evaluation takes considerably longer than for conjunctive queries on all benchmarks apart from Yago and IMDB. The hardest benchmark is again Yago, mainly because it involves evaluating complex

AIDS

# of queries: 429			
Total time for exact eval.: 126,107 ms			
	Basic	Opt	Comb
Total time (ms)	55	62	78
# > exact	0	0	0
# q-$err = \infty$	9	6	5
# q-$err > 10$	127	160	125
Max q-$err \neq \infty$	2,205.8	2,117.2	2,205.8

Yago

# of queries: 820			
Total time for exact eval.: 136,706 ms			
	Basic	Opt	Comb
Total time (ms)	319	315	492
# > exact	7	2	11
# q-$err = \infty$	169	159	130
# q-$err > 10$	311	341	291
Max q-$err \neq \infty$	15,017.0	11,610.4	15,017.0

LUBM-01K-mat

# of queries: 19			
Total time for exact eval.: 22,100 ms			
	Basic	Opt	Comb
Total time (ms)	8	4	8
# > exact	0	0	0
# q-$err = \infty$	0	0	0
# q-$err > 10$	7	6	7
Max q-$err \neq \infty$	603.5	876.5	603.5

WatDiv

# of queries: 41			
Total time for exact eval.: 1,588 ms			
	Basic	Opt	Comb
Total time (ms)	10	9	14
# > exact	0	1	0
# q-$err = \infty$	1	1	1
# q-$err > 10$	4	7	4
Max q-$err \neq \infty$	55.5	421.6	55.5

DBLP

# of queries: 11			
Total time for exact eval.: 751 ms			
	Basic	Opt	Comb
Total time (ms)	2	1	3
# > exact	1	1	2
# q-$err = \infty$	2	0	0
# q-$err > 10$	3	3	2
Max q-$err \neq \infty$	10.2	178.8	178.8

IMDB

# of queries: 103			
Total time for exact eval.: 52,244 ms			
	Basic	Opt	Comb
Total time (ms)	6	1	6
# > exact	1	0	1
# q-$err = \infty$	0	1	0
# q-$err > 10$	49	53	49
Max q-$err \neq \infty$	5,450.7	6,951,172.1	5,450.7

Fig. 5. Summary of the results for complex queries.

queries over a graph of nontrivial size. Nevertheless, our estimation algorithms are still very efficient: total estimation times are orders of magnitude lower than the times for exact query evaluation in all cases. Moreover, the number of queries on which estimation takes longer than exact evaluation is also much lower: only 11 queries of Yago, one query of WatDiv, two queries of DBLP, and one query of IMDB fall into this category.

Accuracy. We obtained accurate estimates on at least half of the queries of all benchmarks: The median q-error is below 11 on IMDB and below six on all other benchmarks. However, estimating complex queries is more difficult: the third quartile and maximum q-errors seem above the ones reported in Section 7.2. The large maximum q-error of Opt on IDMB is due to one query that was underestimated due to an insufficient number of runs; Basic obtained the q-error of 22 for the same query. Duplicate elimination seems to be the main source of difficulty: estimate accuracy increases when the same substitution is encountered multiple times, but the latter can be unlikely when the subquery of `DISTINCT` produces many answers. Nevertheless, our algorithms could handle well many of the benchmark queries. Again, Comb was effective in dealing with zero estimates: only five queries of AIDS, 130 queries of Yago, and one query of WatDiv could not be estimated.

7.4 End-to-end Experiments

We now explore whether our algorithms improve the end-to-end performance of query answering, which comprises both query planning and evaluation times. This would ideally be achieved by replacing the cardinality estimator of an existing graph database, but this is usually quite difficult: state-of-the-art systems are typically not available in open source, and the effort of integrating an algorithm into an existing, foreign code base is often significant. Thus, a common simplification is to precompute cardinalities of subqueries offline and inject them into an existing query planner. As part of their G-CARE study, Park et al. [55] have shown that injecting the cardinalities computed by wj into the query optimiser of RDF-3X [54], an influential RDF data store, considerably improves plan quality. Our algorithms produce comparable estimates to wj, and so injecting them into RDF-3X is likely to produce the same conclusions. Moreover, achieving a true end-to-end comparison would be hard due to various "impedance mismatches"; for example, RDF-3X is a disk-based system, whereas our algorithms have been implemented in RAM.

We thus follow a different strategy and conduct an end-to-end evaluation using our prototype system. Our query planner and query engine have been developed together, which removes any "impedance mismatches" between the two. This makes our results much more indicative of the kind of improvements one might expect in practice, at least for similar RAM-based systems.

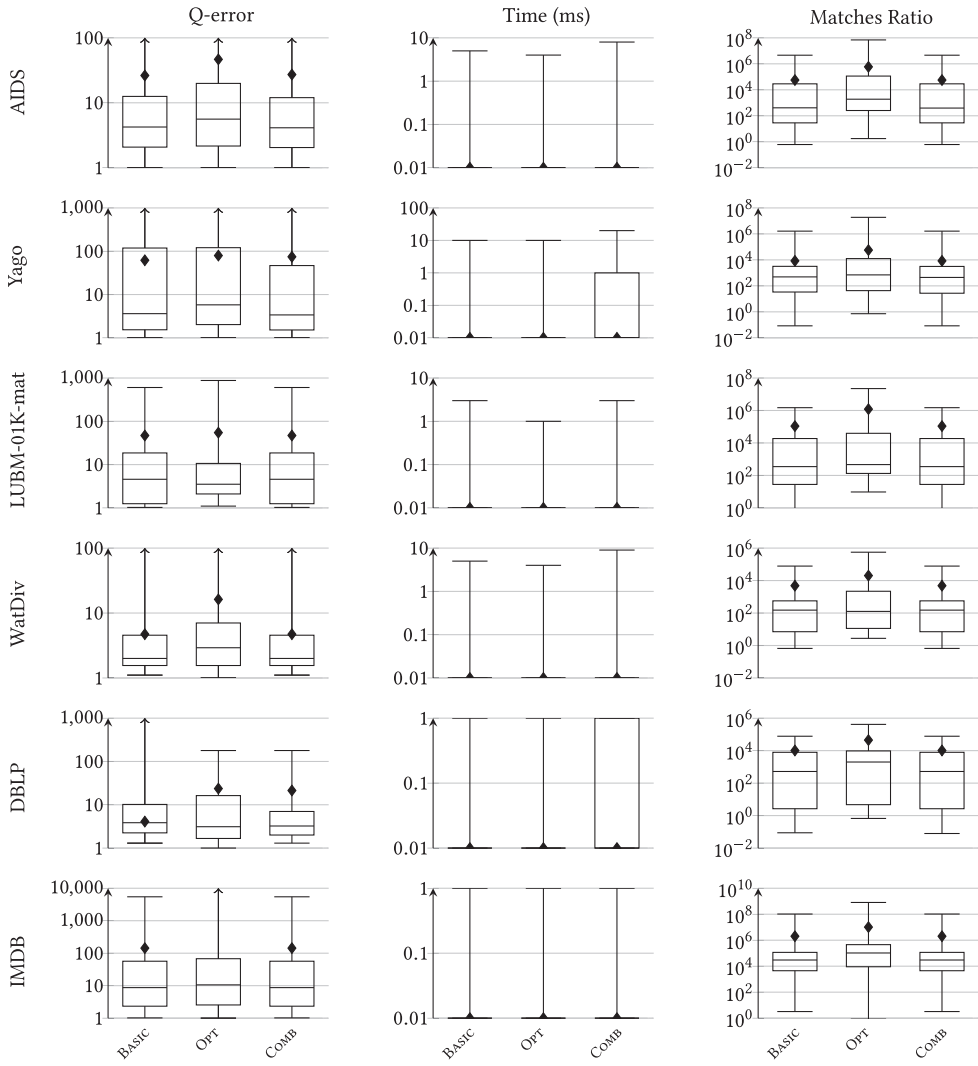


Fig. 6. Distribution of Q-Errors, times, and match ratios for complex queries.

We use the plans produced by the simple reordering approach from Algorithm 4 as the baseline for our evaluation. These plans proved very effective in practice: Query evaluation took longer than 1s only for three queries of AIDS, 17 queries of Yago, five queries of LUBM-01K-mat, and 57 queries of IMDB. We compare these plans with the ones obtained using dynamic programming, and our objective is to see whether using more precise cardinality estimates produces more efficient plans, but without unacceptable overheads. Table 4 summarises our results. The simple reordering is near-instantaneous, so we ignore the planning time; in contrast, we report the planning time ("Plan."), the evaluation time ("Eval."), and the sum of the two ("Total") for the dynamic programming approach. We also report the number of queries ("# Faster") on which the respective approach was faster in terms of total time, as well as the maximum difference ("Max. Δ") in total evaluation time for any query. We report the results for empty and nonempty queries separately.

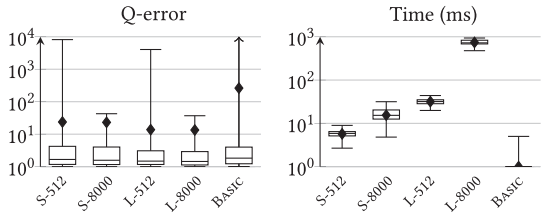
Table 4. Results of the End-to-end Experiments

	reorder–by–fanout			reorder–DP				
	Time (ms)		#	Time (ms)				#
	Total	Max. Δ	Faster	Total	Plan.	Eval.	Max. Δ	Faster
AIDS	22,252	49	207	3,226	687	2,539	11,151	241
Human	3	1	3	3	1	2	1	3
Yago	331,649	5,260	601	49,882	23,646	26,236	172,743	398
LUBM-01K-mat; nonempty Q	10,857	426	6	9,005	31	8,974	695	13
LUBM-01K-mat; empty Q	147	13,046	1	13,193	13,049	144	—	0
WatDiv; nonempty Q	301	18	27	318	150	168	32	20
WatDiv; empty Q	2	13	11	116	112	4	—	0
DBLP	511	10	5	443	5	438	79	6
IMDB	4,189,495	134,074	305	3,647,271	1,475	3,645,796	406,019	404

As one can see, evaluation of nonempty queries is generally faster on all benchmarks when using plans produced by the dynamic programming approach. As shown in the "Max. Δ" column, evaluation can be significantly faster on some queries, showing that having access to precise cardinality estimates can play a critical role in evaluation of complex queries. However, our results also show that calling the cardinality estimator during query planning can be a considerable source of overhead: repeated calls to Comb account for 21%, 47%, and 47% of the overall query evaluation time on AIDS, Yago, and WatDiv, respectively. Switching to the Basic variant does not seem to help: we observed that planning times remain largely unaffected. As shown in Section 7.2, Basic can produce nonzero estimates for almost all queries on benchmarks other than AIDS and Yago, so Opt is called infrequently on these benchmarks anyway. Moreover, the AIDS dataset is small, so the overhead of sample space partitioning is manageable.

The results in Section 7.2 show that the difference between the running times of Basic and Comb is most pronounced on Yago. However, when Basic is used instead of Comb, our dynamic programming algorithms sometimes produce very poor plans. Yago queries are very complex, so sometimes the cardinality of a candidate order $\mathsf{AND}(A_1, \ldots, A_i)$ can be much larger than the cardinality of a prefix $\mathsf{AND}(A_1, \ldots, A_{i'})$ for some $i' < i$. In other words, prefix $\mathsf{AND}(A_1, \ldots, A_{i'})$ acts like a bottleneck that makes finding a valid sample for $\mathsf{AND}(A_1, \ldots, A_i)$ difficult. This, in turn, introduces "blind spots" for the planning algorithm: because of the high selectivity of $\mathsf{AND}(A_1, \ldots, A_{i'})$, the algorithm does not "see" that extending this order with further atoms leads to a massive increase in evaluation cost. In other words, zero estimates should be interpreted as "no information available" rather than "cardinality is small," and they can have a considerable impact on the resulting plan quality. This, in fact, is the main motivation for the Opt approach: the main objective of sample space partitioning is to explore a bigger portion of the sample space and thus produce at least some information about the distribution of the data over which a query is evaluated. Furthermore, the main motivation behind Comb is to avoid a potentially high overhead of sample space partitioning in cases when estimates can be produced easily using the Basic variant.

Table 4 also shows that the planning overhead can be significant on queries with zero cardinality. On LUBM-01K-mat, computing the plan for the one empty query takes longer than the evaluation of all remaining 35 nonempty queries combined. On WatDiv, the planning overheads on empty queries are somewhat smaller, but still significant. The reason for this is simple: a run of Algorithm 5 on an empty query is likely to invoke the cardinality estimator many times on subqueries that are likely to be empty as well; moreover, the algorithm uses the Comb variant, which always invokes Opt on an empty input; thus, the overheads of Opt and Comb are compounded by the large number of invocations. There are several heuristics that can be used to overcome



	S-512	S-8000	L-512	L-8000	Basic
Total time (ms)	6,119	16,549	33,988	792,159	69
# $q\text{-}err = \infty$	0	0	0	0	27
# $q\text{-}err > 10$	152	144	118	108	150
Max $q\text{-}err \neq \infty$	8,169.0	42.7	4,050.0	37.0	126,739.8
# $q\text{-}err > q\text{-}err_{\text{Basic}}$	512	496	473	456	—

Fig. 7. Comparison with NeuroCard.

this problem. For example, one can install a budget for the number of calls to Comb variant and fall back to the Basic variant after this budget is exhausted. Alternatively, one can initially check whether the input query is empty using the Comb variant; if so, one can either resort to the simple ordering only or use the Basic variant instead of Comb in line 18 of Algorithm 5. On the one empty LUBM-01K-mat query, the latter approach reduces the planning time to 53 ms, which seems reasonable.

To summarise, the results of our end-to-end experiments suggest that having access to accurate cardinality estimates can dramatically improve the performance of query answering; however, producing these estimates can incur a nontrivial overhead. In our future work, we shall explore ways to further reduce this overhead. In particular, instead of sampling the data "from scratch" each time a subquery is encountered in line 10 of Algorithm 5, we shall explore ways to sample the data incrementally (e.g., only for the atom added in line 8) while still producing unbiased estimates.

7.5 Comparison with NeuroCard

In this section, we compare our approach with NeuroCard [71], an influential cardinality estimation approach based on machine learning. NeuroCard takes as input a join schema that specifies how to construct a full outer join of the relevant database relations. This join is sampled and the resulting tuples are used to train a deep neural model that approximates the distribution of the tuples in the join. This model can be used to estimate the cardinality of any query whose joins are covered by the join schema. The cardinality of a query covered by the join schema can be estimated by adding up the relevant parts of the approximated distribution. However, computing this sum exactly would be computationally very costly, so NeuroCard only estimates the sum using Monte Carlo integration—a technique that involves sampling the approximated distribution. NeuroCard was shown to be highly accurate on the IMDB benchmark.

The code of NeuroCard is available on GitHub, but applying it to our benchmarks is not straightforward. First, it is unclear which join schema to use: a join schema of NeuroCard must be acyclic and cover the entire query load, but our benchmarks contain many cyclic queries with self-joins that violate these restrictions. Second, many aspects of the NeuroCard code seem to be hardwired to IMDB. Therefore, we limit our comparison to the IMDB benchmark only.

The NeuroCard GitHub repository provides a small pretrained model for the `job-light` queries and a small and a large pretrained model for the `job-light-ranges` queries. Unfortunately, the join schema of neither model covers all 1,070 benchmark queries. Thus, we retrained a small and a large model using a join schema that covers all queries. Then, for each query, we computed the estimate on both models using the sampling rates of 512 and 8,000 for Monte Carlo integration. The model sizes, training parameters, and integration sampling rates were determined by the NeuroCard code. We thus obtained four estimates and estimation times per query. Figure 7 summarises our results, where S- and L- indicate the model size, and 512 and 8,000 indicate the integration sampling rates. The figure also recapitulates the results for Basic, and it shows the number of queries on which Basic achieved a smaller q-error ("# $q\text{-}err > q\text{-}err_{\text{Basic}}$").

Overall, NeuroCard and Basic produced estimates of comparable accuracy: the third quartile of the q-error is always within 4.5. NeuroCard produced no zero estimates, and it was more accurate in the tail end of the distribution, but it was also significantly slower: even in the S-512 variant, the total time for processing all queries is almost two orders of magnitude larger than for Basic, despite the fact that computation could use a specialised graphics card. Using considerably less work, Basic achieved lower q-errors than NeuroCard on between 40% and 50% of queries.

Although NeuroCard and WanderJoin seem fundamentally different at first glance, a deeper comparison actually reveals surprising similarities. To construct training examples for the neural model, NeuroCard uses a variant of random join sampling by Zhao et al. [76], which shares many similarities with WanderJoin. Moreover, sampling during Monte Carlo integration is again closely related to WanderJoin-style sampling. The two techniques thus seem to use closely related principles, which, we believe, explains why they achieve similar levels of accuracy.

A key difference between the two techniques is in how sampling is operationalised. In our case, the data distribution is sampled directly, and the sampling process is guided by the query whose cardinality is to be estimated. This can be very efficient if adequate indexes are available, as is the case in our implementation. In contrast, NeuroCard approximates the data distribution using a synopsis; furthermore, sampling is guided by a join schema, so the resulting synopsis is tailored to the query workload captured by the join schema. Anticipating the query workload may be difficult in graph databases, since graph queries tend to explore the data in ad hoc ways. Moreover, interpreting the synopsis in NeuroCard can require considerable resources. On the upside, the neural models used by NeuroCard are generally orders of magnitude smaller than the database instance and can thus be kept in RAM, which can be beneficial in many use cases.

8 Conclusion

In this article, we presented an in-depth study of sampling-based algorithms for estimating query cardinality. Our work is based on WanderJoin [47]—an algorithm introduced in the context of online aggregation. We reformulate the algorithm in light of sideways information passing, a family of techniques used to optimise query evaluation, which allows us to extend the approach to complex queries with arbitrary operator nesting. We present two variants of our approach and show that the average of repeated estimates realises a strongly consistent estimator of query cardinality. We show on an extensive set of benchmarks that our algorithms can accurately estimate conjunctive and complex queries while using considerably less work than exact evaluation. In addition, we show that a combination of our cardinality estimation algorithms with dynamic programming can often produce join orders that are considerably more efficient than the orders produced by ad hoc assumptions. Finally, we show that our approach can provide estimates of similar accuracy but with much less work than the deep learning–based NeuroCard approach [71].

We see several exciting avenues for future work. On the conceptual side, we shall consider extending our approach to different kinds of recursive queries. We are unaware of any estimation approach that can handle recursive path queries, which is a key problem in CRPQ planing. Moreover, database statistics are typically unavailable for relations defined by Datalog rules, which can prevent successful planning of Datalog queries. On the practical side, we see two important problems. First, it is currently unclear how to apply our approaches when database instances are stored in secondary storage. Our evaluation results show that the number of facts matched to query atoms can be high in some cases, which has the potential to introduce a nontrivial I/O cost. Second, we shall investigate ways to reduce redundancy when our cardinality estimators are called repeatedly during query planning. This could perhaps be achieved by caching samples collected in distinct estimator runs.

Appendices

A Proof of Theorem 5.5

To prove Theorem 5.5, we first relate invocations of Algorithm 2 on some I, Q, and σ to the notion of an estimator from Section 2.2. To this end, Definition A.1 introduces a set of outcomes $\Omega^{I,Q,\sigma}$ representing the choices available to the algorithm, a probability distribution $\mathrm{P}^{I,Q,\sigma}$ on $\Omega^{I,Q,\sigma}$ describing how the algorithm makes these choices, a function $S^{I,Q,\sigma}$ mapping each outcome to the corresponding substitution, and a function $\hat{C}^{I,Q,\sigma}$ mapping each outcome to a cardinality estimate. For convenience, we first introduce the set $\Theta^{I,Q,\sigma}$ of all successful outcomes, and then we extend it to the set $\Omega^{I,Q,\sigma}$ that also contains the failure outcome $\bot$. These definitions are inductive in the sense that $\Omega^{I,Q,\sigma}$, $\mathrm{P}^{I,Q,\sigma}$, $S^{I,Q,\sigma}$, and $\hat{C}^{I,Q,\sigma}$ depend on the definitions of $\Omega^{I,Q',\sigma'}$, $\mathrm{P}^{I,Q',\sigma'}$, $S^{I,Q',\sigma'}$, and $\hat{C}^{I,Q,\sigma}$ for each subquery Q' of Q and each substitution σ' satisfying $\mathrm{dom}(\sigma') \subseteq \mathsf{v}(Q')$.

Definition A.1. *For each database instance I, query Q, and substitution σ with $\mathrm{dom}(\sigma) \subseteq \mathsf{v}(Q)$, let $\Theta^{I,Q,\sigma}$ be a set of outcomes, let $\mathrm{P}^{I,Q,\sigma}$ and $\hat{C}^{I,Q,\sigma}$ be real-valued functions on $\Theta^{I,Q,\sigma}$, and let $S^{I,Q,\sigma}(\omega)$ be a function mapping each $\omega \in \Theta^{I,Q,\sigma}$ to a substitution $S^{I,Q,\sigma}(\omega)$ as in Figure 8 subject to the following.*

- *In the case for $Q = A$, probability $\mathrm{P}(\omega)$ is the sampling probability $\mathrm{P}(F)$ from the corresponding case in Algorithm 2. Analogously, in the case for $Q = Q_1$ UNION Q_2, probabilities p_1 and p_2 are from the corresponding case in Algorithm 2.*
- *In the case for $Q = \mathtt{DISTINCT}(Q_1)$, function $D^Q : \mathrm{eval}_I(Q,\sigma) \to \Theta^{I,Q_1,\sigma}$ is an arbitrary injective mapping fixed for Q such that $S^{I,Q_1,\sigma}(D^Q[\sigma_1]) = \sigma_1$ holds for each $\sigma_1 \in \mathrm{eval}_I(Q,\sigma)$.*[4]

Moreover, let $\Omega^{I,Q,\sigma} = \Theta^{I,Q,\sigma} \cup \{\bot\}$ be the sample space that extends $\Theta^{I,Q,\sigma}$ with a distinct failure outcome *$\bot$, and let*

$$\mathrm{P}^{I,Q,\sigma}(\bot) = 1 - \sum_{\omega \in \Theta^{I,Q,\sigma}} \mathrm{P}^{I,Q,\sigma}(\omega), \qquad S^{I,Q,\sigma}(\bot) = \emptyset, \qquad \text{and} \qquad \hat{C}^{I,Q,\sigma}(\bot) = 0. \tag{16}$$

Lemma A.2 establishes certain important properties of $\Omega^{I,Q,\sigma}$, $\mathrm{P}^{I,Q,\sigma}$, $S^{I,Q,\sigma}$, and $\hat{C}^{I,Q,\sigma}$. In particular, property (P1) says that all usual probability axioms (e.g., that probabilities of all outcomes add up to one) are satisfied, and so $\hat{C}^{I,Q,\sigma}$ is an estimator on $\Omega^{I,Q,\sigma}$. Property (P2) says that $\hat{C}^{I,Q,\sigma}$ satisfies the Horvitz–Thompson property on all successful outcomes. Finally, property (P3) shows that the substitutions produced by all successful outcomes cover precisely $\mathrm{eval}_I(Q,\sigma)$.

Lemma A.2. *Properties (P1)–(P3) are satisfied for each database instance I, query Q, and substitution σ with $\mathrm{dom}(\sigma) \subseteq \mathsf{v}(Q)$.*

(P1) Function $\mathrm{P}^{I,Q,\sigma}$ is a probability distribution on the sample space $\Omega^{I,Q,\sigma}$.
(P2) $\hat{C}^{I,Q,\sigma}(\omega) \cdot \mathrm{P}^{I,Q,\sigma}(\omega) = 1$ for each $\omega \in \Theta^{I,Q,\sigma}$.
(P3) $\mathrm{eval}_I(Q,\sigma) = \{\!\{ S^{I,Q,\sigma}(\omega) \mid \omega \in \Theta^{I,Q,\sigma} \}\!\}$.

Proof. For (P1), it suffices to show that $\sum_{\omega \in \Theta^{I,Q,\sigma}} \mathrm{P}^{I,Q,\sigma}(\omega) \leq 1$; then, the definition of $\mathrm{P}^{I,Q,\sigma}(\bot)$ from Equation (16) ensures that $\mathrm{P}^{I,Q,\sigma}$ is a correctly defined probability distribution on $\Omega^{I,Q,\sigma}$. The proof is by a straightforward induction on the structure of Q. For the induction base $Q = A$, probability distribution $\mathrm{P}^{I,Q,\sigma}$ is obtained from the probability distribution on the sample space, which immediately implies property (P1), and properties (P2) and (P3) follow directly from definitions in Figure 8. For the induction step, consider $Q = Q_1$ AND Q_2. By the induction assumption, properties (P1)–(P3) hold for Q_1 and Q_2 and appropriate substitutions. By the definition of $\mathrm{ans}_I(Q)$, each substitution in $\mathrm{eval}_I(Q,\sigma)$ can be written as $\sigma \cup \sigma_2 \cup \sigma_2$, where $\mu_1 = \sigma|_{Q_1}$,

[4] Note that the domain of D^Q is correctly defined, since $\mathrm{eval}_I(Q,\sigma)$ is a set, rather than a multiset.

- For $Q = A$, let
$$\Theta^{I,Q,\sigma} = \{F \in \text{sspace}_I(\sigma(A)) \mid \text{there exists a matcher of } \sigma(A) \text{ to } F\},$$
and, for each $\omega \in \Theta^{I,Q,\sigma}$, let
$$\mathrm{P}^{I,Q,\sigma}(\omega) = \mathrm{P}(\omega), \quad S^{I,Q,\sigma}(\omega) = \text{the matcher of } \sigma(A) \text{ to } \omega, \quad \text{and} \quad \hat{C}^{I,Q,\sigma}(\omega) = 1/\mathrm{P}(\omega).$$

- For $Q = Q_1$ AND Q_2, let
$$\Theta^{I,Q,\sigma} = \{\langle \omega_1, \omega_2 \rangle \mid \omega_1 \in \Theta^{I,Q_1,\sigma|_{Q_1}} \text{ and } \omega_2 \in \Theta^{I,Q_2,(\sigma \cup \sigma_1)|_{Q_2}} \text{ where } \sigma_1 = S^{I,Q_1,\sigma|_{Q_1}}(\omega_1)\},$$
and, for each $\langle \omega_1, \omega_2 \rangle \in \Theta^{I,Q,\sigma}$, let
$$\begin{aligned} \mathrm{P}^{I,Q,\sigma}(\langle \omega_1, \omega_2 \rangle) &= \mathrm{P}^{I,Q_1,\mu_1}(\omega_1) \cdot \mathrm{P}^{I,Q_2,\mu_2}(\omega_2), \\ S^{I,Q,\sigma}(\langle \omega_1, \omega_2 \rangle) &= \sigma \cup \sigma_1 \cup \sigma_2, \qquad \text{and} \\ \hat{C}^{I,Q,\sigma}(\langle \omega_1, \omega_2 \rangle) &= \hat{C}^{I,Q_1,\mu_1}(\omega_1) \cdot \hat{C}^{I,Q_2,\mu_2}(\omega_2) \end{aligned}$$
where $\mu_1 = \sigma|_{Q_1}$, $\sigma_1 = S^{I,Q_1,\mu_1}(\omega_1)$, $\mu_2 = (\sigma \cup \sigma_1)|_{Q_1}$, and $\sigma_2 = S^{I,Q_2,\mu_2}(\omega_2)$.

- For $Q = Q_1$ UNION Q_2, let
$$\Theta^{I,Q,\sigma} = \{\langle i, \omega_i \rangle \mid i \in \{1, 2\} \text{ and } \omega_i \in \Theta^{I,Q_i,\sigma}\},$$
and, for each $\langle i, \omega_i \rangle \in \Theta^{I,Q,\sigma}$, let
$$\mathrm{P}^{I,Q,\sigma}(\langle i, \omega_i \rangle) = p_i \cdot \mathrm{P}^{I,Q,\sigma}(\omega_i), \quad S^{I,Q,\sigma}(\langle i, \omega_i \rangle) = S^{I,Q_i,\sigma}(\omega_i), \quad \text{and} \quad \hat{C}^{I,Q,\sigma}(\langle i, \omega_i \rangle) = \hat{C}^{I,Q_i,\sigma}(\omega_i)/p_i.$$

- For $Q = Q_1$ MINUS Q_2, let
$$\Theta^{I,Q,\sigma} = \{\omega \in \Theta^{I,Q_1,\sigma} \mid \text{eval}_I(Q_2, \sigma_1)(\omega) = \emptyset \text{ where } \sigma_1 = S^{I,Q_1,\sigma}(\omega)\}$$
and, for each $\omega \in \Theta^{I,Q,\sigma}$, let
$$\mathrm{P}^{I,Q,\sigma}(\omega) = \mathrm{P}^{I,Q_1,\sigma}(\omega), \quad S^{I,Q,\sigma}(\omega) = S^{I,Q_1,\sigma}(\omega), \quad \text{and} \quad \hat{C}^{I,Q,\sigma}(\omega) = \hat{C}^{I,Q_1,\sigma}(\omega).$$

- For $Q = Q_1$ FILTER E, let
$$\Theta^{I,Q,\sigma} = \{\omega \in \Theta^{I,Q_1,\sigma} \mid \sigma_1(E) = \text{true} \text{ where } \sigma_1 = S^{I,Q_1,\sigma}(\omega)\}$$
and, for each $\omega \in \Theta^{I,Q,\sigma}$, let
$$\mathrm{P}^{I,Q,\sigma}(\omega) = \mathrm{P}^{I,Q_1,\sigma}(\omega), \quad S^{I,Q,\sigma}(\omega) = S^{I,Q_1,\sigma}(\omega), \quad \text{and} \quad \hat{C}^{I,Q,\sigma}(\omega) = \hat{C}^{I,Q_1,\sigma}(\omega).$$

- For $Q = Q_1$ BIND $x := E$, let
$$\Theta^{I,Q,\sigma} = \{\omega \in \Theta^{I,Q_1,\sigma} \mid \sigma_1(E) \neq \epsilon \text{ and } \sigma \sim \{x \mapsto \sigma_1(E)\} \text{ where } \sigma_1 = S^{I,Q_1,\sigma}(\omega)\}$$
and, for each $\omega \in \Theta^{I,Q,\sigma}$, let
$$\mathrm{P}^{I,Q,\sigma}(\omega) = \mathrm{P}^{I,Q_1,\sigma}(\omega), \quad S^{I,Q,\sigma}(\omega) = \sigma_1 \cup \{x \mapsto \sigma_1(E)\}, \quad \text{and} \quad \hat{C}^{I,Q,\sigma}(\omega) = \hat{C}^{I,Q_1,\sigma}(\omega)$$
where $\sigma_1 = S^{I,Q_1,\sigma}(\omega)$.

- For $Q = \text{PROJECT}_X(Q_1)$, let
$$\Theta^{I,Q,\sigma} = \Theta^{I,Q_1,\sigma},$$
and, for each $\omega \in \Theta^{I,Q,\sigma}$, let
$$\mathrm{P}^{I,Q,\sigma}(\omega) = \mathrm{P}^{I,Q_1,\sigma}(\omega), \quad S^{I,Q,\sigma}(\omega) = S^{I,Q_1,\sigma}(\omega)|_X, \quad \text{and} \quad \hat{C}^{I,Q,\sigma}(\omega) = \hat{C}^{I,Q_1,\sigma}(\omega).$$

- For $Q = \text{DISTINCT}(Q_1)$, let
$$\Theta^{I,Q,\sigma} = \{\omega \in \Theta^{I,Q_1,\sigma} \mid D^Q[S^{I,Q_1,\sigma}(\omega)] = \omega\}$$
and, for each $\omega \in \Theta^{I,Q_1,\sigma}$, let
$$\mathrm{P}^{I,Q,\sigma}(\omega) = \mathrm{P}^{I,Q_1,\sigma}(\omega), \quad S^{I,Q,\sigma}(\omega) = S^{I,Q_1,\sigma}(\omega), \quad \text{and} \quad \hat{C}^{I,Q,\sigma}(\omega) = \hat{C}^{I,Q_1,\sigma}(\omega).$$

Fig. 8. Equations for $\Theta^{I,Q,\sigma}$, $\mathrm{P}^{I,Q,\sigma}$, $\hat{C}^{I,Q,\sigma}$, and $S^{I,Q,\sigma}(\omega)$ from Definition A.1.

$\sigma_1 = S^{I,Q_1,\mu_1}(\omega_1)$, $\mu_2 = (\sigma \cup \sigma_1)|_{Q_1}$, and $\sigma_2 = S^{I,Q_2,\mu_2}(\omega_2)$ for some $\omega_1 \in \Theta^{I,Q_1,\mu_1}$ and $\omega_2 \in \Theta^{I,Q_2,\mu_2}$. But then, definitions in Figure 8 clearly ensure properties (P1)–(P3). The cases for $Q = Q_1$ UNION Q_2, $Q = Q_1$ MINUS Q_2, $Q = Q_1$ FILTER E, $Q = Q_1$ BIND $x := E$, and $Q = \mathtt{PROJECT}_X(Q_1)$ are analogous, so we omit them for the sake of brevity. For $Q = \mathtt{DISTINCT}(Q_1)$, mapping D^Q associates each $\sigma_1 \in \mathrm{eval}_I(Q, \sigma)$ with a "representative" outcome $D^Q[\sigma_1] \in \Theta^{I,Q_1,\sigma}$ of the subquery Q_1. But then, the definition of $\Theta^{I,Q,\sigma}$ clearly ensures property (P3), and properties (P1) and (P2) hold by the inductive assumption. □

Properties (P2) and (P3) of Lemma A.2 allow us to prove the following lemma:

Lemma A.3. *For each database instance I, query Q, and substitution σ with $\mathrm{dom}(\sigma) \subseteq \mathsf{v}(Q)$, random variable $\hat{C}^{I,Q,\sigma}$ is an unbiased estimator of $|\mathrm{eval}_I(Q, \sigma)|$.*

Proof. For arbitrary I, Q, and σ as in the lemma, the expectation of $\hat{C}^{I,Q,\sigma}$ is given by

$$\mathbb{E}[\hat{C}^{I,Q,\sigma}] = \sum_{\omega \in \Omega^{I,Q,\sigma}} \mathrm{P}^{I,Q,\sigma}(\omega) \cdot \hat{C}^{I,Q,\sigma}(\omega) = \sum_{\omega \in \Theta^{I,Q,\sigma}} \mathrm{P}^{I,Q,\sigma}(\omega) \cdot \hat{C}^{I,Q,\sigma}(\omega) + \mathrm{P}^{I,Q,\sigma}(\bot) \cdot \hat{C}^{I,Q,\sigma}(\bot) =$$
$$= \sum_{\omega \in \Theta^{I,Q,\sigma}} 1 = |\Theta^{I,Q,\sigma}| = |\mathrm{eval}_I(Q, \sigma)|.$$

The last equality is ensured by property (P3). Hence, estimator $\hat{C}^{I,Q,\sigma}$ is unbiased, as required. □

We are now ready to prove Theorem 5.5.

Theorem 5.5. *Let $\hat{\theta}_1, \hat{\theta}_2, \ldots$ be the sequence of random variables representing the third component of the results of successive calls to $\mathrm{estimate}_I(Q, \sigma)$ for some I, Q, and σ with $\mathrm{dom}(\sigma) \subseteq \mathsf{v}(Q)$.*

— *The sequence of averages $\frac{1}{n} \cdot \sum_{i=1}^{n} \hat{\theta}_i$ is a strongly consistent estimator of $|\mathrm{eval}_I(Q, \sigma)|$.*

— *If Q does not contain* DISTINCT*, then each $\hat{\theta}_i$ is an unbiased estimator of $|\mathrm{eval}_I(Q, \sigma)|$.*

Proof. Fix a database instance I, query Q, and substitution σ such that $\mathrm{dom}(\sigma) \subseteq \mathsf{v}(Q)$ holds, and let $\hat{\theta}_1, \hat{\theta}_2, \ldots$ be the sequence of random variables representing the third component of the results of successive calls to $\mathrm{estimate}_I(Q, \sigma)$. Moreover, let $\boldsymbol{Q}$ be the (possibly empty) set of all DISTINCT subqueries of Q, and, for each $Q' \in \boldsymbol{Q}$, let $\mathcal{S}^{Q'}$ be the set of all substitutions produced in line 30 when Algorithm 1 is applied to Q' and I. We say that, for $Q' \in \boldsymbol{Q}$, mapping $D^{Q'}$ used in Algorithm 2 is *fully populated* if $D^{Q'}[\sigma']$ is defined for each $\sigma' \in \mathcal{S}$ (and so the condition in line 37 of Algorithm 2 is never satisfied when the algorithm is applied to Q' and I).

Now consider any random variable $\hat{\theta}_i$ representing a run of Algorithm 2 in which all mappings $D^{Q'}$ are fully populated. Algorithm 2 then returns $[\omega, S^{I,Q,\sigma}(\omega), \hat{C}^{I,Q,\sigma}(\omega)]$ with probability $\mathrm{P}^{I,Q,\sigma}(\omega)$ for some $\omega \in \Omega^{I,Q,\sigma}$. This is because definitions in Figure 8 closely follow the structure of Algorithm 2. For example, for $Q = Q_1$ AND Q_2, the recursive calls for Q_1 and Q_2 are made with substitutions $\mu_1 = \sigma|_{Q_1}$ and $\mu_1 = (\sigma \cup \sigma_1)|_{Q_2}$; thus, if the recursive call for each $i \in \{1, 2\}$ returns $[\omega_i, S^{I,Q_i,\mu_i}(\omega_i), \hat{C}^{I,Q_i,\mu_i}(\omega_i)]$ with probability $\mathrm{P}^{I,Q_i,\mu_i}(\omega_i)$ where $\sigma_i = S^{I,Q_i,\mu_i}(\omega_i)$, then definitions in Figure 8 clearly ensure the required property. The analysis is analogous for all other query types, and we omit the details for brevity. But then, $\hat{\theta}_i$ is an unbiased estimator of $|\mathrm{eval}_I(Q, \sigma)|$ by Lemma A.3. Moreover, the assumption that mappings $D^{Q'}$ are fully populated is vacuously true when $\boldsymbol{Q} = \emptyset$, which implies the second claim of this theorem.

To prove the first claim, let $\hat{\mu}_n = \frac{1}{n} \cdot \sum_{i=1}^{n} \hat{\theta}_i$ be the sequence of random variables of estimate averages. We can define $\hat{\mu}_n$ on a sample space Ω consisting of infinite words of the form $\omega_1, \omega_2, \ldots$ where each ω_i reflects the random choices that Algorithm 2 makes in i-th run. We also identify the following two events on this probability space, and our objective is to show that $\mathrm{P}(\Omega_1) = 1$.

$$\Omega_1 = \{\omega \in \Omega \mid \lim_{n\to\infty} \hat{\mu}(\omega) = |\mathrm{eval}_I(Q,\sigma)|\}$$

$$\Omega_2 = \{\omega \in \Omega \mid \text{there exists a run } \textit{min}\omega\text{at which all} D^{Q'} \text{ become fully populated}\}$$

We first prove $\mathrm{P}(\Omega \setminus \Omega_2) = 0$. Consider any DISTINCT subquery $Q' \in \boldsymbol{Q}$ of Q and any substitution $\sigma' \in \mathcal{S}^{Q'}$, and let $\Psi^{Q',\sigma'}$ be the event containing each $\omega \in \Omega$ such that $D^{Q'}[\sigma']$ is never defined. Let p be the smallest probability with which a run of Algorithm 2 produces σ'. Clearly, $p > 0$, since producing σ' is possible. The probability that σ' is not produced after n runs is then at most $(1-p)^n$; and, since $\lim_{n\to\infty}(1-p)^n = 0$, we have $\mathrm{P}(\Psi^{Q',\sigma'}) = 0$. Thus, the probability of the intersection of arbitrary sets $\Psi^{Q',\sigma'}$ is zero as well, and, by decomposing $\mathrm{P}(\Omega \setminus \Omega_2)$ in terms of intersections of $\Psi^{Q',\sigma'}$ using the inclusion–exclusion principle, we have $\mathrm{P}(\Omega \setminus \Omega_2) = 0$. Hence, $\mathrm{P}(\Omega_2) = 1$.

Now, for each $\omega \in \Omega_2$, step m at which all $D^{Q'}$ become fully defined in ω, and $k > m$, let $\hat{\rho}_k^{\omega}$ be the random variable defined by

$$\hat{\rho}_k^{\omega} = \frac{1}{k-m} \sum_{i=m+1}^{k} \hat{\theta}_i,$$

and let

$$\Omega_3 = \{\omega \in \Omega_2 \mid \lim_{i\to\infty} \hat{\rho}_{m+i}^{\omega}(\omega) = |\mathrm{eval}_I(Q,\sigma)|\}.$$

Consider an arbitrary $\omega \in \Omega_3$ and the corresponding m. Then, for $k > m$, we have

$$\hat{\mu}_k(\omega) = \frac{1}{k}\sum_{i=1}^{k} \hat{\theta}_i(\omega) = \frac{1}{k}\sum_{i=1}^{m} \hat{\theta}_i(\omega) + \frac{k-m}{k}\frac{1}{k-m}\sum_{i=m+1}^{k} \hat{\theta}_i(\omega) = \frac{1}{k}\sum_{i=1}^{m} \hat{\theta}_i(\omega) + \frac{k-m}{k}\hat{\rho}_k(\omega).$$

As k approaches infinity, the first term approaches zero, since $\sum_{i=1}^{k} \hat{\theta}_i(\omega)$ is a constant, and $\frac{k-m}{k}$ approaches one. But then, $\omega \in \Omega_3$ implies that $\hat{\rho}_{m+1}^{\omega}(\omega)$ approaches $|\mathrm{eval}_I(Q,\sigma)|$, and so $\hat{\mu}_k(\omega)$ approaches $|\mathrm{eval}_I(Q,\sigma)|$ as well. Hence, we have $\omega \in \Omega_1$, which implies $\Omega_3 \subseteq \Omega_1$.

Finally, we prove $\mathrm{P}(\Omega_3) = 1$. Together with $\Omega_3 \subseteq \Omega_1$, this implies $1 = \mathrm{P}(\Omega_3) \leq \mathrm{P}(\Omega_1) \leq 1$, which proves our first claim. Let ℓ be the number of distinct ways in which mappings $D^{Q'}$ can be fully defined is finite. This ℓ is finite, so we can decompose Ω_2 as $\Omega_2 = \bigcup_{i=1}^{\ell} \Omega_2^i$ such that each Ω_2^i contains precisely all $\omega \in \Omega_2$ that instantiate mappings $D^{Q'}$ in the same way. Clearly, we have $\Omega_2^i \cap \Omega_2^j = \emptyset$ for $1 \leq i < j \leq \ell$, which implies $1 = \mathrm{P}(\Omega_2) = \sum_{i=1}^{\ell} \mathrm{P}(\Omega_2^i)$. Moreover, $\Omega_3 = \bigcup_{i=1}^{\ell} \Omega_3 \cap \Omega_2^i$. Now, for each $1 \leq i \leq \ell$, Algorithm 1 instantiated as dictated by Ω_2^i is an unbiased estimator of $|\mathrm{eval}_I(Q,\sigma)|$, as argued at the beginning of this proof. By the Kolmogorov's strong law of large numbers, the sequence of averages of consecutive runs is a strongly consistent estimator of $|\mathrm{eval}_I(Q,\sigma)|$, and so $\mathrm{P}(\Omega_3 \mid \Omega_2^i) = 1$. But then, $\mathrm{P}(\Omega_3) = \sum_{i=1}^{\ell} \mathrm{P}(\Omega_2^i) \cdot \mathrm{P}(\Omega_3 \mid \Omega_2^i) = 1$, as required. □

B Proof of Theorem 5.9

Our proof strategy resembles the one in Appendix A: in Definition B.1, for each query Q and context substitution σ, we introduce the set of outcomes $\Psi^{I,Q,\sigma}$, a corresponding probability distribution $\mathrm{R}^{I,Q,\sigma}$, and the estimator $\hat{D}^{I,Q,\sigma}$. In Lemma B.2, we prove that $\hat{D}^{I,Q,\sigma}$ is an unbiased estimator of $|\mathrm{eval}_I(Q,\sigma)|$. Finally, Theorem 5.9 shows that these definitions describe the properties of Algorithm 3.

Definition B.1. *For each database instance I, query Q, and substitution σ with $\mathrm{dom}(\sigma) \subseteq \mathrm{v}(Q)$, let $\Psi^{I,Q,\sigma}$ be a set of outcomes, and let $\mathrm{R}^{I,Q,\sigma}$ and $\hat{D}^{I,Q,\sigma}$ be real-valued functions on $\Theta^{I,Q,\sigma}$ in Figure 9.*

Lemma B.2. *Properties (R1) and (R2) are satisfied for each database instance I, query Q, and substitution σ with $\mathrm{dom}(\sigma) \subseteq \mathrm{v}(Q)$.*

- For $Q = Q_1$ AND Q_2, let
$$\Psi^{I,Q,\sigma} = \{\langle \omega_1, \omega_2 \rangle \mid \omega_1 \in \Omega^{I,Q_1,\sigma|_{Q_1}} \text{ and } \omega_2 \in \Psi^{I,Q_2,(\sigma \cup \sigma_1)|_{Q_2}} \text{ where } \sigma_1 = S^{I,Q_1,\sigma|_{Q_1}}(\omega_1)\},$$
and, for each $\langle \omega_1, \omega_2 \rangle \in \Psi^{I,Q,\sigma}$, let
$$\mathrm{R}^{I,Q,\sigma}(\langle \omega_1, \omega_2 \rangle) = \mathrm{P}^{I,Q_1,\mu_1}(\omega_1) \cdot \mathrm{R}^{I,Q_2,\mu_2}(\omega_2) \quad \text{and} \quad \hat{D}^{I,Q,\sigma}(\langle \omega_1, \omega_2 \rangle) = \hat{C}^{I,Q_1,\mu_1}(\omega_1) \cdot \hat{D}^{I,Q_2,\mu_2}(\omega_2),$$
where $\mu_1 = \sigma|_{Q_1}$ and $\mu_2 = (\sigma \cup S^{I,Q_1,\mu_1}(\omega_1))|_{Q_1}$.

- For $Q = Q_1$ UNION Q_2, let
$$\Psi^{I,Q,\sigma} = \{\langle \omega_1, \omega_2 \rangle \mid \omega_1 \in \Psi^{I,Q_1,\sigma} \quad \text{and} \quad \omega_2 \in \Psi^{I,Q_1,\sigma}\},$$
and, for each $\langle \omega_1, \omega_2 \rangle \in \Psi^{I,Q,\sigma}$, let
$$\mathrm{R}^{I,Q,\sigma}(\langle \omega_1, \omega_2 \rangle) = \mathrm{R}^{I,Q_1,\sigma}(\omega_1) \cdot \mathrm{R}^{I,Q_2,\sigma}(\omega_2) \quad \text{and} \quad \hat{D}^{I,Q,\sigma}(\langle \omega_1, \omega_2 \rangle) = \hat{D}^{I,Q_1,\sigma}(\omega_1) + \hat{D}^{I,Q_2,\sigma}(\omega_2).$$

- For $Q = \mathtt{PROJECT}_X(Q_1)$, let $\Psi^{I,Q,\sigma} = \Psi^{I,Q_1,\sigma}$, $\mathrm{R}^{I,Q,\sigma} = \mathrm{R}^{I,Q_1,\sigma}$, and $\hat{D}^{I,Q,\sigma} = \hat{D}^{I,Q_1,\sigma}$.

- For $Q = A$, $Q = Q_1$ MINUS Q_2, $Q = Q_1$ FILTER E, $Q = Q_1$ BIND $x := E$, and $Q =$ DISTINCT(Q_1), let
$$\Psi^{I,Q,\sigma} = \Omega^{I,Q,\sigma}, \qquad \mathrm{R}^{I,Q,\sigma} = \mathrm{P}^{I,Q,\sigma}, \qquad \text{and} \qquad \hat{D}^{I,Q,\sigma} = \hat{C}^{I,Q,\sigma}.$$

Fig. 9. Equations for $\Psi^{I,Q,\sigma}$, $\mathrm{R}^{I,Q,\sigma}$, and $\hat{D}^{I,Q,\sigma}$ From Definition B.1.

(R1) Function $\mathrm{R}^{I,Q,\sigma}$ is a probability distribution on the sample space $\Psi^{I,Q,\sigma}$.
(R2) Function $\hat{D}^{I,Q,\sigma}$ is an unbiased estimator of $|\mathrm{eval}_I(Q,\sigma)|$.

Proof. The proof is by induction on the structure of query Q. For $Q = A$, $Q = Q_1$ MINUS Q_2, or $Q =$ DISTINCT(Q_1), Lemmas A.2 and A.3 and the definitions of $\Psi^{I,Q,\sigma}$, $\mathrm{R}^{I,Q,\sigma}$, and $\hat{D}^{I,Q,\sigma}$ clearly ensure properties (R1) and (R2). For $Q = \mathtt{PROJECT}_X(Q_1)$, the inductive assumption ensures that properties (R1) and (R2) hold for Q_1, so these properties clearly hold for Q as well.

Assume $Q = Q_1$ AND Q_2. Lemma A.2 ensures that $\mathrm{P}^{I,Q_1,\sigma|_{Q_1}}$ is a probability distribution on the sample space $\Omega^{I,Q_1,\sigma|_{Q_1}}$, and the inductive assumption for Q_2 ensures that $\mathrm{R}^{I,Q_2,(\sigma\cup\sigma_1)|_{Q_2}}$ is a probability distribution on the sample space $\Psi^{I,Q_2,(\sigma\cup\sigma_1^i)|_{Q_2}}$; but then, property (R1) obviously holds for Q. To prove property (R2), we compute the expectation of $\hat{D}^{I,Q,\sigma}$ as follows, where $\mu_1 = \sigma|_{Q_1}$ and $\mu_2 = (\sigma \cup S^{I,Q_1,\mu_1}(\omega_1))|_{Q_2}$ for each ω_1.

$$\begin{aligned}
\mathbb{E}[\hat{D}^{I,Q,\sigma}] &= \sum_{\omega_1 \in \Omega^{I,Q,\mu_1}} \sum_{\omega_2 \in \Psi^{I,Q_2,\mu_2}} \mathrm{P}^{I,Q_1,\mu_1}(\omega_1) \cdot \mathrm{R}^{I,Q_2,\mu_2}(\omega_2) \cdot \hat{C}^{I,Q_1,\mu_1}(\omega_1) \cdot \hat{D}^{I,Q_2,\mu_2}(\omega_2) = \\
&= \sum_{\omega_1 \in \Omega^{I,Q,\mu_1}} \mathrm{P}^{I,Q_1,\mu_1}(\omega_1) \cdot \hat{C}^{I,Q_1,\mu_1}(\omega_1) \cdot \sum_{\omega_2 \in \Psi^{I,Q_2,\mu_2}} \mathrm{R}^{I,Q_2,\mu_2}(\omega_2) \cdot \hat{D}^{I,Q_2,\mu_2}(\omega_2) = \\
&= \sum_{\omega_1 \in \Omega^{I,Q,\mu_1}} 1 \cdot \mathbb{E}[\hat{D}^{I,Q_2,\mu_2}] = \sum_{\omega_1 \in \Omega^{I,Q,\mu_1}} |\mathrm{eval}_I(Q_2,\mu_2)| = |\mathrm{eval}_I(Q,\sigma)|.
\end{aligned}$$

Lemma A.2 ensures $\mathrm{P}^{I,Q_1,\mu_1}(\omega_1) \cdot \hat{C}^{I,Q_1,\mu_1}(\omega_1) = 1$, and $\mathbb{E}[\hat{D}^{I,Q_2,\mu_2}] = |\mathrm{eval}_I(Q_2,\mu_2)|$, since $\hat{D}^{I,Q_2,\mu_2}$ is unbiased by the induction assumption. Thus, $\hat{D}^{I,Q,\sigma}$ is an unbiased estimator of $|\mathrm{eval}_I(Q,\sigma)|$.

For $Q = Q_1$ UNION Q_2, Definition B.1 ensures $\hat{D}^{I,Q,\sigma} = \hat{D}^{I,Q_1,\sigma} + \hat{D}^{I,Q_2,\sigma}$, so property (R1) holds. By the induction assumption, $\hat{D}^{I,Q_i,\sigma}$ is an unbiased estimator of $|\mathrm{eval}_I(Q_i,\sigma)|$ for $i \in \{1,2\}$. Since $\mathrm{eval}_I(Q,\sigma)$ is the multiset union of $\mathrm{eval}_I(Q_1,\sigma)$ and $\mathrm{eval}_I(Q_2,\sigma)$, we have

$$|\mathrm{eval}_I(Q,\sigma)| = |\mathrm{eval}_I(Q_1,\sigma)| + |\mathrm{eval}_I(Q_2,\sigma)| = \mathbb{E}[\hat{D}^{I,Q_1,\sigma}] + \mathbb{E}[\hat{D}^{I,Q_2,\sigma}] = \mathbb{E}[\hat{D}^{I,Q,\sigma}],$$

where the last equality holds by the well-known properties of sums of random variables. Consequently, property (R2) is satisfied. □

Theorem 5.9. *Let $\hat{\theta}_1, \hat{\theta}_2, \ldots$ be the sequence of random variables representing the results of successive calls to* $\text{estimate}_I^{opt}(Q, \sigma)$ *for some I, Q, and σ with* $\text{dom}(\sigma) \subseteq \text{v}(Q)$.

— *The sequence of averages $\frac{1}{n} \cdot \sum_{i=1}^{n} \hat{\theta}_i$ is a strongly consistent estimator of $|\text{eval}_I(Q, \sigma)|$.*

— *If Q does not contain* `DISTINCT`*, then each $\hat{\theta}_i$ is an unbiased estimator of $|\text{eval}_I(Q, \sigma)|$.*

Proof. Assume that all mappings $D^{Q'}$ used to process `DISTINCT` subqueries of Q are fully populated (see the proof of Theorem 5.9). Also, consider an arbitrary partition $\mathcal{S}_1, \ldots, \mathcal{S}_N$ of $\text{sspace}_I(\sigma(A_1))$ from line 4 of Algorithm 3. Since $\mathcal{S}_i \cap \mathcal{S}_j = \emptyset$ for all $1 \leq i < j \leq N$, we have

$$\text{eval}_I(Q, \sigma) = \bigcup_{i=1}^{N} \bigcup_{\beta \in \text{eval}_{\mathcal{S}_i}(A_1, \sigma)} \text{eval}_I(Q_2, (\sigma \cup \beta)|_{Q_2}).$$

By Lemma B.2, line 8 provides an unbiased estimator of the latter union for each i with $1 \leq i \leq N$. All of these estimates are added in line 8 of Algorithm 3, so the resulting sum is an unbiased estimator of $|\text{eval}_I(Q, \sigma)|$. With this observation in mind, the proof of both claims is completely analogous to the proof of Theorem 5.5, so we omit the details for the sake of brevity. □

Acknowledgments

We thank Felix Pahl for his key insights that allowed us to prove Theorem 5.5. For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript (AAM) version arising from this submission.

References

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. 2007. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07)*. VLDB Endowment, 411–422.

[2] A. Aboulnaga and S. Chaudhuri. 1999. Self-tuning histograms: Building histograms without looking at data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'99)*. ACM, 181–192.

[3] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. 1999. Join synopses for approximate query answering. In *Proceedings of the International Conference on Management of Data (SIGMOD'99)*. ACM Press, 275–286.

[4] G. Aluç, O. Hartig, M. Tamer Özsu, and K. Daudjee. 2014. Diversified stress testing of RDF data management systems. In *Proceedings of the 13th International Semantic Web Conference (ISWC'14)*. Springer, 197–212.

[5] S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, M. A. Martínez-Prieto, and G. Navarro. 2015. Compressed vertical partitioning for efficient RDF management. *Knowl. Inf. Syst.* 44, 2 (2015), 439–474.

[6] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. 2017. Foundations of modern query languages for graph databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40.

[7] P. Barceló Baeza. 2013. Querying graph databases. In *Proceedings of the 32nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'13)*. ACM, 175–188.

[8] C. Beeri and R. Ramakrishnan. 1991. On the power of magic. *J. Logic Program.* 10, 3&4 (1991), 255–299.

[9] D. P. Bertsekas and J. N. Tsitsiklis. 2008. *Introduction to Probability* (2nd ed.). Athena Scientific, Belmont, MA, USA.

[10] N. Bruno. 2003. *Statistics on Query Expressions in Relational Database Management Systems*. Ph. D. Dissertation. Columbia University.

[11] N. Bruno and S. Chaudhuri. 2004. Conditional selectivity for statistics on query expressions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*. ACM, 311–322.

[12] N. Bruno, S. Chaudhuri, and L. Gravano. 2001. STHoles: A multidimensional workload-aware histogram. *SIGMOD Rec.* 30, 2 (2001), 211–222.

[13] W. Cai, M. Balazinska, and D. Suciu. 2019. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *Proceedings of the 40th International Conference on Management of Data (SIGMOD'19)*. ACM Press, 18–35.

[14] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. 2001. Approximate query processing using wavelets. *VLDB J.* 10, 2 (2001), 199–223.

[15] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. 2000. Towards estimation error guarantees for distinct values. In *Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'00)*. ACM, 268–279.

[16] J. Chen, Y. Huang, M. Wang, S. Salihoglu, and K. Salem. 2022. Accurate summary-based cardinality estimation through the lens of cardinality estimation graphs. *Proc. VLDB Endow.* 15, 8 (2022), 1533–1545.

[17] X. Chen and J. C. S. Lui. 2016. Mining graphlet counts in online social networks. In *Proceedings of the 16th International IEEE Conference on Data Mining (ICDM'16)*. IEEE Computer Society, 71–80.

[18] X. Chen and J. C. S. Lui. 2018. Mining graphlet counts in online social networks. *ACM Trans. Knowl. Discov. Data* 12, 4 (2018), 1–38.

[19] C. A. Galindo-Legaria, M. Joshi, F. Waas, and M.-C. Wu. 2003. Statistics on views. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB'03)*. Morgan Kaufmann, 952–962.

[20] H. Garcia-Molina, J. D. Ullman, and J. Widom. 2000. *Database System Implementation*. Prentice-Hall, Upper Saddle River, NJ, USA.

[21] M. Garofalakis and P. B. Gibbons. 2002. Wavelet synopses with error guarantees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*. ACM, 476–487.

[22] L. Getoor, B. Taskar, and D. Koller. 2001. Selectivity estimation using probabilistic models. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*. ACM, 461–472.

[23] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. 2000. Approximating multi-dimensional aggregate range queries over real attributes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*. ACM, 463–474.

[24] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. 2005. Selectivity estimators for multidimensional range queries over real attributes. *VLDB J.* 14, 2 (2005), 137–154.

[25] Y. Guo, Z. Pan, and J. Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *J. Web Semant.* 3, 2–3 (2005), 158–182.

[26] P. J. Haas. 1997. Large-sample and deterministic confidence intervals for online aggregation. In *Proceedings of the 9th International Conference on Scientific and Statistical Database Management (SSDBM'97)*. IEEE Computer Society, 51–63.

[27] P. J. Haas and J. M. Hellerstein. 1999. Ripple joins for online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'99)*. ACM Press, 287–298.

[28] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. 1996. Selectivity and cost estimation for joins based on random sampling. *J. Comput. Syst. Sci.* 52, 3 (1996), 550–569.

[29] P. J. Haas and A. N. Swami. 1992. Sequential sampling procedures for query size estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'92)*. ACM Press, 341–350.

[30] S. Harris and A. Seaborne. 2013. SPARQL 1.1 Query Language, W3C Recommendation. Retrieved from https://www.w3.org/TR/sparql11-query/

[31] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das. 2020. Deep learning models for selectivity estimation of multi-attribute queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. ACM, 1035–1050.

[32] M. Heimel, M. Kiefer, and V. Markl. 2015. Self-tuning, GPU-accelerated kernel density models for multidimensional selectivity estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. ACM, 1477–1492.

[33] J. M. Hellerstein, P. J. Haas, and H. J. Wang. 1997. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*. ACM Press, 171–182.

[34] A. Hertzschuch, G. Moerkotte, W. Lehner, N. May, F. Wolf, and L. Fricke. 2021. Small selectivities matter: Lifting the burden of empty samples. In *Proceedings of the International Conference on Management of Data (SIGMOD'21)*. ACM, 697–709.

[35] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig. 2020. DeepDB: Learn from data, not from queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005.

[36] D. G. Horvitz and D. J. Thompson. 1952. A generalization of sampling without replacement from a finite universe. *J. Amer. Statist. Assoc.* 47, 260 (1952), 663–685.

[37] Pan Hu and Boris Motik. 2024. Accurate Sampling-based Cardinality Estimation for Complex Graph Queries: Code, Datasets, and Experimental Results. Retrieved from https://krr-nas.cs.ox.ac.uk/2024/cardinality-sampling/

[38] Y. E. Ioannidis. 2003. The history of histograms (abridged). In *Proceedings of the 29th International Conference on Very Large Databases (VLDB'03)*. Morgan Kaufmann, Berlin, Germany, 19–30.

[39] Y. E. Ioannidis and S. Christodoulakis. 1991. On the propagation of errors in the size of join results. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'91)*. ACM, 268–277.

[40] Z. G. Ives and N. E. Taylor. 2008. Sideways information passing for push-style query processing. In *Proceedings of the 24th International Conference on Data Engineering (ICDE'08)*. IEEE Computer Society, 774–783.

[41] C. Jin, S. S. Bhowmick, B. Choi, and S. Zhou. 2012. PRAGUE: Towards blending practical visual subgraph query formulation and query processing. In *Proceedings of the 28th IEEE International Conference on Data Engineering (ICDE'12)*. IEEE Computer Society, 222–233.

[42] M. A. Khamis, H. Q. Ngo, and D. 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-SIGART Symposium on Principles of Database Systems (PODS'17)*. ACM, 429–444.

[43] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. 2019. Learned cardinalities: Estimating correlated joins with deep learning. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR'19)*. Retrieved from www.cidrdb.org

[44] G. Klyne, J. J. Carroll, and B. McBride. 2014. RDF 1.1 Concepts and Abstract Syntax. Retrieved from https://www.w3.org/TR/rdf11-concepts/

[45] J.-H. Lee, D.-H. Kim, and C.-W. Chung. 1999. Multi-dimensional selectivity estimation using compressed histogram information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'99)*. ACM, 205–214.

[46] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[47] F. Li, B. Wu, K. Yi, and Z. Zhao. 2019. Wander join and XDB: Online aggregation via random walks. *ACM Trans. Datab. Syst.* 44, 1 (2019), 2:1–2:41.

[48] R. J. Lipton and J. F. Naughton. 1990. Query size estimation by adaptive sampling. In *Proceedings of the 9th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'90)*. ACM Press, 40–46.

[49] Y. Liu, T. Safavi, A. Dighe, and D. Koutra. 2018. Graph summarization methods and applications: A survey. *ACM Comput. Surv.* 51, 3 (2018), 62:1–62:34.

[50] Y. Matias, J. S. Vitter, and M. Wang. 1998. Wavelet-based histograms for selectivity estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*. ACM, 448–459.

[51] M. Müller, L. Woltmann, and W. Lehner. 2023. Enhanced featurization of queries with mixed combinations of predicates for ML-based cardinality estimation. In *Proceedings of the 26th International Conference on Extending Database Technology (EDBT'23)*. OpenProceedings.org, 273–284.

[52] T. Neumann and G. Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proceedings of the 27th International Conference on Data Engineering (ICDE'11)*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 984–994.

[53] T. Neumann and G. Weikum. 2009. Scalable join processing on very large RDF graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*. ACM, 627–640.

[54] T. Neumann and G. Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *VLDB J.* 19, 1 (2010), 91–113.

[55] Y. Park, S. Ko, S. S. Bhowmick, K. Kim, K. Hong, and W.-S. Han. 2020. G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In *Proceedings of the 41st ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. ACM Press, 1099–1114.

[56] Y. Park, S. Zhong, and B. Mozafari. 2020. QuickSel: Quick selectivity learning with mixture models. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. ACM, 1017–1033.

[57] J. Pérez, M. Arenas, and C. Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Datab. Syst.* 34, 3 (2009), 1–45.

[58] V. Poosala and Y. E. Ioannidis. 1997. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 23rd International Conference on Very Large Databases (VLDB'97)*. Morgan Kaufmann, 486–495.

[59] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. 1996. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*. ACM, 294–305.

[60] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. 2008. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.* 1, 1 (2008), 364–375.

[61] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear. 2013. Materialization strategies in the vertica analytic database: Lessons learned. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE'13)*. IEEE Computer Society, 1196–1207.

[62] J. Spiegel and N. Polyzotis. 2009. TuG synopses for approximate query answering. *ACM Trans. Datab. Syst.* 34, 1 (2009), 3:1–3:56.

[63] G. Stefanoni, B. Motik, and E. V. Kostylev. 2018. Estimating the cardinality of conjunctive queries over RDF data using graph summarisation. In *Proceedings of the World Wide Web Conference (WWW'18)*. ACM, 1043–1052.

[64] F. M. Suchanek, G. Kasneci, and G. Weikum. 2008. YAGO: A large ontology from Wikipedia and WordNet. *J. Web Semant.* 6, 3 (2008), 203–217.

[65] The Neo4j Team. 2023. Neo4j Cypher Manual. Retrieved from https://neo4j.com/docs/cypher-manual/current/introduction/
[66] P. Terlecki, H. Bati, C. A. Galindo-Legaria, and P. Zabback. 2009. Filtered statistics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*. ACM, 897–904.
[67] Apache TinkerPop. 2023. TinkerPop Documentation. Retrieved from https://tinkerpop.apache.org/docs/current/reference/
[68] K. Tzoumas, A. Deshpande, and C. S. Jensen. 2011. Lightweight graphical models for selectivity estimation without independence assumptions. *Proc. VLDB Endow.* 4, 11 (2011), 852–863.
[69] K. Tzoumas, A. Deshpande, and C. S. Jensen. 2013. Efficiently adapting graphical models for selectivity estimation. *VLDB J.* 22, 1 (2013), 3–27.
[70] D. Vengerov, A. C. Menck, M. Zaït, and S. Chakkappen. 2015. Join size estimation subject to filter conditions. *Proc. VLDB Endow.* 8, 12 (2015), 1530–1541.
[71] Z. Yang, A. Kamsetty, S. Luan, E. Liang, Y. Duan, X. Chen, and I. Stoica. 2020. NeuroCard: One cardinality estimator for all tables. *Proc. VLDB Endow.* 14, 1 (2020), 61–73.
[72] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. 2019. Deep unsupervised cardinality estimation. *Proc. VLDB Endow.* 13, 3 (2019), 279–292.
[73] F. Yu, W.-C. Hou, C. Luo, D. Che, and M. Zhu. 2013. CS2: A new database synopsis for query estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM, 469–480.
[74] P. Yuan, C. Xie, H. Jin, L. Liu, G. Yang, and X. Shi. 2014. Dynamic and fast processing of queries on large-scale RDF data. *Knowl. Inf. Syst.* 41, 2 (2014), 311–334.
[75] S. Zhang, S. Li, and J. Yang. 2009. GADDI: Distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology (EDBT'09)*. ACM, 192–203.
[76] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. 2018. Random sampling over joins revisited. In *Proceedings of the 39th ACM SIGMOD International Conference on Management of Data (SIGMOD'18)*. ACM, 1525–1539.
[77] L. Zou, J. Mo, L. Chen, M. Tamer Özsu, and D. Zhao. 2011. gStore: Answering SPARQL queries via subgraph matching. *Proc. VLDB Endow.* 4, 8 (2011), 482–493.

Received 25 July 2023; revised 4 March 2024; accepted 1 August 2024