

# PICC Counting: Who Needs Joins when you Can Propagate Efficiently?\*

Jong Wook Kim<sup>†</sup>

K. Selçuk Candan<sup>†</sup>

## Abstract

Counting is a common task in many data mining applications, including market basket data analysis, scientific inquiry, and other high dimensional data management applications. Given a single table, obtaining the instance counts of the entries in the table is relatively cheap. In situations where the attributes of interest are distributed across different tables, however, the problem of computing instance counts can be very expensive. The naive solution, joining all the relevant relations to obtain a single table suitable for counting, is rarely practical. In this paper, we propose PICC (Propagation-based Instance Counts on Concise Graphs), a novel counting technique for discovering instance counts in databases. We first propose a propagation-based instance counting scheme which avoids joins to obtain a single table. We then present a method for summarizing a database into a concise synopsis and describe how to use this along with the propagation scheme to estimate the required counts efficiently. The experiment results show that the proposed technique, PICC, provides significant execution time and accuracy gains over the existing solutions to this problem.

## 1 Motivation and Related Work

In many data analysis applications, such as business intelligence or scientific inquiry, it is common for the analysts to look for relationships and patterns across various data attributes. For example consider an archaeologist carrying out a study on animal bone specimens collected at various sites using the relation `boneDB` shown in Figure 1. For her analysis, this scientist may want to answer the following question, “What is the number of Coyote bones found in AZ?”. In algebraic terms, her question would translate to

$$\mathfrak{S}_{COUNT(*)}(\sigma_{spe.='Coyote' \wedge found\_in='AZ'}(boneDB))$$

In this example, counts of the relevant value pairs can be computed (relatively) cheaply by a single pass over the `boneDB` table. This is, however, only a best case

boneDB					
researcher	inst.	found_in	bCode	bType	species
Tom	ASU	AZ	bC_1	Skull	Coyote
Tom	ASU	AZ	bC_1	Skull	Wolf
Mike	ASU	AZ	bC_1	Skull	Coyote
Mike	ASU	AZ	bC_1	Skull	Wolf
Brady	USC	CA	bC_1	Skull	Coyote
Brady	USC	CA	bC_1	Skull	Wolf

Figure 1: Motivating example: an archaeology database scenario. In the real-world, databases commonly consist of multiple tables and users’ interest (specified using queries) usually spans multiple such tables. Counting becomes costlier when the context of analysis spans multiple tables in a database. Consider, for instance, the database schema and instances shown in Figures 2 (a) and (b), respectively.

EXAMPLE 1.1. Given the database schema in Figure 2 (a), one way to compute the instance count in the database for the pair of attributes-values “species=’Coyote’” and “found\_in=’AZ’” is to join the individual tables to obtain a combined “counting-context” table on which counting can be performed:

$$SBE \leftarrow (SpecBone \bowtie_{bType} BoneCode) \bowtie_{bCode} Excavation$$

$$CountingContext \leftarrow SBE \bowtie_{inst.} Researcher.$$

The resulting `CountingContext` table (Figure 2(c)), obtained by joining `Researcher`, `Excavation`, `BoneCode`, and `SpecBone`, contains the same information as the `boneDB` table in Figure 1 and thus can be used to obtain the counts the analyst needs.

Combining all the tables, however, can be expensive for large databases. In general, neither on demand construction of a combined table to be used as the counting context, nor constant maintenance of the context is practical. Thus our goal in this paper is to develop an efficient and effective approach to estimate counts, without having to combine all relevant relations in the database into a single relation.

**1.1 Selectivity Estimation and Counting** In many data mining applications, such as market basket analysis and transactional data analysis, knowledge about multi-attribute instance counts (e.g., numbers of

\*Supported by NSF Grant “Archaeological Data Integration for the Study of Long-Term Human and Social Dynamics (0624341)”

<sup>†</sup>Computer Science and Engineering Dept. Arizona State University, Tempe, AZ, 85287, USA. {jong, candan}@asu.edu

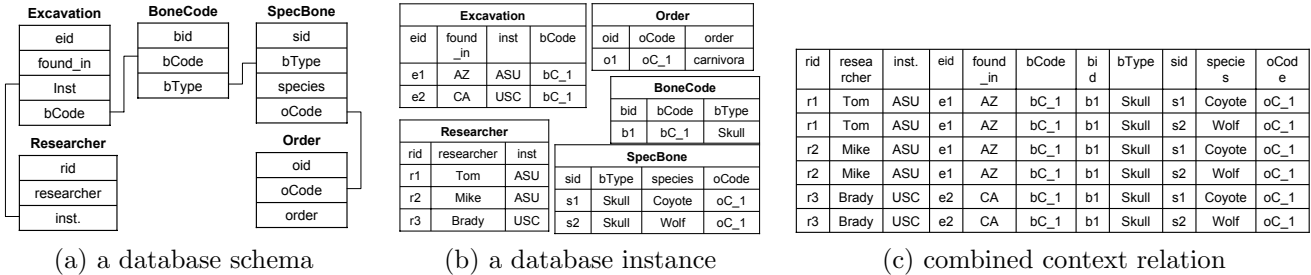


Figure 2: Motivating example: an archaeology database (continued)

occurrences of related values across different attributes) is used for discovering association rules [3, 13]. Counting the number of instances of related values is also common task in database management. For example, counts of attribute values are often used to compute selectivities of queries, which are then used for estimating the execution costs of these queries [2, 11, 14, 6, 5].

Histograms [6] and wavelets [5] that summarize counts of values of a given table are extensively used for query selectivity estimation. However, these techniques have shown to provide poor quality estimations when queries contain multiple joins [14]. Recently, there have been growing interest on relational data synopses to estimate the result sizes of queries in relational databases. Join synopses approach is commonly based on a random sampling of join results. The algorithm in [2] obtains random samples of join results in the schema and estimates the result size of queries over the sample. To capture the correlation among attributes in the database, [11] leverages a probabilistic relational model that is mapped onto a probabilistic graph model (i.e., Bayesian networks) which supports estimation of selectivities. These two techniques have limitations in that they can be applied to the queries with only one-to-many relations (i.e., foreign key based joins). To estimate the result sizes of the queries containing many-to-many relations, [14] proposed the *tuple graph* synopses (*TuG*) approach. In *TuG*, the data graph is summarized into a concise synopsis. Given a query, also represented as a graph, *TuG* approximately estimates the selectivity of the query by traversing the summarized graph. As we will show in the experiments section (Section 8), however, these approaches (fine-tuned for selectivity estimation task) are not always effective and efficient when applied to the count estimation problem.

**1.2 Contributions of this Paper** In some ways, the PICC (*Propagation-based Instance Counts on Concise Graphs*) scheme we propose in this paper is similar to *TuG*. For example, as in *TuG*, we also create and summarize a data graph into a concise synopsis to support our propagation mechanism. However, there are fundamental differences between *TuG* and PICC:

- Foremost, PICC is based on a novel propagation technique through which an individual node in the graph is informed with the global query context. Given a query containing many join operations, *TuG* has to traverse all relations involved. PICC, on the other hand, first creates a, so called, *contribution-graph* that annotates tuples in the database with meta-data that help limit the runtime propagation only to a *data-skeleton*, i.e., the subset of the database that is directly related to the attribute-values. The *contribution-graph* can be reused for counting different attribute-value instances within the same context query. Since it is common that, within the same query context, users may assign multiple counting tasks focussing on different data instances, this can provides savings.
- To reduce the overhead of accessing large data from the disk, in memory constrained environments, counting approaches have to rely on data synopses. We show that PICC works more effectively on summarized graphs than state-of-the-art method, *TuG*.

The rest of this paper is structured as follows. In the next section, we formalize the problem. In Section 3, we introduce the data graph structure that is the basis of the PICC approach to counting. In Section 4, we first describe how to distill a *contribution-graph* structure from the data graph and in Section 4.3, we present an algorithm to compute instance counts relying on this *contribution-graph* structure. In Section 6.1, we describe a method for summarizing the data graph into a concise synopsis suitable for memory constrained environments and in the rest of Section 6, we describe the PICC algorithm for estimating instance counts. In Section 7, we describe how to adjust the estimates if there is a sample of counting query result available. In Section 8, we experimentally evaluate PICC. The experiment results show that the PICC provides significant execution time and accuracy gains over existing solutions.

## 2 Problem Formulation

In this section, we formally define the problem and introduce some of the fundamental concepts on which we will rely when developing the PICC approach.

**2.1 Counting-Context** Let  $Q$  be a join query over the set of relations,  $\mathcal{S}$ , over which the analyst issues her attribute-value counting tasks. We call the query  $Q$  the *counting-context query* (or *context-query* in short).

Given a counting-context query,  $Q$ , over the set of relations,  $\mathcal{S}$ , let  $C_Q[A_1, A_2, \dots, A_n]$  be the table containing the result of the query. We call  $C_Q$  as the *counting-context relation*<sup>1</sup> (or *context-relation* in short).

**2.2 Instance Counting Task** An *instance counting task* over the counting-context,  $Q$ , is a set,  $AV_Q = \{(attr_1, v_1), \dots, (attr_m, v_m)\}$ , of attribute-value pairs. Here, each  $av_i = (attr_i, v_i) \in AV_Q$  is an attribute; i.e.,  $attr_i \in \{A_1, \dots, A_n\}$  in the result.

EXAMPLE 2.1. In our motivating example, the join query,  $Q$ , (i.e., the counting-context) is

$SBE \leftarrow (SpecBone \bowtie_{bType} BoneCode) \bowtie_{bCode} Excavation$   
*CountingContext*  $\leftarrow SBE \bowtie_{inst.} Researcher.$

while the corresponding instance counting task is

$AV_Q = \{(Excavation.found\_in, "AZ"), (SpecBone.spe, "Coyote")\}$

Intuitively, the user wants to know how many tuples in  $C_Q$  have the set of attribute-value pairs specified in  $AV_Q$ . In other words, given a counting task,  $AV_Q$ , against the context relation,  $C_Q$ , the instance count  $instCount(C_Q, AV_Q)$  is defined as follows;

$$instCount(C_Q, AV_Q) = \sum_{n_i \in C_Q} match(n_i, AV_Q),$$

where

- $n_i = \langle a_{1,i}, a_{2,i}, \dots, a_{n,i} \rangle$  denotes a tuple in  $C_Q[A_1, A_2, \dots, A_n]$  and
- $match(n_i, AV_Q)$  is 1 if the tuple  $n_i$  has matching values for the attribute value pairs in  $AV_Q$  and 0 otherwise.

The instance counting task,  $AV_Q$ , asks for  $instCount(C_Q, AV_Q)$ .

**2.3 Problem Statement** As we mentioned in the introduction, computing  $instCount(C_Q, AV_Q)$  precisely through materialization of  $C_Q$  may be prohibitively expensive. Thus, our objective in this paper is to develop an efficient algorithm to estimate instance counts in such a way that, given a count task,  $AV_Q$ , against  $C_Q$ , the absolute value of the estimation error,

$$\frac{|instCount(C_Q, AV_Q) - instCount_{est}(AV_Q)|}{|instCount(C_Q, AV_Q)|},$$

where  $instCount_{est}(AV_Q)$  is the instance count estimate obtained without materializing  $C_Q$ , is minimized.

<sup>1</sup>Note that our goal in this paper is to avoid materialization of this counting-context relation.

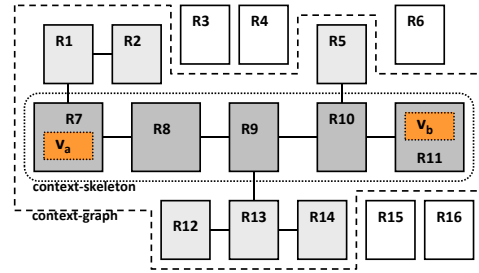


Figure 3: The structure of the context-graph reflects the given query,  $Q$ , while the context-skeleton (Section 4.3) is identified based on the given counting task,  $AV_Q$ .

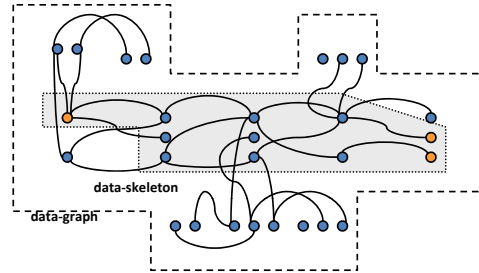


Figure 4: A data-graph and data-skeleton (Section 4.3.2) for the context-graph in Figure 3

### 3 Context and Data Graphs

As highlighted in Section 1, PICCC relies on a data graph for building the relevant data structures to enable efficient propagation. The data graph reflects the structure of the underlying counting-context.

**3.1 Counting-Context Graph** The *counting-context graph* (or *context-graph* in short) specifies which attributes of which relations are joined to obtain the counting-context relation,  $C_Q$  (Figure 3). Given a query,  $Q$ , the corresponding counting-context graph,  $\mathcal{G}_Q(N_Q, E_Q)$ , is an undirected acyclic<sup>2</sup> graph, where  $N_Q$  is the set of relations over which the query is defined and  $E_Q$  denotes the attribute correspondences among these relations. See for example Figure 5(a) for the counting-context graph for our running example.

Given  $\mathcal{G}_Q$ ,  $leaves(\mathcal{G}_Q)$  denotes the set of relations that correspond to the leaves in the context graph. Also, given a relation  $R$  in the database, the set,  $neighbors(R)$ , denotes  $R$ 's neighbors in the context graph. For example, in Figure 5,  $neighbors(BoneCode)$  is  $\{Excavation, SpecBone\}$ .

**3.2 Data Graph** The *data graph*,  $\mathcal{G}(N \cup V, E)$ , is an undirected graph where  $N$  is the set of tuples in the relations in the database and  $V$  is the set of all attribute values in the database. Let  $n_i \in N$  be a tuple node

<sup>2</sup>If a database table occurs multiple times in the query (e.g. a self-join), each table instance is included separately in  $N_Q$ , thus ensuring acyclicity of the context graph.

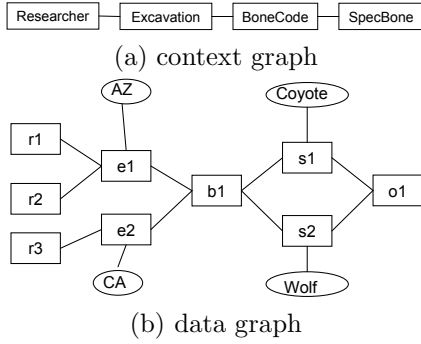


Figure 5: Motivating example (continued) – due to space constraints, in the data graph, only the values of “species” and “found\_in” are shown

representing a tuple in relation  $R_i$  and  $n_j$  be a tuple node of relation  $R_j$ . Then,

- the edge,  $(n_i, n_j) \in E$ , represents the fact that these two tuples satisfy a join condition specified in the query context graph and,
- if the tuple,  $n_i$ , has value,  $v_{h,k}$ , for attribute  $A_h$ , then  $(n_i, v_{h,k}) \in E$  is also included in  $\mathcal{G}$ .

See Figure 5(b) for the data graph for our running example. In the literature, similar data graphs have been used for join synopses generation. For example, [14] uses a *tuple graph* for join synopses creation. The PICC data graph is different from the *tuple graph* presented in [14] in that the PICC data graph reflects the structure of the underlying context-graph (shown in Figure 4), while *tuple graph* in [14] is generated based on the database schema.

**3.3 How Costly is it to Generate the Data Graph?** The number of join operations required to create the data graph and the number of joins needed to obtain the context relation are the same: both are equal to the number of edges in the query context graph. However, when creating the PICC data-graph, the input size to the join operations is much smaller than in full context relation materialization. This is because, the data graph is created by considering only pairs of tables, whereas context-relation accumulates joined tuples (potentially leading very large joins) to create the full combined context-relation. In Section 8.3, we report construction times of data graphs versus complete context relations.

#### 4 Propagation-based Counting – First Attempt

In this paper, we propose a novel strategy for efficiently and effectively estimating instance counts of a given set of attribute-value pairs. Our approach is based on a *propagation* technique, which leverages a pre-processing stage through which the individual tuples in the database are annotated with meta-data to help reduce the cost of instance count computation.

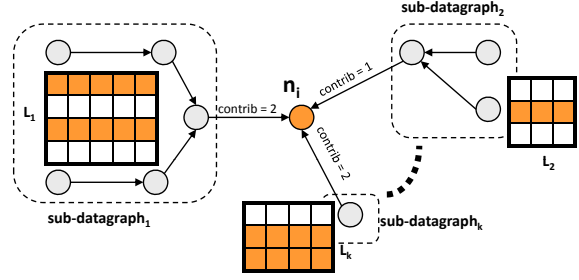


Figure 6: The number of  $n_i$ s in the result depends on the numbers of matching rows in the sub-results,  $L_1, L_2, \dots, L_k$

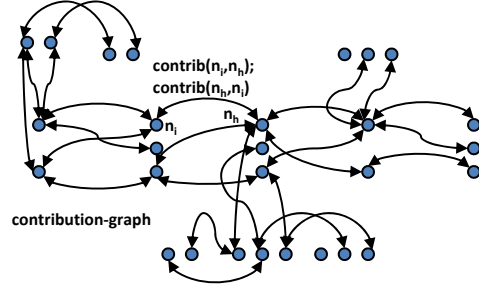


Figure 7: A contribution-graph for data-graph in Figure 4 (the edge weights are omitted)

**4.1 Intuition** Let us consider the example data graph segment shown in Figure 6. Here tuple node  $n_i$  is connected with tuple nodes in  $L_1, L_2, \dots, L_k$ , each representing the result of a sub-query to the corresponding portion of the context-graph. Essentially, the tuple node,  $n_i$  acts as a connector of  $k$  subgraphs, each corresponding to a sub-result table,  $L_i \in \{L_1, L_2, \dots, L_k\}$ .

If we were given the  $k$  sub-query results, we could easily compute the number of times  $n_i$  appears in the context-relation: we would simply identify the number of rows<sup>3</sup> in each sub-result table that joins with  $n_i$ .

**4.2 Contribution-Graph: A Pre-Computed Data Structure to Support Efficient Run-time Propagations** Bearing the above intuition in mind, PICC constructs a weighted, directed *contribution-graph*  $\mathcal{G}_c(N_c \cup V_c, E_c)$  to reduce to run-time cost of computing instance counts (Figure 7). Intuitively, the contribution-graph pre-computes *contributions* of the nodes of the data-graph onto each other’s counts:

**DEFINITION 4.1. (CONTRIBUTION-GRAPH)** *Let us be given a query context graph  $\mathcal{G}_Q(N_Q, E_Q)$  and a data graph  $\mathcal{G}(N \cup V, E)$ . The corresponding contribution-graph,  $\mathcal{G}_c(N_c \cup V_c, E_c)$ , is as follows:*

- $V_c = V$  and  $N_c = N$ ,
- For all  $(n_i, v_j) \in E$  where  $n_i \in N$  and  $v_j \in V$ , there are directed edges  $\langle n_i, v_j \rangle, \langle v_j, n_i \rangle \in E_c$ , such

<sup>3</sup>Illustrated using darker rows in Figure 6.

that<sup>4</sup> the corresponding edge weights are 1.

- For all  $(n_i, n_j) \in E$  where  $n_i, n_j \in N$ , there are directed edges  $\langle n_i, n_j \rangle, \langle n_j, n_i \rangle \in E_c$ , such that the edge weights,  $\text{contrib}(n_i, n_j)$ , represents the contribution of  $n_i$  to the count of  $n_j$ . ( $\text{contrib}(n_j, n_i)$  is similarly defined.)

If the relation containing tuple node  $n_i$  is a leaf relation in the context graph (that is,  $rl(n_i) \in \text{leafs}(\mathcal{G}_Q)$ ), then  $\text{contrib}(n_i, n_j) = 1$ .

Otherwise, if we let “neighbors\*” be a shorthand for “neighbors( $rl(n_i)$ ) \setminus  $rl(n_j)$ ” (i.e., all neighbors of the relation containing tuple  $n_i$  except for the relation containing tuple  $n_j$ ), then

$$\text{contrib}(n_i, n_j) = \prod_{L_h \in \text{neighbors}^*} \text{match}(L_h \rightarrow n_i).$$

Here  $\text{match}(L_h \rightarrow n_i)$  denotes the number of rows in the sub-result,  $L_h$ , that matches  $n_i$  (Figure 6):

$$\begin{aligned} \text{match}(L_h \rightarrow n_i) &= \sum_{n \in L_h \wedge (n, n_i) \in E} \text{match}(n \rightarrow n_i) \\ &= \sum_{n \in L_h \wedge (n, n_i) \in E} \text{contrib}(n, n_i). \end{aligned}$$

Intuitively, when the node  $n_i$  is a tuple of a leaf relation, the outgoing edge weights for  $n_i$  are set to 1. Otherwise, the edge weights depend on the weights of the incoming edges (i.e., the instance counts of the sub-queries).

EXAMPLE 4.1. Figure 8 shows the contribution-graph built from the data graph in Figure 5. Here, for instance,  $\text{contrib}(e_1, b_1)$  is computed as follows:

$$\begin{aligned} \text{contrib}(r_1, e_1) &= \text{contrib}(r_2, e_1) = 1 \\ \text{contrib}(e_1, b_1) &= \text{match}(\text{Researcher} \rightarrow e_1) = 2 \end{aligned}$$

In other words, computing the contribution of  $n_i$  to  $n_j$  requires contributions of  $n_i$ ’s neighbors to  $n_i$ . Thus, the edge weights can be computed by propagating contributions starting from the leaves of the data graph.

**4.3 Computing Instance Counts Using the Contribution-Graph – The Naive Approach** The basic idea of instance count computation process is to propagate relevant contributions, represented as the edge weights of the contribution graph, until the relevant instance counts are obtained. As we mentioned in Section 1, a major difference between PICC and other schemes is that PICC limits the run-time propagation process only to part of the data graph that is relevant to the given counting task  $AV_q = \{(v_i, \text{attr}(v_i)) \mid \text{attr}(v_i) \in A \text{ and } v_i \in V\}$ .

<sup>4</sup>In this paper, we use  $\langle a, b \rangle$  to represent a directed edge from node  $a$  to node  $b$ , while  $(a, b)$  denotes an undirected edge.

<sup>5</sup>We denote the relation that contains the tuple,  $n_i$ , as  $rl(n_i)$ .

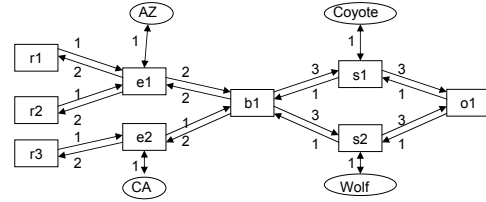


Figure 8: The contribution-graph for the example in Figure 5.

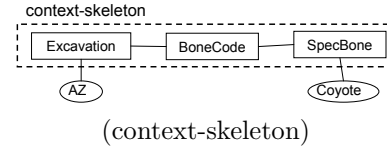


Figure 9: Context-skeleton for the motivating example

**4.3.1 Context-Skeleton** Given a context-graph  $\mathcal{G}_Q(N_Q, E_Q)$ , PICC identifies a subgraph  $\mathcal{G}_{csk}(N_{csk}, E_{csk}) \subseteq \mathcal{G}_Q$  that contains the relations and join edges sufficient to connect the set of attribute-values in  $AV_Q$ . This subgraph,  $\mathcal{G}_{csk} \subseteq \mathcal{G}_Q$  is referred to as the context-skeleton (Figure 3).

EXAMPLE 4.2. Figure 9 shows the context-skeleton that relates the two attribute-value pairs “species=’Coyote’” and “found\_in=’AZ’” in our running example. The context skeleton consists of three tables: SpecBone, BoneCode and Excavation:

$$SBE \leftarrow (\text{SpecBone} \bowtie_{b_{Type}} \text{BoneCode}) \bowtie_{b_{Code}} \text{Excavation}.$$

How many “species=’Coyote’” entries there are in the SpecBone table for each found\_in=’AZ’ entries in the Excavation table (and vice versa) can be estimated solely considering these three tables. On the other hand, as we have illustrated in Example 1.1, for correctly counting the instances where attribute-value pairs “species=’Coyote’” and “found\_in=’AZ’” occur together, we have to also account for the matching tuples in the Researcher relation:

$$\text{CountingContext} \leftarrow SBE \bowtie_{inst.} \text{Researcher}.$$

**4.3.2 Data-Skeleton** For the above example, existing join synopsis techniques, such as TuG [14], would require considering all tuples of all the relations in the database (to gather sufficient information to approximately reconstruct the Context relation). This however can be costly if there are many tables to include in the computation. PICC eliminates the need to consider the relations that do not directly help relate the attribute-value pairs that are in the counting task. In the above example, this would involve considering only the tuples in Excavation, BoneCode, and SpecBone tables.

Given a data-graph  $\mathcal{G}(N \cup V, E)$ , PICC identifies a subgraph  $\mathcal{G}_{dsk}(N_{dsk} \cup V_{dsk}, E_{dsk}) \subseteq \mathcal{G}$  that consists

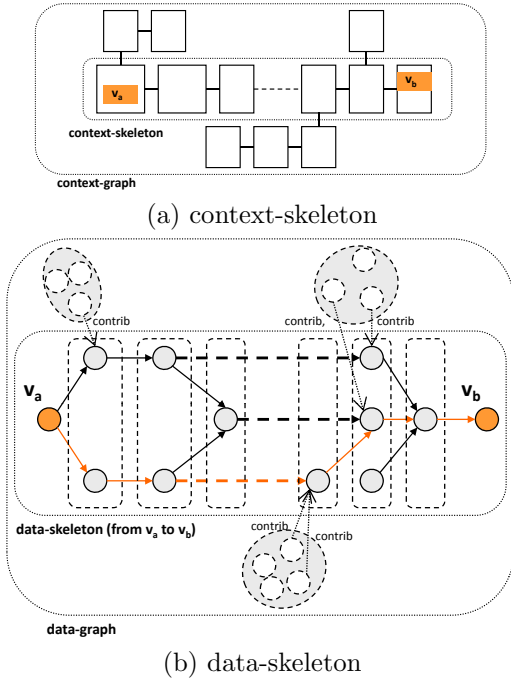


Figure 10: (a) A path-structured context-skeleton; (b) counts are propagated on the corresponding data-skeleton

only of the tuples of relations in  $\mathcal{G}_{csk}$  and the join edges between them. This subgraph,  $\mathcal{G}_{dsk} \subseteq \mathcal{G}$  is referred to as the *data-skeleton* graph (Figure 4). The run-time propagation is limited only to the *data-skeleton*, while the *contributions* of the relations that are not on the *data-skeleton* is read from a pre-computed lookup table representing the *contribution-graph*.

**4.3.3 Instance Counting on Path-Structured Context-Skeletons** We first focus on the special case when the instance counting task,  $AV_Q$ , contains only two attribute values,  $v_a$  and  $v_b$ . In this case, the context-skeleton is a path as shown in Figure 10 (a). Figure 10 (b) visualizes the propagation-based instance count computation process:

1. One of the values ( $v_a$  in this example) is selected as the propagation source.
2. The other ( $v_b$ ) is selected as the propagation destination where the pairs' count will be accumulated.
3. Instance counts are propagated from the source to the destination, while the *contributions* of the relations that are not on the context-skeleton are accounted for using the information encoded in the contribution graph.
4. At the end of the propagation process, the result to the  $AV_Q$  is accumulated at the destination,  $v_b$ .

As shown in Figure 10 (b), it is likely that there will be multiple data-paths that connect the given attributes

values. In this section, we describe a propagation process assuming that all relevant data-paths have been enumerated. In Section 5, we will relax this and describe how a propagation operator can be used to avoid enumeration of the individual data-paths. Let  $\mathcal{P} \subseteq \mathcal{G}_{dsk}$  be the set of paths that connect the given attribute-value pairs  $v_a$  and  $v_b$ . Let

$$p_h = v_a \cdot n_{h,1} \cdot n_{h,2} \cdots n_{h,l} \cdot v_b$$

be a path in  $\mathcal{P}$  and  $cnt(p_h)$  denote the number of times the tuple nodes in  $p$  appear together in the result. Then,

$$instCount(AV_Q) = \sum_{p_h \in \mathcal{P}} cnt(p_h).$$

We omit the proof of this due to space constraint. To compute the number,  $cnt(p_h)$ , we need to calculate the number of times the tuple nodes in  $p_h$  would appear together in the context-relation  $C_Q$  (if  $C_Q$  was enumerated):

$$cnt(p_h) = cnt^*(n_{h,1}) \times \cdots \times cnt^*(n_{h,l-1}) \times cnt^*(n_{h,l}).$$

where  $cnt^*(n_i)$  is the count of the node  $n_i$  computed based on the contributions of the relations that are not along in the context-skeleton:

$$cnt^*(n_i) = \prod_{(L_h, rl(n_i)) \in E_Q \setminus E_{csk}} \left( \sum_{n \in L_h \wedge (n, n_i) \in E} contrib(n, n_i) \right)$$

where  $cnt^*(n_i)$  is 1 when  $(L_h, rl(n_i)) \in E_Q \setminus E$  is null. Since the  $contrib()$  values are pre-computed, we can simply read these existing values from the contribution-graph look-up table. Thus, computing  $cnt(p_h)$  requires only considering the tuple nodes on the data-skeleton from  $v_a$  to  $v_b$  (Figure 10).

**4.3.4 Instance Counting on Tree-Structured Context-Skeletons** When there are more than two attribute-value pairs in the instance counting task,  $AV_Q$ , the corresponding context-skeleton is tree structured. As in the case of path-structured context-skeletons, there can be multiple ways to connect all the attribute-value nodes in the instance counting task. However, in this case, these do not result in data-paths but result in data-trees. Let  $\mathcal{T}$  be the set of all data-trees linking the attribute-value nodes in the task,  $AV_Q$ . Then, the instance count among attribute-values  $v_i \in V$  can be computed as

$$instCount(AV_Q) = \sum_{t_h \in \mathcal{T}} cnt(t_h),$$

where  $cnt(t_h)$  is the number of times the nodes in data-tree  $t_h$  appear together in the context relation.

If all the data trees have been enumerated, given  $t_h \in \mathcal{T}$ , let us randomly pick a  $root_{t_h} \in V$ . Then,  $leaves(t_h)$  is the set of remaining value nodes; i.e., each value node in  $V$  is assigned as a root or a leaf.  $cnt(t)$  can be computed by propagating contributions (i.e., pre-computed  $contrib()$  and  $match()$  values) from the leaves to the root through a reverse breathfirst (BFS) traversal. In next section, we describe how to avoid enumeration of all matching subtrees.

## 5 Computing Instance Counts without Enumerating all Paths and Trees

In Subsections 4.3.3 and 4.3.4, we assumed that all relevant sub-graphs in the data-graph linking the attribute values in the counting task are enumerated. This is obviously very expensive, and thus, undesirable. Hence, our first optimization will be to avoid having to enumerate all relevant data-graphs.

Given a context-skeleton  $\mathcal{G}_{csk}(N_{csk}, E_{csk}) \subseteq \mathcal{G}_Q$ , and a randomly selected root relation,  $R_{root} \in N_{csk}$ , let us consider a reverse BFS traversal of the relations in the context-skeleton. As in the data-graph enumeration methods, we will count instances by propagating contributions from leaves to the root in this reverse BFS order. However, in this case, the local instance counts are propagated (along the context-skeleton edges) among relations, instead of individual tuples. We define the *propagation operator* on two relations as follows:

**DEFINITION 5.1. (PROPAGATION OPERATOR)** *Let us be given two relations,  $R_a$  and  $R_b$ . For any tuple  $n_i$ , let  $agg(n_i)$  be a count aggregator associated to the tuple  $n_i$  (each  $agg(n_i)$  is initially set to 1). Then, the propagation operator,  $R_a \odot R_b$ , is such that*

$$agg(n_j) = agg(n_j) \times \sum_{(n_i, n_j) \in E_{csk} \wedge n_i \in R_a} agg(n_i) \times cnt^*(n_i)$$

Here,  $cnt^*(n)$  is obtained using the pre-computed contribution table as in Section 4.3.3.

PICC applies the propagation operator,  $\odot$ , in the reverse BFS traversal order, to accumulate the instance count information on the tuple nodes of the root relation. Thus, if  $R_m$  is the root of the counting skeleton, the instance count for the query can be computed as

$$instCount(AV_Q) = \sum_{n_i \in R_m} agg(n_i) \times cnt^*(n_i).$$

When computing the instance counts using the *propagation operator*, the number of propagation steps is bound by the number of edges in the data-skeleton.

**EXAMPLE 5.1.** *In our running example in Section 2, the instance count for  $AV_Q$  =*

$\{(Excavation.found\_in, "AZ"), (SpecBone.spe., "Coyote")\}$  is computed (using the contribution-graph in Figure 8) as follows: Let **Excavation** be the root relation. Initially, the count aggregator associated with the tuple nodes in the data-skeleton are set to 1. Then,  $agg(b_1)$  is computed as

$$agg(b_1) = agg(b_1) \times (agg(s_1) \times cnt^*(s_1)) = 1 \times (1 \times 1) = 1.$$

Using this, the value of  $agg(e_1)$  is computed as

$$agg(e_1) = agg(e_1) \times (agg(b_1) \times cnt^*(b_1)) = 1 \times (1 \times 1) = 1.$$

Since **Excavation** is the root relation, the final instance count is obtained as

$$\begin{aligned} instCount(AV_Q) &= \sum_{n_i \in Excavation} agg(n_i) \times cnt^*(n_i) \\ &= agg(e_1) \times cnt^*(e_1) = 1 \times 2 = 2. \end{aligned}$$

## 6 Estimating Instance Counts based on a Summarized Graph

In Sections 4 and 5, we described a method to discover instance counts relying on a data graph. While the pre-computed *contribution-graph* presented in Section 4.2 and the propagation operator presented in Section 5 help eliminate redundant computations in run-time, the process can still be prohibitively expensive for data sets which do not fit into the main memory. Thus, in this section, we discuss how to leverage graph summarization techniques along with the propagation technique presented in Section 5 to approximately discover the instance counts of a given set of attribute-value pairs.

**6.1 Summarization of the Data Graph** Summarization (or clustering) of graph-structured data has been extensively studied in the literature [15, 10]. Graph summarization algorithms can be classified into two: some (such as [9, 8]) partition the graph by finding strongly linked components and grouping these nodes into super-nodes. Others, such as [4, 12], find and cluster nodes that are similar to each other in terms of their connections to the rest (even if they are not connected to each other). The algorithm presented in this section summarizes the data graph by finding and merging nodes that have *similar* neighborhood patterns in the data graph. In a sense, the algorithm is related to the later type of clustering algorithms.

**6.1.1 Tuple-to-Cluster Similarity** One way to measure the neighborhood similarity between tuple nodes is to define the similarity in terms of their connections to the rest. Given two tuple nodes  $n_i$  and  $n_j$ , using Jaccard similarity coefficient, the structural similarity between them can be defined as

$$sim(n_i, n_j) = \frac{|E(n_i) \cap E(n_j)|}{|E(n_i) \cup E(n_j)|},$$

Figure 11: Graph summarization algorithm

---

**Input** : a data graph  $\mathcal{G}(N \cup V, E)$ , a set of relation  $R = \{R_1, R_2, \dots, R_m\}$  and a compression level  $\phi$   
**Output** : a summarized graph  $\mathcal{G}'(N' \cup V', E')$

1. Initialize current threshold ( $\theta$ ) to 1.
2. Initialize  $\mathcal{G}'(N' \cup V', E')$  as  $\mathcal{G}(N \cup V, E)$
3. **repeat**
4.     **for**  $i = 1$  to  $m$  do
5.     Apply agglomerative hierarchical clustering method to select nodes  $M = \{n_1, n_2, \dots, n_k\}$  in  $R_i$  that are merged into  $n'$  if  $\text{sim}(M, n') \geq \theta$
6.     Update edges,  $E'$
7.     **if** there is no change in  $N'$  and  $E'$
8.     decrease the current threshold  $\theta$
9. **until** the current compression level reaches to  $\phi$ ,
10. **return** summarized graph  $\mathcal{G}'(N' \cup V', E')$

---

where  $E(n)$  be the set of edges connected to node  $n$ . In this paper, we use a similar metric to measure the similarity between cluster nodes.

Let us assume that a set of  $k$  nodes  $n_1, n_2, \dots, n_k$  are chosen to be clustered into a single node in the summary. Naturally, by substituting one node for these  $k$  nodes, the individuality of  $k$ -nodes is lost and this will result in inaccuracy in the estimations of instance counts involving any of these nodes. Therefore, the critical issue in data graph summarization is to pick a similarity measure capturing how well the neighborhood patterns of summarized clustered data nodes match. Let us assume that we are given a data graph  $\mathcal{G}(N \cup V, E)$  and a set of  $k$  nodes  $M = \{n_1, n_2, \dots, n_k\}$  to be clustered. Let  $n'$  be a new node created by merging these  $k$  nodes. Then,  $E(n') = E(n_1) \cup E(n_2) \cup \dots \cup E(n_k)$ . Let also  $NE(R, E(n))$  is the number of edges in  $E(n)$  that connect between  $n$  with relation  $R$ . Given these, node-to-node Jaccard similarity leads to the following *similarity* between  $n'$  and  $n_i \in M$ :

$$\text{sim}(n_i, n') = \prod_{R_h \in \text{neighbors}(nl(n_i))} \frac{|NE(R_h, E(n_i))|}{|NE(R_h, E(n'))|}.$$

**6.1.2 TupleSet-to-Cluster Similarity** Using this definition, we can further compute the similarity of a given set of nodes  $M = \{n_1, n_2, \dots, n_k\}$  to the cluster node  $n'$  as follows:

$$\text{sim}(M, n') = \frac{1}{|M|} \times \sum_{n_i \in M} \text{sim}(n, n').$$

A high value of  $\text{sim}(M, n')$  denotes that merged nodes share similar data neighborhoods. Thus, our goal is to maximize  $\text{sim}(M, n')$  when merging data nodes.

**6.2 Data Graph Summarization** The data graph summarization algorithm used in PICC is presented

in Figure 11. In PICC, in order to chose the nodes to be merged, we employ agglomerative hierarchical based clustering algorithm [7]. However, we also note that other clustering algorithms can also be used. The algorithm starts with an initial data graph  $\mathcal{G}(N \cup V, E)$ , a set of relations  $R = \{R_1, \dots, R_m\}$ , and a compression level ( $\phi$ ). The algorithm returns a summarized graph  $\mathcal{G}'(N' \cup V', E')$ .

In the initialization step, the current threshold,  $\theta$ , is set to 1. For each relation  $R_i$ , the algorithm iteratively merges nodes  $M = \{n_1, n_2, \dots, n_k\}$  into  $n'$  if the similarity of a set of nodes  $M$  to the cluster node  $n'$  is greater than or equal to the current threshold and updates edge sets (line 5,6). The algorithm decreases the current threshold  $\theta$  if no nodes can be merged at the current level. This process iteratively is repeated until reaching to the desired compression level (line 9).

**6.3 Computing Instance Counts using Summarized Data Graph** Let  $\mathcal{G}'(N' \cup V', E')$  be a summarized data graph obtained from a given data graph  $\mathcal{G}$  based on the algorithm in Section 6.1. The first step for counting instances is to create a *contribution-graph* based on the summarized graph  $\mathcal{G}'$ . On the other hand, to reduce the errors due to the summarization, instead of using the process described in Section 4 as is, we inject corrective steps in the process.

**6.3.1 Revision of the Contribution-Graph Construction Process** The *contribution-graph*  $\mathcal{G}'_c(N'_c \cup V'_c, E'_c)$  based on a summarized data graph  $\mathcal{G}'(N' \cup V', E')$  is constructed similarly to Definition 4.1. The major difference in this case is that the edge weights are computed in a way that accounts for the potential loss of precision due to clustering of tuple nodes.

Given a cluster node<sup>6</sup>  $n \in N'$ , let  $s(n)$  be the set of tuple nodes in the original data graph clustered in  $n$ . The edge weights,  $\text{contrib}(n_i, v_j) = \text{contrib}(v_j, n_i)$ , between cluster node ( $n_i \in N'_c$ ) and value node ( $v_j \in V'_c$ ) are re-defined as follows:

$$\text{contrib}(n_i, v_j) = \frac{1}{|s(n_i)|} \times \sum_{n_u \in s(n_i)} \text{occur}(n_u, v_j),$$

instead of as  $\text{contrib}(n_i, v_j) = 1$  as in Definition 4.1. Here  $\text{occur}(n_u, n_v)$  is 1 if edge  $(n_u, n_v)$  exists in  $\mathcal{G}$  (otherwise,  $\text{occur}(n_u, n_v)$  is 0). In other words, unlike in the original contribution-graph construction process, the edge weight between cluster and the value node is weighted by the number of tuples with attribute-value  $v_j$  in the cluster. This is needed because, the cluster may consist of tuples with different values for the same

<sup>6</sup>In this paper, we use term “cluster node” to denote a node created by merging similar tuple nodes in the data graph.

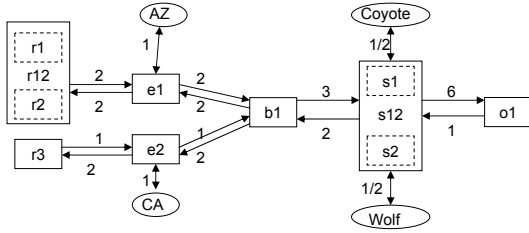


Figure 12: A summarized *contribution-graph* built for the example in Figure 8.

attribute. Thus,  $\text{contrib}(n_i, v_j)$  and  $\text{contrib}(v_j, n_i)$  is 1 only when all tuples in the cluster node have the same value,  $v_j$ , for the corresponding attribute.

The edge weights between cluster nodes,  $n_i, n_j \in N'_c$ , are also obtained by revising Definition 4.1 similarly: if tuple node  $n_i$  belongs to the leaf relation, then

$$\text{contrib}(n_i, n_j) = \frac{1}{|s(n_j)|} \times \sum_{n_u \in s(n_i)} \sum_{n_v \in s(n_j)} \text{occur}(n_u, n_v)$$

otherwise,

$$\text{contrib}(n_i, n_j) = \frac{1}{|s(n_j)|} \times \sum_{n_u \in s(n_i)} \sum_{n_v \in s(n_j)} \text{occur}(n_u, n_v) \times \prod_{L_h \in \text{neighbor}^*} \text{match}(L_h \rightarrow n_i)$$

EXAMPLE 6.1. Figure 12 shows a summarized contribution-graph obtained for our running example in Figure 5, assuming that  $s_{12}$  and  $r_{12}$  are cluster nodes created by the merge operation of the algorithm in Figure 11.

### 6.3.2 Revision of the Propagation Operator

The *propagation operation* presented in Definition 5.1 also needs to be re-defined to take into account the loss of precision due to clustering. Let  $R_a$  and  $R_b$  be two relations along the counting skeleton. The *propagation operator*,  $R_a \odot R_b$ , is revised such that

- for each cluster node  $n_a \in R_a$  and  $n_b \in R_b$ , during propagation the loss of precision from  $n_a$  to  $n_b$  should be multiplied by

$$\text{loss}(n_a, n_b) = \frac{1}{|s(n_b)|} \times \sum_{n_u \in s(n_a)} \sum_{n_v \in s(n_b)} \text{occur}(n_u, n_v).$$

- for each cluster node  $n$  that consists of tuples having different values for the same attribute,  $\text{contrib}(n, v_n)$  needs to be multiplied. Thus, if  $n$  is directly linked to the attribute-value,  $v_n$ , in  $AV_Q$ , the count aggregator associated with  $n$  is initialized as  $\text{contrib}(n, v_n)$ . Otherwise, it is set to 1.

The aggregated instance counts (i.e., the *agg* values), are accumulated in the tuple nodes of the root relation of the counting skeleton by applying the revised propagation operator in the reverse BFS traversal order. If  $R_m$  is the root of the counting skeleton, PICC estimates the instance count for the query,  $AV_Q$ , as

$$\text{instCount}_{est}(AV_Q) = \sum_{n_i \in \mathcal{R}_m} \text{agg}(n_i) \times \text{cnt}^*(n_i) \times |n_i|.$$

EXAMPLE 6.2. Using the summarized contribution-graph in Figure 12, the instance count for  $AV_Q = \{(\text{Excavation.found\_in}, \text{"AZ"}), (\text{SpecBone.spec.}, \text{"Coyote"})\}$  is estimated as follows: First initialized:

$$\text{agg}(s_{12}) = \text{contrib}(s_{12}, \text{"Coyote"}) = 0.5.$$

Then,  $\text{agg}(b_1)$  is updated:

$$\begin{aligned} \text{agg}(b_1) &= \text{agg}(b_1) \times (\text{agg}(s_{12}) \times \text{cnt}^*(s_{12}) \times \text{loss}(s_{12}, b_1)) \\ &= 1 \times (0.5 \times 1 \times 2) = 1 \end{aligned}$$

Since,  $e_1$  is linked to the attribute-value, "AZ", it is initialized as:

$$\text{agg}(e_1) = \text{contrib}(e_1, \text{"AZ"}) = 1.$$

Then,  $\text{agg}(e_1)$  is updated as

$$\begin{aligned} \text{agg}(e_1) &= \text{agg}(e_1) \times (\text{agg}(b_1) \times \text{cnt}^*(b_1) \times \text{loss}(b_1, e_1)) \\ &= 1 \times (1 \times 1 \times 1) = 1. \end{aligned}$$

Finally, this is propagated to the cluster  $r_{12}$ :

$$\text{instCount}_{est}(AV_Q) = \text{agg}(e_1) \times \text{cnt}^*(e_1) \times |e_1| = 2.$$

## 7 Improving Estimates based on Available Query/Result Samples

As we will see in the experiments section, propagation of the instance counts over the summarized contribution-graph is faster than performing the same task on the original data graph. However, obviously, this comes with the penalty of a certain degree of reduced precision. One way to further minimize the estimation error is to correct the initial estimates by taking into account the *expected error rates* computed based on query/result samples (if they are available). Given a query  $AV_Q$  against a context relation  $C_Q$ , the *relative estimation error*,  $\text{err}(C_Q, AV_Q)$ , can be computed as

$$\frac{|\text{instCount}(C_Q, AV_Q) - \text{instCount}_{est}(C_Q, AV_Q)|}{\text{instCount}(C_Q, AV_Q)}.$$

Then, given a set,  $\mathcal{A}_Q$ , of query/result samples, the *average relative estimation error*,  $\text{avg\_err}(\mathcal{A}, C_Q)$ , can be computed similarly

$$\text{avg\_err}(C_Q, \mathcal{A}_Q) = \frac{1}{|\mathcal{A}_Q|} \times \sum_{AV_{Q,i} \in \mathcal{A}_Q} \text{err}(C_Q, AV_{Q,i}).$$

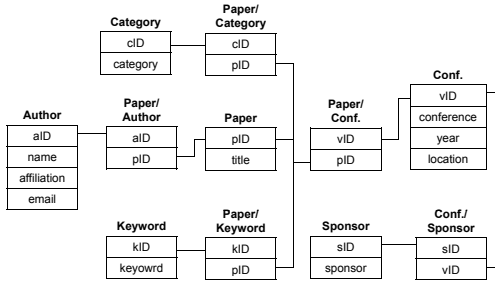


Figure 13: Schema of the database used in experiments  
 Given an  $avg\_err(\mathcal{A}, C_Q)$  value computed based on a set,  $\mathcal{A}_Q$ , of available query samples, PICC adjusts the estimated instance counts for a new query  $AV_Q$  as follows:

$$instCount_{adj}(C_Q, AV_Q) = \frac{instCount_{est}(C_Q, AV_Q)}{1 - avg\_err(C_Q, \mathcal{A})}.$$

Note that PICC adjusts the estimate assuming that the instance counts are underestimated. This is because the graph summarization process usually results in losses that lead to underestimations in instance counts. In Section 8.3, we will evaluate the effectiveness of adjustments for real data sets.

## 8 Experiments

In this section, we describe the experiments we carried out to evaluate the effectiveness and efficiency of PICC approach to counting. In particular,

- in Section 8.1, we compare PICC proposed in this paper with state-of-the-art method, *TuG*, without the graph summarization,
- in Section 8.2, we show that PICC correctly estimates instance counts on the summarized-graph without having to rely on costly join operations, and
- in Section 8.3, we show that PICC achieves a significant time gain against existing solutions with the graph summarization.

**Experimental Setup:** We evaluated PICC with a data set collected from ACM digital library [1] on Feb. 2008. The data set consists of 11 relations whose schema is shown in Figure 13. This database contains 109,490 tuples (before the join). As the context relation,  $C_Q$ , we use a single table obtained by joining all relations in Figure 13. We compute instance counts using following alternatives:

- **case *TuG<sub>csk</sub>*** [14]: Here, like PICC, the *TuG* graph is generated by considering join relations connecting all attribute values in instance counting tasks.
- **case *TuG<sub>full</sub>***: In this case, query graph is generated using the full *context-graph*. That is, a query graph contains all join relations provided by the context relation  $C_Q$ .

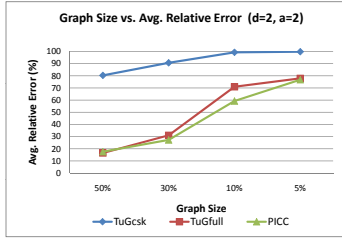
- **case PICC:** This is the proposed scheme. In this case, instance counts are computed based on the graph summarization technique and the propagation scheme in Section 6.

In order to evaluate the above three methods, we used the relative estimation error described in Section 7. We ran all experiments on a machine with 2.33 GHz of CPU.

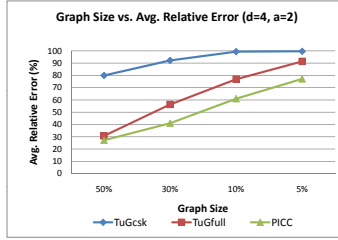
**8.1 Efficiency** In this section, we test the efficiency of proposed approach on real data sets. First, we compare the construction time of a context relation and a data graph (Figure 13). In both cases, we use the same indexes and tables to perform joins in *MySQL* DBMS. Under identical conditions, it took 173 minutes to create the full context relation by joining all tables. In contrast, it took only 26 seconds to construct the data-graph. While it is expected that performing a full join across all tables is more expensive than performing pairwise joins to create the data-graph, the size of difference is striking and motivated propagation-based schemes for counting. *Contribution-graph* construction, the second step in PICC, took 34.5 seconds. Note that this is a one-time cost for the given context-graph and negligible against the full-join cost. For comparison with *TuG*, we randomly generated 50 instance counting tasks, each containing two attribute-value pairs of join distance 6. On the average, PICC took 61 milliseconds per instance counting task, while *TuG<sub>full</sub>* took 1837 milliseconds. This shows that PICC effectively leverages the pre-computed *contribution-graph* for situations when the user will perform a large number of counting tasks within the same context. Moreover, as we will see next, PICC performs significantly better than *TuG* in memory constrained environments.

**8.2 Effectiveness on Summarized Graph** In previous experiment, we assumed that the whole of the data graph can be loaded into the main memory. In many applications, however, it is impossible to load the data into the memory. Thus, in this subsection, we compare the effectiveness of PICC and *TuG* in memory constrained environments by constraining the amount of the data-graph loaded into the memory to 5% to 50% of the original graph.

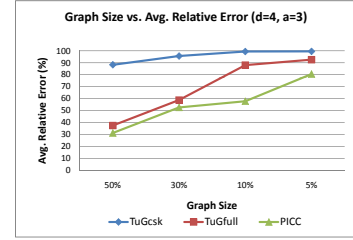
In the experiment, we generated three different instance counting task sets. The first two task sets contain two attribute values that are connected by tuples whose join distance is 2 and 4 joins respectively. For the evaluation of the propagation scheme in tree-structured context-skeletons, we also generated an instance counting task set containing three attribute values whose join distance is 4. Each task set contains randomly generated 500 task queries. Figure 14 shows the average rel-



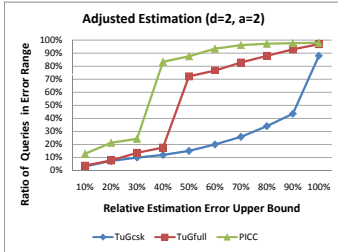
(a) Join dist.=2, Attr. Num=2



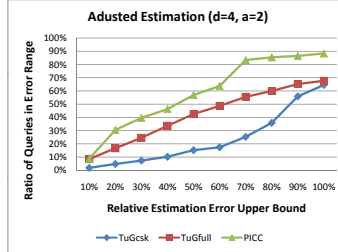
(b) Join dist.=4, Attr. Num=2



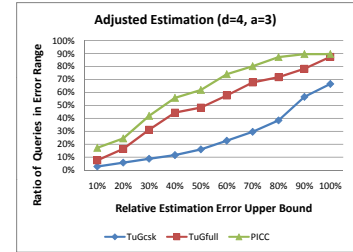
(c) Join dist.=4, Attr. Num=3

Figure 14: Average relative estimation error vs. graph size for three query sets (*the lower the curve, the better are the results*)

(a) Join dist.=2, Attr. Num=2



(b) Join dist.=4, Attr. Num=2



(c) Join dist.=4, Attr. Num=3

Figure 15: Adjusted estimation using 10% sample queries (*the higher the curve, the better are the results*): the results reported in this figure are obtained when 10% graph size is used for three alternatives.

ative estimation error of the algorithms under various summarization levels (the *graph size* is defined as the ratio of the number of nodes in the summarized graph to the number of nodes in the original graph). In the experiment, the threshold ( $\theta$  in Figure 11) of data graph summarization algorithm decreases by 5% on every iteration. Key observations based on Figure 14 can be summarized as follows:

- When using only the context-skeleton,  $TuG$ 's average error rate is close to 100% for highly summarized graphs. PICC, on the other hand, does not have this problem as it can leverage the *summarized contribution-graph* well.
- PICC, on the other hand, outperforms the state-of-the-art method,  $TuG_{full}$ . This implies that the PICC is more suitable for computing instance counts under memory constraints than  $TuG$ .
- As the join distance and the number of attribute values in the counting task increase, the performance gap between PICC and  $TuG_{full}$  also increases. Furthermore, the accuracy gains of PICC over existing solutions get larger as the size of the memory decreases.

We also studied the suitability of the three alternatives when samples for adjusted estimations are available. For this experiment, we randomly generated and executed 50 sample counting tasks for each configuration,

Table 1: Graph summarization time (sec)

	Output Size (rel. original graph)			
	5%	10%	30%	50%
Based on agglomerative hierarchical clustering	532	529	523	473
Based on K-means	211	243	340	462

computed the *average relative estimation error*, and adjusted estimations presented in Figure 14 based on this information. To compare the three schemes, in Figure 15, we use the *ratio of queries within a given relative estimation error*,  $e$ :

$$\frac{\#queries\ whose\ adjusted\ estimation\ is\ within\ e}{\#total\ queries}$$

The figure plots the relative estimation error upper bound versus *Ratio of Queries within Relative Error*. As shown in the figure, PICC outperforms the other approaches, which implies that the estimation error of PICC can be corrected more effectively leveraging a statistical process than  $TuG_{csk}$  and  $TuG_{full}$ .

**8.3 Efficiency on Summarized Graphs** In this section, we test the efficiency of the proposed approach on summarized graphs.

First of all, Table 1 shows the graph summarization times under various output graph size constraints. For these experiments, we tried two well-known algorithms, agglomerative hierarchical clustering and K-means, to

Table 2: Construction time for contribution-graphs

	Graph Size (rel. original graph)			
	5%	10%	30%	50%
Construction Time (sec)	5.6	6.5	7.5	9.1

pick the tuples to be merged for the algorithm in Figure 11. As can be seen in the table, the graph summarization algorithm based on K-means is faster than that based on agglomerative hierarchical clustering and should be used when cost of summarization is a concern. However, we observed that the quality of the summarized graph is better when agglomerative hierarchical clustering is used.

Table 2 shows the *contribution-graph* construction time on varying sizes of summarized graphs. As expected the construction time decreases as the size of the summarized graph decreases.

Finally, we evaluate the counting performance of alternative schemes:  $TuG_{csk}$ ,  $TuG_{full}$ , and PICC. There are two main factors that affect the execution time: the graph size and the distance between the pair of attribute values in the data graph.

- *The effect of the graph size on the execution time:* In order to study the effect of the graph size on the execution time, we varied the summarized graph size from 5% to 50% of the original graph and randomly generated 50 counting tasks containing two attribute-value pairs of join distance 2.
- *The effect of distance between a pair of attribute values on the execution time:* For this experiment, we randomly generated three task sets that contain two attribute-value pairs with join distance 2, 4, and 6, respectively. Each task set contains 50 instance counting task queries and a graph size of 10% is used for this experiment.

As can be seen in Figure 16,  $TuG_{full}$  performs poorly because it needs to traverse all *context relations*, while both  $TuG_{csk}$  and PICC traverse only those relations that connect attribute values in the query. In fact,  $TuG_{csk}$  and PICC requires almost the same execution time. However, as described in Section 8.2,  $TuG_{csk}$  performs very poorly in estimating the instance count. Thus, we can conclude that, *PICC provides the best performance both in query execution time and in the estimation of the instance counts.*

## 9 Conclusion

In this paper, we proposed a novel PICC technique for estimating the instance counts of sets of attribute-value pairs in complex multi-table databases. Our proposed approach relies on a contribution-graph data structure to reduce the time required for instance counting tasks and to provide robustness against errors when the data

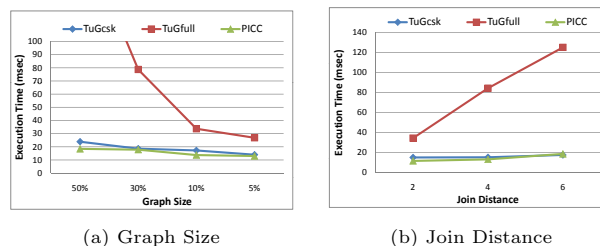


Figure 16: Execution times on varying (a) graph sizes and (b) join distances

is summarized into a concise synopsis to fit into the memory. The experiment results show that the proposed technique (PICC) provides significant execution time and accuracy gains over the existing solutions. Experiments also showed that PICC can leverage available query result samples to adjust its counts estimates more effectively than the other schemes.

## References

- [1] ACM Digital Library. <http://portal.acm.org/dl.cfm>.
- [2] S. Acharya, P.B. Gibbons, V. Poosala and S. Ramaswamy. Join Synopses for Approximate Query Answering. In *SIGMOD*, 1999.
- [3] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *VLDB*, 1994.
- [4] G. Aggarwal, T. Feder, K. Kenthapadi, S. Khuller, R. Panigrahy, D. Thomas and A. Zhu. Achieving Anonymity via Clustering. In *SIGMOD*, 2006.
- [5] K. Chakrabarti, M. Garofalakis, R. Rastogi and K. Shim. Approximate Query Processing Using Wavelets. In *VLDB*, 2000.
- [6] S. Chaudhuri, R. Motwani and V. Narasayya. On Random Sampling over Joins. In *SIGMOD*, 1999.
- [7] W.H.E. Day and H. Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification*, v.1 n.1, p.7-24, 1984.
- [8] I.S. Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *SIGKDD'01*.
- [9] D. Harel and Y. Koren. On Clustering Using Random Walks. In *FSTTCS*, 2001.
- [10] G. Jeh and J. Widom. SimRank: A measure of structural-context similarity. In *SIGKDD*, 2002.
- [11] L. Getoor, B. Taskar and D. Koller. Selectivity Estimation using Probabilistic Models. *SIGMOD'01*.
- [12] K. Lefevre, D.J. Dewitt and R. Ramakrishnan. Incognito: efficient full-domain K-anonymity. *SIGMOD'05*.
- [13] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal and M.C. Hsu. PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. In *ICDE*, 2001.
- [14] J. Spiegel and N. Polyzotis. Graph-Based Synopses for Relational Selectivity Estimation. In *SIGMOD*, 2006.
- [15] X. Yin, J. Han and P.S. Yu. LinkClus: Efficient Clustering via Heterogeneous Semantic Links. In *VLDB*, 2006.