TuG Synopses for Approximate Query Answering

JOSHUA SPIEGEL
BEA Systems
and
NEOKLIS POLYZOTIS
University of California at Santa Cruz

This paper introduces the Tuple Graph (TuG) synopses, a new class of data summaries that enable accurate approximate answers for complex relational queries. The proposed summarization framework adopts a "semi-structured" view of the relational database, modeling a relational data set as a graph of tuples and join queries as graph traversals respectively. The key idea is to approximate the structure of the induced data graph in a concise synopsis, and to approximate the answer to a query by performing the corresponding traversal over the summarized graph. We detail the TuG synopsis model that is based on this novel approach, and we describe an efficient and scalable construction algorithm for building accurate TuGs within a specific storage budget. We validate the performance of TuGs with an extensive experimental study on real-life and synthetic data sets. Our results verify the effectiveness of TuGs in generating accurate approximate answers for complex join queries, and demonstrate their benefits over existing summarization techniques.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing*; G.2.1 [**Discrete Mathematics**]: Combinatorics—*Combinatorial algorithms*

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Data synopses, approximate query processing, selectivity estimation

1. INTRODUCTION

Consider a relational query optimizer, faced with the challenging task of optimizing a complex join query. In order to approximate effectively the cost factors of a candidate plan, the optimizer needs to obtain accurate estimates on the sizes of results that are generated by operators, or equivalently, accurate *selectivity estimates* for the corresponding query expressions. These estimates are typically provided by *data synopses* (commonly referred to as "data statistics" or "summary structures") that approximate the underlying data distribution and can thus estimate the number of results generated by a query. The accuracy of these data synopses is therefore crucial for the effectiveness of the optimization process.

The aforementioned problem of selectivity estimation can be classified under the more general problem of approximate query answering. At an abstract level, the

goal is to process a query over a data synopsis (treating the synopsis as a highly compressed and lossy version of the database) and to compute an answer that approximates the true query result. Approximate query answering can mitigate the increased cost of query evaluation over large data sets and is becoming increasingly popular in the context of interactive data exploration. As an example, consider a query that computes an average aggregate over the results of a complex query. By inspecting the approximate answer, the user can gauge whether the query is interesting and therefore whether it is useful to "invest" in its evaluation. In certain cases, it may even suffice to obtain only the approximate answer, when accuracy to the last decimal digit is not required.

Clearly, the effectiveness of approximate query answering is tied to the accuracy of the underlying data synopses, which in turn depends on the synopses' capacity to capture in limited space the main correlations that exist in the data. Consider, for instance, a simple movie database that consists of three tables, namely, `Movies`, `Actors`, and `Cast`. `Cast` has foreign keys to the other two tables, and essentially records in which movie each actor appeared along with their wages. In this simple database, an example correlation across joins might be that movies in the last decade have typically higher wages for their actors. In other words, there is a dependency between the distribution of different values through the chain of joins. Another example may be that movies released in the 90s tend to have more actors than movies made in the 50s. Here, the number of join results is affected by a selection on the value of one of the participating tables.

Table-level synopses, such as, histograms [Poosala et al. 1996] or wavelets [Matias et al. 1998], are typically ineffective in capturing these complex join-based correlations as they focus on the summarization of a single table at a time. This has led to the introduction of *schema-level* synopses, e.g., Join Synopses [Acharya et al. 1999] and Probabilistic Relational Models [Getoor et al. 2001], that enable accurate approximations by summarizing the *combined* join and value distribution across *several tables*. The proposed techniques, however, are not applicable to the class of relational schemata that contain many-to-many relationships. Such a schema is our toy movie database, where a single movie is associated with multiple actors and vice versa through a chain of foreign-key joins. This type of join relationship is very common in real-world applications, and hence providing effective summarization techniques for such schemata is an important and practical problem; at the same time, however, the presence of many-to-many joins greatly complicates the potential statistical correlations, introducing significant new challenges to the summarization problem.

Motivated by these observations, we tackle the problem of schema-level data synopses for data sets with arbitrary join relationships. We describe the class of *Tuple Graph* synopses (TuGs for short) that rely on graph-based models in order to summarize the combined distribution of joins and values in a relational database. The inspiration for our proposed solution comes from an unlikely source, namely, the field of XML summarization. Conceptually, we adopt a "semi-structured" view of a relational database, where tuples and joins become nodes and edges respectively in a data graph, so that join queries now correspond to graph traversals. With this view in mind, we propose to summarize the structure of this data graph in order to

derive accurate approximate answers for complex queries. To address the specifics of relational summarization, however, our work relies on novel techniques that form a clear departure from previously proposed XML models.

As we show in this paper, our TuG synopses enable accurate approximate answers for a large class of practical join queries: queries that compute standard SQL aggregates (i.e., *MIN*, *MAX*, *AVG*, *SUM*, and *COUNT*), having complex join graphs (e.g., containing cycles and many-to-many relationships), and applying several selection predicates on different tables. To the best of our knowledge, the TuG model is the first schema-level technique to enable the combined summarization of joins and values for schemata of such complex join relationships. More concretely, the contributions of our work can be summarized as follows:

• TuG **Synopsis Model.** We introduce the TuG synopses for summarizing the combined join- and value-distribution of a relational database. Our novel TuG model promotes joins and values to an equal status and thus allows a uniform treatment of their potential correlations in the underlying data distributions. We demonstrate several key properties of the TuG model and develop a systematic framework for the generation of approximate answers to aggregate queries.

• TuG **Construction.** We introduce an efficient construction algorithm, termed TuGBuild, for building accurate TuGs given a specific storage budget. The proposed algorithm takes advantage of the unique properties of our TuG model in order to derive a highly-compressed, yet accurate approximation of the original data distribution. A key feature is the use of disk-based processing techniques that enable TuGBuild to scale to large data sets under limited memory constraints. The presented techniques are of general interest, as they can be transferred to the XML domain and thus enable the scalable summarization of large XML data sets.

• **Experimental Evaluation of** TuG**s.** We conduct an extensive empirical study to validate the effectiveness of our proposed TuG model. Our results on synthetic and real-life data sets verify the effectiveness of TuG summaries as schema-level data synopses. Moreover, our study demonstrates the scalability of our approach and its many advantages over previously proposed summarization techniques.

The remainder of the paper is organized as follows. Section 2 provides a formal definition of the summarization problem and introduces some necessary notation. Section 3 presents in detail the proposed TuG model and describes the computation of approximate answers over a concise TuG summary. The construction of TuG synopses is covered in Sections 4 and 5. Section 6 presents the results of our experimental study for evaluating the effectiveness of TuG summaries. We cover related work in Section 7 and conclude with Section 8.

2. PRELIMINARIES

Data Model. Our work focuses on the summarization of a relational database with a set of relation names $\mathcal{R} = \{R_1, \ldots, R_n\}$. We assume that the schema of each relation R_j, $1 \leq j \leq n$, comprises a set of value attributes $\mathcal{A}_j$ and a set of join attributes $\mathcal{J}_j$. Join attributes are used to define join relationships among relations, e.g., using key/foreign-key constraints, and are not involved in selection predicates

over the relation. Value attributes, on the other hand, encode information relevant to the semantic concept captured by each relation, and are thus used naturally in selection predicates. This distinction is common in real-world applications, where join relationships are typically expressed in terms of key and foreign key attributes. Moreover, the two attribute sets tend to be disjoint, i.e., $\mathcal{A}_j \cap \mathcal{J}_j$ is empty. We adopt this assumption to simplify our presentation, but we note that it is not necessary for the techniques that we develop in our work. In what follows, we use $\mathcal{A} = \cup_{j=1}^{n} \mathcal{A}_j$ to denote the set of value attributes over all the relations of the schema. In the context of our work, we assume that a value attribute can be either numerical or categorical.

We represent schema information as an undirected graph G_S, termed the *schema graph*. The node-set of G_S is the set of relation and attribute names $\mathcal{R} \cup \mathcal{A}$. The graph contains an edge (R, A) for each relation R in $\mathcal{R}$ and corresponding value attribute A in $\mathcal{A}$. Moreover, G_S contains edges that encode predefined join relationships among the relations. More specifically, given distinct relations R and R' in $\mathcal{R}$, the graph contains an edge (R, R') if the two relations have a join relationship that can be encoded as an equi-join over the corresponding join attributes. Such pre-defined join relationships are part of the meta-data of the database, and they can be either specified by the DBA or inferred automatically, e.g., by interpreting the key/foreign key constraints of the schema or analyzing a sample workload. We focus on equi-joins, since they are prevalent in real-world applications. We do not place any restrictions on the number of joins per relation or the type of each join, e.g., one-to-one, one-to-many, or many-to-many. Overall, the schema graph G_S records the value attributes for each relation and also a set of join relationships among relations. In this sense, G_S can be viewed as a template for the meaningful queries that can be evaluated over the database. In what follows, we use $neighbors(R)$ to denote the attributes and relations that are linked to R in the schema graph.

EXAMPLE 2.1.: Figure 1(a) shows the schema definition for a database that records information on movies. In this example, the `Movies` relation has two value attributes (`year` and `genre`) and a join attribute `mid` that defines an equi-join to another relation `Cast`. Overall, there are two join relationships that can be inferred by the constraints of the schema: `Movie` $\bowtie$ `Cast` and `Cast` $\bowtie$ `Actors`. The corresponding schema graph is shown in Figure 1(b). ∎

An instance of the database schema is represented in a similar fashion, using the concept of a *data graph*. More formally, let D be a database instance, let $\mathcal{T}_D$ denote the set of tuples in the instance, and let $\mathcal{V}_D$ denote the set of values that appear in the value attributes of the tuples. The data graph G_D is an undirected graph over the node-set $\mathcal{V}_D \cup \mathcal{T}_D$. Given a tuple τ of relation R and value ν in the active domain of an attribute A, the edge (τ, ν) appears in G_D if the edge (R, A) appears in the schema graph and τ has value ν for attribute A. Similarly, given tuples τ of R and τ' of R', the edge (τ, τ') appears in G_D if there exists an edge (R, R') in the schema graph G_S and the tuples join according to the corresponding join relationship. Intuitively, therefore, the data graph can be seen as a specific instantiation of the schema graph, where relation names are substituted with tuples and attribute names with values. An example data graph is shown in Figure 1 for

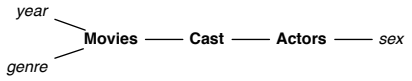
a sample instance of the movie database.

```
CREATE TABLE Movies (
    mid INTEGER PRIMARY KEY,
    genre VARCHAR(40),
    year INTEGER );

CREATE TABLE Actors (
    aid INTEGER PRIMARY KEY,
    sex CHAR(1) CHECK( sex in ('M','F') ) ) ;

CREATE TABLE Cast (
    mid INTEGER REFERENCES Movie,
    aid INTEGER REFERENCES Actor,
    PRIMARY KEY (mid,aid));
```

(a)



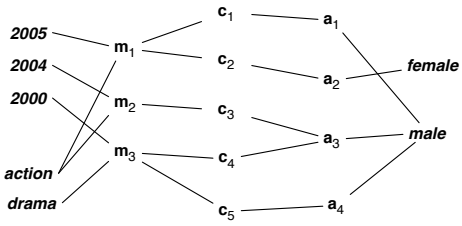
(b)

Movies	*mid*	*year*	*genre*
	1	2005	Action
	2	2004	Action
	3	2000	Drama

Cast	mid	aid
	1	1
	1	2
	2	3
	3	3
	3	4

Actors	*aid*	*sex*
	1	Male
	2	Female
	3	Male
	4	Male

(c)



(d)

Fig. 1. An example database schema and instance: (a) Database schema, (b) Schema Graph G_S, (c) Sample instance, (d) Corresponding Data Graph.

Overall, the data graph provides a semi-structured view of a flat relational data set, which enables in turn the interpretation of relational queries as graph traversals. Later, we show how this observation forms the cornerstone of the TuG summarization framework that we present in this paper. To simplify notation, we use D to denote both the database instance and the corresponding data graph.

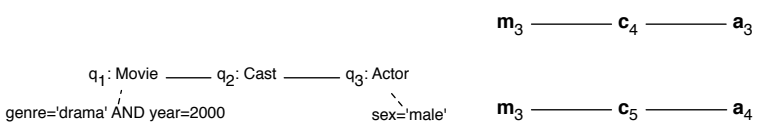
Query Model. We focus on SQL queries that compute an aggregate over the join of a subset of the relations. Without loss of generality, we assume that the relations joined are $R_1, \ldots, R_K$. A query can thus be described as follows:

SELECT $Aggr$
FROM R_1, R_2, …, R_K
WHERE $\bigwedge_{1 \le i \le K} C_i$ AND $\bigwedge_{1 \le i \ne j \le K} R_i \bowtie R_j$

Each C_i denotes a (potentially empty) conjunction of selection predicates on a subset of the attributes of R_i. We assume that a selection predicate specifies

```
SELECT COUNT(*)
FROM Movies m, Cast c, Actors a
WHERE m.year=2000 AND m.genre='drama' AND a.sex='male' AND
    m.mid = c.mid AND c.aid = a.aid
```

(a)



(b)

(c)

Fig. 2. Example query: (a) SQL Specification , (b) Query graph Q, (c) The possible matchings over the data graph of Figure 1.

either a range condition over a numerical attribute, or an IN condition over a categorical attribute (e.g., `genre IN {action, adventure, drama }`). The query also specifies a conjunction of binary join predicates that are compatible with the join relationships specified in the schema graph. In what follows, we refer to the combination of the FROM and WHERE clause as the select-join expression of the query. Finally, $Aggr$ is an aggregate function that is computed over the tuples in the output of the select-join expression. We consider the common aggregate functions that are part of the SQL language, i.e., $COUNT$, SUM, AVG, MIN, and MAX. Overall, the query template captures a large class of queries that arise naturally in interactive data exploration and are thus amenable to approximate query answering. Moreover, the $COUNT$ aggregate is of particular interest, since the resulting query essentially computes the selectivity of the select-join expression. This selectivity is an important cost factor for the evaluation of the select-join expression, and thus the computation of the $COUNT$ aggregate has obvious significance in the context of query optimization.

We represent a query as a pair $(Aggr, Q)$, where Q is an undirected graph that encodes the corresponding select-join expression. Formally, Q has the node-set $T_Q = \{q_1, \ldots, q_K\}$, where each query node q_i corresponds to relation R_i appearing the select-join expression. Given nodes q_i and q_j, the query graph contains the edge (q_i, q_j) if the query contains a join predicate between R_i and R_j. Moreover, each variable q_i is annotated with the conjunction of selection predicates over the corresponding relation. Figure 2 shows an example query over the sample movie database and the corresponding query graph. (We use dashed-lines to show condition annotations.)

The concept of a query graph allows us to define the result of the select-join expression as a sub-graph matching problem. More formally, a *matching* h of Q in D is a mapping from query nodes to data nodes that satisfies the following properties: (a) $h(q_i)$ is a tuple node of relation R_i such that its values satisfy the predicates attached to q_i, (b) each edge (q_i, q_j) of Q is mapped to a data graph edge $(h(q_i), h(q_j))$. Intuitively, a matching is the graph-based representation of a single tuple in the output of the select-join expression, containing the witnesses in the base relations and how they join. An example matching is shown in Figure 2(c).

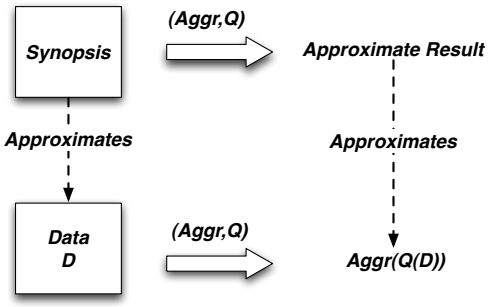


Fig. 3. Schematic overview of approximate query answering.

It is straightforward to show that there exists an isomorphism between the set of unique matchings and the output of the select-join expression. In turn, the result of Q can be computed by evaluating $Aggr$ over the values that appear in the unique matchings of Q. Returning to the example of Figure 2, the result of the query is computed as the count of distinct matchings of Q. In what follows, we use $Q(D)$ to denote the set of unique matchings and $Aggr(Q(D))$ for the result of the query.

On a final note, we observe that our query model precludes queries whose FROM clause references the same relation more than once. This restriction is necessary in order to prove certain properties of the techniques that we develop[1]. Our techniques can still be applied on queries whose FROM clause references the same relation several times, except that we can no longer guarantee the same properties.

Problem Definition. In this paper, we tackle the computation of approximate answers in the aforementioned data and query model. Following common practice, we adopt an approach where the approximate answer is generated by processing the query over a concise *data synopsis*. The general process is depicted in Figure 3. The synopsis captures the main statistical traits of the data that are relevant for the computation of approximate answers, and its size is bounded by a storage budget B that is much smaller than the size of the database. Thus, our goal of approximate query answering involves two interrelated problems:

Generation of Approximate Answers. Given a synopsis of database instance D and a query $(Aggr, Q)$, approximate the result $Aggr(Q(D))$.

Synopsis Construction. Given a database instance D and a storage budget B, construct a synopsis of D of size at most B.

Given that the storage budget B is much smaller than the data, we expect the synopsis to represent a lossy compression of the data. As a result, the generation of approximate answers typically involves the application of certain assumptions that are designed to compensate for the lost information. The validity of these assumptions affects greatly the accuracy of the synopsis, and thus an effective construction algorithm must balance out the loss of information with respect to the assumptions that have to be applied during approximate query answering. It

[1]Specifically, the lossless property of all-but-one similar merges (Section 4).

becomes clear, therefore, that the two problems must be tackled jointly in order to develop an accurate summarization framework.

3. TUG SYNOPSES

In this section, we introduce the TuG synopsis model that is the focus of this paper. We begin with the formal definition of the model, and then discuss the computation of approximate answers over a concise TuG synopsis. The problem of synopsis construction is discussed in the sections that follow.

3.1 Synopsis Model

The intuition behind TuG*s* is that the computation of $Q(D)$ is inherently linked to the structural characteristics of the data graph G_D. Thus, the key idea is to approximate the structure of G_D in a concise synopsis, and to generate an approximation of $Q(D)$ by matching Q over the summarized graph.

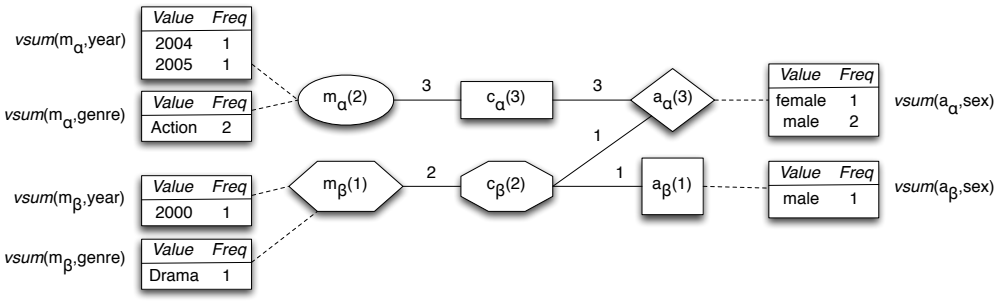
The TuG model employs a graph summarization framework that partitions the tuples of the data graph G_D and stores aggregate statistical information for each partition. Formally, let $T_{\mathcal{S}}$ be a partitioning of the set of tuples in G_D such that each partition r in $T_{\mathcal{S}}$ corresponds to a subset of tuples from the same relation. We define the extent of a partition r as the set of tuples that it represents. To simplify our notation, we use r to refer both to the partition and its extent, and hence $|r|$ is the size of the partition.

Consider two partitions r and s corresponding to relations R and S respectively. We say that r and s *join* if there are tuples $\tau_r \in r$ and $\tau_s \in s$ such that the edge (τ_r, τ_s) appears in the data graph, i.e., τ_r joins with τ_s. We use $|r \bowtie s|$ to denote the number of joining tuples in the cross product $r \times s$, or equivalently, the number of data graph edges between the tuples of r and s. Each partition also implies a distribution of values for the attributes of the corresponding relation. In what follows, we use $values(r, A)$ to denote the frequency distribution of the values of A in the tuples of r (assuming of course that A is an attribute of the corresponding relation R).

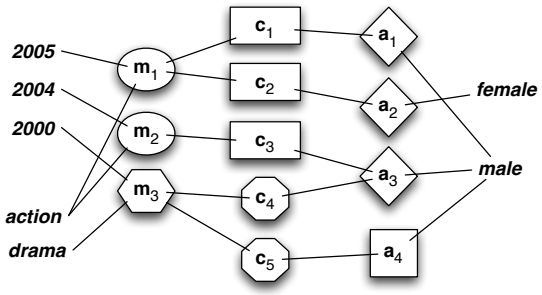
A TuG synopsis is a graph-based representation of a partitioning $T_{\mathcal{S}}$ along with information on partition sizes, the number of joining tuples between partitions, and value summaries that approximate the distribution of values in each partition. Formally, a TuG synopsis is defined as follows.

DEFINITION 3.1. *A* TuG *synopsis of a data graph G_D is a graph $\mathcal{TG}$ such that: (a) the nodes of $\mathcal{TG}$ correspond to the partitions of a partitioning $T_{\mathcal{S}}$, (b) each node r is labeled with the corresponding relation R and is associated with a counter $tcount(r) = |r|$, (c) for each attribute A of R, the node is associated with a value summary $vsum(r, A)$ that summarizes $values(r, A)$, (d) $\mathcal{TG}$ contains an edge (r, s) for each pair of joining partitions r and s, and (e) each edge is associated with a counter $jcount(r, s) = |r \bowtie s|$.*

The value summary $vsum(r, A)$ approximates the distribution $values(r, A)$ and it can be realized with well known approximation techniques such as histograms [Poosala et al. 1996], wavelets [Matias et al. 1998], or samples [Lipton et al. 1993]. To simplify exposition, we henceforth assume that $vsum(r, A)$ is realized using a single-dimensional histogram [Poosala et al. 1996]. We note that our choice of single-



(a)



(b)

Fig. 4. TuG summary (a) and the corresponding data graph (b). The enclosing shapes are used to denote the correspondence between summary nodes and tuples.

dimensional summaries has specific implications on the approximation of the joint value distribution across all the attributes of the tuples in r. We revisit this point later, when we discuss the properties of the TuG summarization model. We also note that the TuG summary does not record the extents of partitions but just their sizes. In effect, this implies that it is not possible to recover the data graph from a given TuG synopsis. As we discuss in Section 8, inferring this inverse mapping is an interesting problem that is relevant in the generation of set-valued answers.

Example 3.1.: Figure 4(a) shows an example TuG for the data graph of Figure 4(b). Consider nodes m_α and c_α. Node m_α represents two `Movie` tuples, and node c_α represents three `Cast` tuples. Their join is denoted by the linking edge and it includes $jcount(m_\alpha, c_\alpha) = 3$ results. Moreover, m_α is linked with value summaries $vsum(m_\alpha, \text{year})$ and $vsum(m_\alpha, \text{genre})$ that summarize the distribution of year and genre values respectively among the tuples corresponding to m_α. In this example, we assume that the value summaries devote a distinct bucket to each value and thus record the exact frequency distribution of each attribute. ∎

Overall, each summary node r serves as a representative for the tuples in its extent. The edges between r and other nodes summarize the join relationships to other relations, and the attached value summaries approximate the distribution of values in the attributes of the tuples. As will be shown later, the TuG framework

uses the recorded aggregate information in order to estimate the probability that a query expression is satisfied on the tuples of each node. As an example, consider again the TuG shown in Figure 4(a) and in particular the node $\mathtt{m}_\alpha$. Using the attached value summary and $tcount(\mathtt{m}_\alpha)$, it is possible to estimate the probability that a random tuple of the node satisfies a specific predicate on `year`. Similarly, using $jcount(\mathtt{m}_\alpha, \mathtt{c}_\alpha)$ and the counts $tcount(\mathtt{m}_\alpha)$ and $tcount(\mathtt{c}_\alpha)$, it is possible to derive the probability that a tuple in $\mathtt{m}_\alpha$ joins with a tuple in $\mathtt{c}_\alpha$ (in this case, $3/6 = 0.5$).

Clearly, this per-node approximation may introduce error, since all the tuples of the corresponding partition are represented with the same aggregate information. Moreover, the per-edge and per-attribute information for each partition r is decoupled from other join relationships or attributes. Returning to the example of Figure 4(a), note that the synopsis does not record explicitly the correlation between the year attribute of movies and the joins to cast. As we discuss in more detail later, this property implies a localized independence assumption for each node that may introduce additional error. It becomes obvious that the quality of approximation is inherently linked to the partitioning $T_{\mathcal{S}}$ that determines the graph structure of the synopsis. Deriving a good partitioning is the goal of the construction algorithm that we discuss in Section 5.

3.2 TuG-based Approximate Query Answering

Having described the TuG model, we now discuss the generation of approximate answers over concise TuG synopses. The problem can be formulated as follows: Given a TuG synopsis $\mathcal{TG}$ of a database D and a query $(Aggr, Q)$, compute an approximation to $Aggr(Q(D))$ using the information stored in $\mathcal{TG}$.

The proposed approximation framework is based on the notion of embedding Q in $\mathcal{TG}$, a process that can be seen as the analog of matching Q in the data graph (see Section 2). More concretely, an embedding e of Q in $\mathcal{TG}$ can be defined as a mapping from query nodes to synopsis nodes such that: (a) $e(q)$ corresponds to the same relation as q, (b) $e(q)$ contains tuples that satisfy the predicates attached to q, and (c) each edge (q, q') is mapped to a synopsis edge $(e(q), e(q'))$. It is straightforward to show that each matching of Q, and hence each result of the select-join expression, corresponds to a distinct embedding, since each tuple corresponds to a distinct synopsis node. In that sense, an embedding intuitively represents a disjoint subset of matchings from $Q(D)$. (Due to the lossy nature of the synopsis, however, there may be embeddings that do not represent any actual matchings.) In what follows, we use $Q(\mathcal{TG})$ to denote the set of embeddings for Q in $\mathcal{TG}$.

Given an embedding e in $Q(\mathcal{TG})$, we define $Aggr(e)$ as the approximation of the aggregate on the set of matches represented by e. Hence, we can derive an approximation of $Aggr(Q(D))$ by accumulating the per-embedding approximations $Aggr(e)$. As an example, an approximation of $COUNT(Q(D))$ can be derived as $\sum_{e \in Q(\mathcal{TG})} COUNT(e)$. It is possible to define similar expressions for all the aggregates of the query model.

Example 3.2.: Consider again the query $(Q, COUNT)$ shown in Figure 2. Given the synopsis of Figure 4(a), we can identify two embeddings e_1 and e_2 defined as follows: $e_1(q_1) = m_\beta$, $e_1(q_2) = c_\beta$, $e_1(q_3) = a_\alpha$, and $e_2(q_1) = m_\beta$, $e_2(q_2) = c_\beta$,

$e_2(q_3) = a_\beta$, respectively. The total count can be computed as $COUNT(Q(\mathcal{T}G)) = COUNT(e_1) + COUNT(e_2)$. ∎

Thus, the problem of approximate query answering over TuGs is reduced down to two sub-problems: (a) Approximating the result of $Aggr(e)$, and (b) Efficiently combining the individual estimates $COUNT(e)$ across all the embeddings in $Q(\mathcal{T}G)$. We discuss each of these sub-problems in the following sub-sections.

3.2.1 *Approximating Aggregates On Single Embeddings.* We start with the following problem: Given an embedding e of Q and an aggregate $Aggr$, approximate $Aggr(e)$, i.e., the value of the aggregate over the matches of Q represented by e. To simplify exposition, we focus initially on $COUNT(e)$ and present the extension to other aggregates later.

Recall that Q represents the join expression $\sigma_{C_1}(R_1) \bowtie \ldots \sigma_{C_K}(R_K)$, where each C_i is a (potentially empty) conjunction of range predicates on a subset of attributes of R_i. We use the notation $c \in C_i$ to denote a range predicate in C_i on a single attribute of R_i. Let r_i denote the summary node in embedding e corresponding to relation R_i, $1 \leq i < K$. The aggregate $COUNT(e)$ denotes the number of output tuples of Q if each relation is assumed to have just the tuples of the corresponding summary node. Hence, we can express $COUNT(e)$ as follows:

$$COUNT(e) = |r_1 \times \cdots \times r_K| \cdot Prob(\bigwedge_{1 \leq i \leq K} \sigma_{C_i}(r_i) \quad \wedge \bigwedge_{(R_i,R_j) \text{ in } Q} r_i \bowtie r_j)$$

The first factor represents the size of the cross product of the corresponding partitions, and it can be readily computed as $\prod_{1 \leq i \leq K} tcount(r_i)$. The second factor denotes the probability that a random tuple in the cross product satisfies the selection and join predicates of the query. More specifically, $\sigma_{C_i}(r_i)$ denotes the event that a tuple in r_i satisfies the predicates in C_i, and $r_i \bowtie r_j$ denotes the event that a pair of tuples in $r_i \times r_j$ join according to the join predicate (R_i, R_j) in Q.

Using the chain rule, we can rewrite the expression of $COUNT(e)$ as follows:

$$COUNT(e) = \prod_{1 \leq i \leq K} tcount(r_i) \cdot Prob(\bigwedge_{(R_i,R_j) \text{ in } Q} r_i \bowtie r_j) \cdot Prob(\bigwedge_{1 \leq i \leq K} \sigma_{C_i}(r_i) \mid \bigwedge_{(R_i,R_j) \text{ in } Q} r_i \bowtie r_j) \quad (1)$$

The first probability factor captures the distribution of join relationships across the partitions $r_1, \ldots, r_K$. The TuG synopsis, however, records a edge-specific join counts, and thus does not record information on the joint distribution of join relationships. Hence, to be able to utilize the TuG synopsis in order to approximate the first probability factor, we apply the following independence assumption:

Join Independence Assumption. Let (r, s) be a synopsis edge. The probability that a tuple in r joins with a tuple in s is independent from the join relationships of the tuples in r or s with other nodes.

This resembles the independence assumption that is applied by query optimizers (in the absence of statistics) in order to de-correlate the join of two relations with other joins in the same query. The difference is that the assumption is applied locally on the particular edge (r, s), and hence its validity depends on the tuples that appear in the extents of r and s, and in turn on the partitioning T_S that

defines the synopsis. As we discuss later, this provides the means by which we can limit the error of approximation through the partitioning. Using this assumption, it is possible to de-correlate the individual events $r_i \bowtie r_j$ and apply the following approximation:

$$Prob(\bigwedge_{(R_i,R_j) \text{ in } Q} r_i \bowtie r_j) \approx \prod_{(R_i,R_j) \text{ in } Q} Prob(r_i \bowtie r_j)$$

The second probability factor in Equation 1 captures the joint distribution of attribute values and its correlation to the distribution of join relationships. The $\mathcal{TG}$ model does not record this type of information, because the value summaries of each summary node are not linked to its join relationships, and there is a single-dimensional summary for each attribute. Hence, to be able to approximate the second probability factor using the information stored in $\mathcal{TG}$, we resort to the following independence assumption:

Value Independence Assumption. Let r be a summary node corresponding to relation R. Let A be an attribute of R. The distribution of values of A in the tuples of r is independent from: (a) the value distribution of other attributes in the tuples of r, and (b) the join relationships between the tuples in r and tuples of other partitions.

Similar to the case of join relationships, this resembles the independence assumption applied by optimizers in order to de-correlate the value distribution of different attributes within the same relation. The difference again is that the assumption is localized to the tuples of r, and hence its validity can be controlled through the partitioning $T_{\mathcal{S}}$ that defines the synopsis. The assumption enables the following approximation:

$$Prob(\bigwedge_{1 \le i \le K} \sigma_{C_i}(r_i) \mid \bigwedge_{R_i \bowtie R_j \text{ in } Q} r_i \bowtie r_j) \approx \prod_{1 \le i \le K} \prod_{c \in C_i} Prob(\sigma_c(r_i))$$

Using the two approximations, we can write the expression for $COUNT(e)$ as follows:

$$COUNT(e) = \prod_{1 \le i \le K} tcount(r_i) \cdot \prod_{R_i \bowtie R_j \text{ in } Q} Prob(r_i \bowtie r_j) \cdot \prod_{1 \le i \le K} \prod_{c \in C_i} Prob(\sigma_c(r_i))$$

At this point, the result of the expression can be computed based on the information stored in the synopsis. More concretely, each probability factor $Prob(\sigma_c(r_i))$ can be estimated directly from the corresponding value summary attached to r_i. This estimation depends on the actual histogram technique, but in general it involves a straightforward summation of frequencies across the histogram buckets that overlap with the range predicate c. Each factor $Prob(r_i \bowtie r_j)$ can be estimated using the stored edge- and node-counts. More specifically, there is a total of $jcount(r_i, r_j)$ join results out of $tcount(r_i) \cdot tcount(r_j)$ tuples in the cross product, and thus it follows that $Prob(r_i \bowtie r_j) = jcount(r_i, r_j)/(tcount(r_i) \cdot tcount(r_j))$.

EXAMPLE 3.3.: Consider again the embedding e_1 described in Example 3.2 that is defined as $e_1(q_1) = m_\beta$, $e_1(q_2) = c_\beta$, $e_1(q_3) = a_\alpha$. Using Equation 2, the estimated count can be computed as follows:

$$COUNT(e_1) = tcount(m_\beta)tcount(c_\beta)tcount(a_\alpha)Prob(m_\beta \bowtie c_\beta)Prob(c_\beta \bowtie a_\alpha)$$
$$Prob(\sigma_{year=2000}(m_\beta))Prob(\sigma_{genre=Drama}(m_\beta))Prob(\sigma_{sex=male}(a_\alpha))$$

Based on the information present in the synopsis, we can estimate the probability factors as follows:

$$\begin{aligned} Prob(m_\beta \bowtie c_\beta) &= \frac{jcount(m_\beta, c_\beta)}{tcount(m_\beta)tcount(c_\beta)} = 1 \\ Prob(c_\beta \bowtie a_\alpha) &= \frac{jcount(c_\beta, a_\alpha)}{tcount(c_\beta)tcount(a_\alpha)} = 1/6 \\ Prob(\sigma_{year=2000}(m_\beta)) &= 1/1 \\ Prob(\sigma_{genre=Drama}(m_\beta)) &= 1/1 \\ Prob(\sigma_{sex=male}(a_\alpha)) &= 2/3 \end{aligned}$$

The final estimate for $COUNT(e_1)$ is thus 2/3. ∎

We observe that it is possible to associate directly the join probability $Prob(r \bowtie s)$ with each synopsis edge instead of the join count $jcount(r, s)$. This optimization is straightforward to apply after the synopsis is constructed and it can speed up considerably the computation of estimates.

Clearly, the accuracy of the estimate is linked directly to the validity of the two independence assumptions that have to be applied. Such assumptions are typically dangerous in practice, as real-world data tend to exhibit correlations within and across the distribution of joins and values. The crucial point of the TUG framework, however, is that these assumptions are applied locally on each summary node r, and hence their validity can be controlled through the partitioning $T_\mathcal{S}$ that defines the synopsis. As an example, the partitioning that assigns a single tuple to each summary node will satisfy the previous assumptions and yield a zero-error synopsis. As the partitioning becomes coarser, the assumptions may become less valid which in turn increases the error of estimation. The goal of course is to determine a partitioning that achieves a good trade-off between the size of the synopsis and the validity of these assumptions. As a final note, we observe that the estimation error is also affected by the error of approximation within each individual summary $vsum(r, A)$. This error contribution is orthogonal to the independence assumptions, which means that the quality of the synopsis can be improved by the application of better histogramming techniques.

Clustering offers an alternative and intuitive method to interpret the TUG estimation framework. Recall that V_D represents the set of values that are present in the data graph. Let $k_\mathcal{S} = |T_\mathcal{S}| + |V_D|$. We define a conceptual $k_\mathcal{S}$-dimensional space where each of the first $|T_\mathcal{S}|$ dimensions corresponds to a distinct summary node, and each of the remaining $|V_D|$ dimensions corresponds to a distinct value. To simplify notation, we index the dimensions of this space using the summary nodes and the values in V_D. Given a node r, we define a corresponding point p_r in

this space as follows:

$$p_r[s] = jcount(r,s)/tcount(r), s \in T_{\mathcal{S}}$$
$$p_r[\nu] = \text{fraction of tuples in } r \text{ that have value } \nu, \nu \in V_D$$

The coordinate $p_r[\nu]$ can be viewed as the selectivity factor of a selection on the specific value, while $p_r[s]$ can be viewed as the average selectivity factor of the join $r \bowtie s$ per tuple in r. Essentially, p_r captures the aggregate join and value characteristics of the tuples corresponding to node r. In a similar fashion, we can define the point p_τ for tuple τ in r assuming that τ forms a partition of a single tuple. Hence, $p_\tau[s]$ is the number of results that are generated by joining τ and the tuples of s, and $p_\tau[\nu]$ is 1 if τ carries the specific value and 0 otherwise. Based on this formulation, we can show that the independence assumptions become trivially valid if $p_r = p_\tau$ for each tuple τ in r. To see this, observe that there is no meaningful correlation in the join and value distribution if all the tuples have the same join relationships and values. Conversely, the independence assumptions become less valid if node r groups tuples of diverse characteristics, which in turn implies that the distance between p_r and p_τ increases. Hence, if we consider r as a cluster of the corresponding tuple points p_τ, we can say that p_r represents the centroid of the cluster, and also that the tightness of the cluster reflects directly the validity of the independence assumptions on r. Later, we show how this interpretation can help us quantify the quality of approximation in a TuG synopsis.

Extension to other aggregates. We now discuss the extension of the previous technique to the other aggregates in our query model, namely *SUM*, *AVG*, *MIN*, and *MAX*. Let A be the attribute on which the aggregate is computed, and assume that R_a is the corresponding relation, $1 \leq a \leq k$. Let r_a be the summary node in the embedding corresponding to R_a. We say that a value ν is represented in r_a if the estimated frequency is positive based on $vsum(r_a, A)$ and the predicates of the query.

The computation of *MIN* and *MAX* can be performed in a straightforward fashion, by examining the value summary $vsum(r, A)$ and returning the minimum and maximum represented value respectively. For *SUM*, we first consider the case where attribute A is not referenced in any query predicate. The value independence assumption dictates that each represented value ν occurs with probability $Prob(\sigma_{A=\nu}(r))$ in the join tuples represented by the embedding. Similar to the case of $COUNT(e)$, this probability can be approximated using the estimated frequency of ν and $tcount(r_a)$. The sum aggregate can thus be computed as $SUM(e) = \sum_\nu \nu Prob(\sigma_{A=\nu}(r))COUNT(e)$, where the summation ranges over all represented values. In the case that A is referenced in a predicate, say c, then the probability factor $Prob(\sigma_{A=\nu}(r))$ in the previous expression is changed to $Prob(\sigma_{A=\nu \wedge c}(r))$. Finally, the estimate for *AVG* can be derived from the estimates on *SUM* and *COUNT*.

It is possible to extend our framework to cover the computation of aggregates over an expression that combines several attributes, e.g., $SUM(A_1 \cdot A_2)$ for attributes A_1 and A_2. Let r and r' denote the nodes in the embedding corresponding to attributes A_1 and A_2 respectively. Based on the value independence assumption, it is possible to combine the two histograms $vsum(r, A_1)$ and $vsum(r', A_2)$ to derive

the joint frequency distribution of the expression involving the two attributes. Past this point, the computation of aggregates proceeds as described in the previous paragraph. Of course, the feasibility of this scheme depends on the efficiency of computing the result of the expression on the cross product of the values represented in the relevant histograms. The cost may be low for simple expressions, like the one mentioned earlier, but it may rise sharply if the expression involves expensive user-defined functions.

3.2.2 *Combining Estimates Across All Embeddings.* Having described the computation of $Aggr(e)$ for a single embedding e, we switch our attention to the problem of combining the individual estimates across all embeddings in $Q(\mathcal{T}G)$.

A straightforward approach is to enumerate all the distinct embeddings in $Q(\mathcal{T}G)$, compute $Aggr(e)$ over each embedding, and then combine the estimates based on the semantics of $Aggr$. The enumeration essentially needs to identify sub-graphs of the synopsis graph that are isomorphic to the query graph, and can be performed by performing a depth-first traversal of the synopsis graph. We omit the details of the sub-graph matching algorithm as they are straightforward.

The drawback of the previous approach is that it may have to enumerate an exponential number of embeddings, thus increasing substantially the cost of approximate query answering. As we show next, it is possible to avoid this overhead if the query Q has an acyclic join graph. In particular, it is possible to compute $Aggr(Q(\mathcal{T}G))$ for an acyclic (i.e., tree-join) query Q without enumerating the distinct embeddings in $Q(\mathcal{T}G)$. This result is significant for the application of our techniques in practice, as tree join queries arise frequently in several application domains.

The idea behind the algorithm is to apply a dynamic-programming approach to estimation, taking advantage of the structure of the estimation expressions for tree-join queries. We discuss the algorithm for the case of $COUNT$, and then extend it to the other aggregates in our framework. We begin with a simple example that illustrates the basic idea behind the approach.

EXAMPLE 3.4.**:** We consider again the query of Example 3.2. For convenience, we repeat it here: $Q = \sigma_{year=2000 \wedge\ genre=Drama}(\texttt{movie}) \bowtie \texttt{cast} \bowtie \sigma_{sex=male}(\texttt{actor})$. As noted, the query has two embeddings e_1 and e_2 defined as follows: $e_1(q_1) = m_\beta$, $e_1(q_2) = c_\beta$, $e_1(q_3) = a_\alpha$, and $e_2(q_1) = m_\beta$, $e_2(q_2) = c_\beta$, $e_2(q_3) = a_\beta$, respectively.

The tree-join property of Q asserts that we can generate the answers by joining the movie tuples of $\sigma_{year=2000 \wedge\ genre=Drama}(\texttt{movie})$ to the cast tuples of $(\texttt{cast} \bowtie \sigma_{sex=male}(\texttt{actor}))$. This property can be carried over to the identification of embeddings and also to the estimation of $COUNT$. More concretely, consider the query Q' that involves only variables q_2 and q_3. We can identify two embeddings e' and e'' for Q': $e'(q_2) = c_\beta$, $e'(q_3) = a_\alpha$, and $e''(q_2) = c_\beta$, $e''(q_3) = a_\beta$, respectively. Then, the embeddings for Q, which "augments" Q' with variable q_1, can be formed by linking the synopsis nodes for q_1 and the embeddings for Q' through a synopsis edge that corresponds to $q_1 \bowtie q_2$. In this case, there is one such node, namely m_β, and it is possible to connect it to the embeddings e' and e'' through the synopsis edge (m_β, c_β).

This composition process also carries in the computation of $COUNT(e_1) + COUNT(e_2)$. By substituting the count of each embedding with Equation 2 and by

performing standard algebraic manipulations, we arrive at the following expression:

$$\begin{aligned} & COUNT(e_1) + COUNT(e_2) = \\ & tcount(m_\beta) Prob(\sigma_{year=2000}(m_\beta)) Prob(\sigma_{genre=Drama}(m_\beta)) \\ & Prob(m_\beta \bowtie c_\beta)(tcount(c_\beta)(Prob(c_\beta \bowtie a_\alpha) tcount(a_\alpha) Prob(\sigma_{sex=male}(a_\alpha)) \\ & + Prob(c_\beta \bowtie a_\beta) tcount(a_\beta) Prob(\sigma_{sex=male}(a_\beta))) \\ = & \; tcount(m_\beta) Prob(\sigma_{year=2000}(m_\beta)) Prob(\sigma_{genre=Drama}(m_\beta)) Prob(m_\beta \bowtie c_\beta) \\ & (COUNT(e') + COUNT(e'')) \end{aligned}$$

Hence, the count of embeddings of Q that map q_1 to m_β can be computed based on the counts of embeddings of Q' that map the joining variable q_2 to neighbors of m_β. ∎

The previous discussion suggests a dynamic-programming approach to the computation of the total count, by computing the aggregate on the embeddings of sub-queries and then "stitching" together the sub-estimates. More concretely, let $q_{k_1}, \ldots, q_{k_K}$ denote an enumeration of the query variables based on a depth-first traversal of the query graph starting from variable $q_1 = q_{k_1}$. We note that the same traversal defines a spanning tree of the query graph that includes all the query edges. (This stems from the acyclicity of the query graph.) We use $Q[i]$ to denote the sub-tree of the spanning tree that is rooted at q_{k_i}. Note that $Q[i]$ corresponds to a sub-query of Q, and also that Q is equivalent to $Q[1]$. Let $count(r, i)$ denote the total count of embeddings for $Q[i]$ that map q_{k_i} to r. Hence, the desired $COUNT(Q(\mathcal{T}G))$ aggregate can be computed as the sum of $count(r, 1)$ for all synopsis nodes r. Assuming that q_{k_i} is a leaf in the spanning tree, i.e., $Q[i]$ represents a single-relation query with only selection predicates, the expression for $count(r, i)$ is readily computed as follows:

$$count(r, i) = \prod_{c \text{ in } C_{k_i}} Prob(\sigma_c(r)) \cdot tcount(r)$$

Note that the expression involves C_{k_i} as the conjunction of selection predicates that is attached to variable q_{k_i}. Now, assume that q_{k_i} is not a leaf of the spanning tree, and define $J(i) = \{j | (q_{k_i}, q_{k_j})$ is a query edge and $i < j\}$ as the indices of the children of q_{k_i} in the spanning tree. Following the idea of the previous example, $count(r, i)$ can be computed by combining r with the embeddings of $Q[j]$, $j \in J(i)$, that are rooted at neighbors of r. More concretely, the following result can be shown:

$$count(r, i) = \prod_{c \text{ in } C_{k_i}} Prob(\sigma_c(r)) \cdot tcount(r) \cdot \left(\prod_{j \in J(i)} \sum_{(r,s) \text{ in } \mathcal{T}G} Prob(r \bowtie s) \cdot count(s, j) \right)$$

This recurrence expression forms the basis for the estimation algorithm shown in Figure 5. The algorithm performs a nested iteration over the query variables and the synopsis nodes, computing $count(r, i)$ based on the previous expressions. The key point is that the embeddings in $Q(\mathcal{T}G)$ are not enumerated and thus we expect the algorithm to perform much more efficiently compared to the naive approach.

Procedure TugComputeCount(Q, $\mathcal{TG}$)
Input: Query Q corresponding to the expression $\sigma_{C_1}(R_1) \bowtie \cdots \bowtie \sigma_{C_K}(R_K)$; Synopsis $\mathcal{TG}$.
Output: $COUNT(Q(\mathcal{TG}))$
begin
1.Initialize a hash table $count : T_{\mathcal{S}} \times \{1, \ldots, K\} \rightarrow Real$.
2.Let $q_{k_1}, \ldots, q_{k_K}$ be an enumeration of the query variables based on a breadth-first traversal
3.$count(r, i) \leftarrow 0, r \in T_{\mathcal{S}}, 1 \leq i \leq K$
4.**for** $i = K$ down to 1 **do**
5. **for each** r in $T_{\mathcal{S}}$ such that r matches the relation name and value conditions of q_{k_i} **do**
6. Let $J(i) = \{j | (q_{k_i}, q_{k_j})$ is a query edge and $i < j\}$
7. $count(r, i) \leftarrow \left(\prod_{c \text{ in } C_{k_i}} Prob(\sigma_c(r)) \right) \cdot tcount(r)$
8. **if** $J(i) \neq \emptyset$ **then** $count(r, i) \leftarrow count(r, i) \cdot \left(\prod_{j \in J(i)} \sum_{(r,s) \text{ in } \mathcal{TG}} Prob(r \bowtie s) \cdot count(s, j) \right)$
9. **done**
10.**done**
11.**return** $\sum_{r \text{ in } \mathcal{TG}} count(r, 1)$
end

Fig. 5. Algorithm TugComputeCount.

Indeed, it is straightforward to verify that the time complexity is $\mathcal{O}(|T_{\mathcal{S}}|^2 K)$, which compares favorably to the $\mathcal{O}(|T_{\mathcal{S}}|^K)$ complexity of the naive approach.

The computation of other aggregates can be performed in a similar fashion. In all cases, we set q_1 to the variable corresponding to the aggregated attribute and then invoke TugComputeCount with a modified return expression. For SUM, the return expression is $\sum_{r:count(r,1)>0} \sum_{\nu} \nu Prob(\sigma_{A=\nu}(r)) count(r, 1)$, where r ranges over all synopsis nodes that map to q_1 in at least one embedding, and ν ranges over the represented values of r. We note that the probability factor may have to be adjusted if A is constrained by a predicate, as discussed in Section 3.2.1. For MIN, the return expression is $min\{\nu \mid \nu$ is represented in $r \wedge \ count(r, 1) > 0\}$[2]. The expression for MAX is similar. Finally, AVG can be estimated through SUM and $COUNT$.

4. COMPRESSING TUG SYNOPSES

Having described the TuG model and the computation of approximate answers, we now shift our attention to the important problem of constructing accurate TuG synopses. We begin in this section with the introduction of a basic operation that compresses the size of a TuG synopsis. This operation forms the basis for the more general TuGBuild construction algorithm that we discuss in Section 5.

4.1 Node-Merge Operation

Our basic compression operation, termed `merge`, reduces the storage of a TuG by collapsing several synopsis nodes in a single new node. More formally, let $\mathcal{M}$ be a set of summary nodes that correspond to the same relation R. The opera-

[2]This expression can be easily modified so that a node r is considered only if it is estimated to appear in at least one embedding, i.e., $count(r, 1) > 1$.

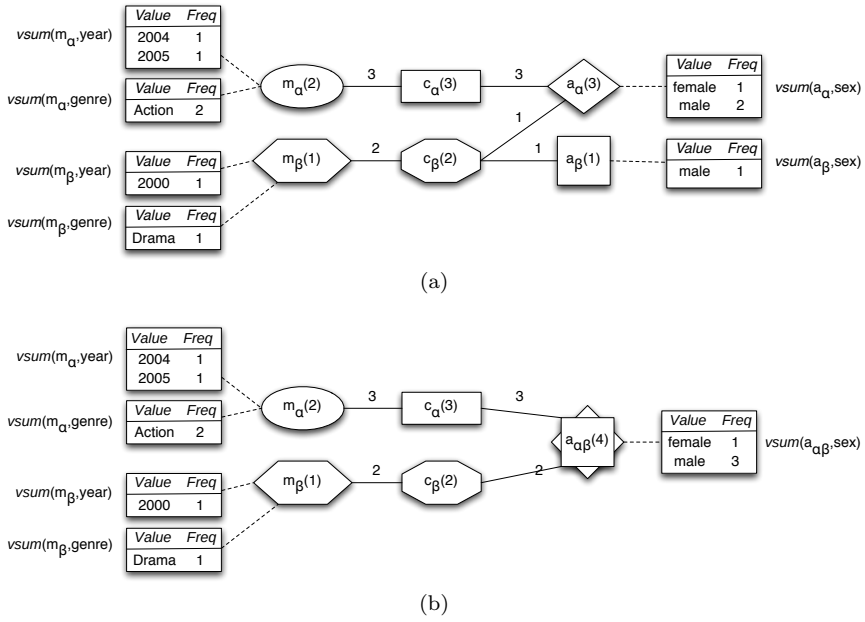


Fig. 6. Example node-merge operation: (a) Starting TuG summary $\mathcal{TG}$, (b) Resulting summary $\mathcal{TG}'$ after the application of $\texttt{merge}(\{a_\alpha, a_\beta\})$

.

tion $\texttt{merge}(\mathcal{M})$ substitutes the nodes in $\mathcal{M}$ with a single new node $\tilde{r}$ that represents the union of the individual partitions. Hence, the count statistics of the new node are computed as follows: $tcount(\tilde{r}) = \sum_{r\in\mathcal{M}} tcount(r)$, and $jcount(\tilde{r}, s) = \sum_{r\in\mathcal{M}} jcount(r, s)$ for every neighbor s of the nodes in $\mathcal{M}$. Moreover, for each attribute A of R, the value summary $vsum(\tilde{r}, A)$ now summarizes the value distribution $values(\tilde{r}, A)$ that results from the union of the individual distributions $values(r, A)$ for $r \in \mathcal{M}$. Figure 6 illustrates the result of a merge operation on the synopsis of our running example.

An important issue is the effect of this localized compression on approximate query answering. In particular, we can ask the following question: How different are the approximate answers that are generated by the synopsis after the application of $\texttt{merge}(\mathcal{M})$? This question is important in the context of synopsis construction, as we can start with a large synopsis that is very accurate, and subsequently compress its size through merge operations of small difference. One possible way to answer the question is to compare the join and value statistics of the new node $\tilde{r}$ with the join and value statistics of the merged nodes in $\mathcal{M}$. This approach is intuitive, since the generation of approximate answers employs these statistics and $\tilde{r}$ is meant to substitute every node in $\mathcal{M}$. Hence, if the statistics are similar, then we can expect a small difference in the approximate answers before and after the merge.

To quantify the difference in statistics, we resort again to the clustering model that was described in Section 3. More concretely, recall that it is possible to map each synopsis node r to a $k_{\mathcal{S}}$-dimensional point p_r that records the average join and value characteristics of the tuples in r. The intuition is that p_r defines a centroid

that represents all the tuples of r. Similarly, we define the centroid $p_{\tilde{r}}$, and note that it can be seen as the representative of the tuples in $\cup_{r \in \mathcal{M}} r$. If $p_{\tilde{r}}$ is similar to p_r for each node $r \in \mathcal{M}$, this implies that the new node has similar join and value statistics to the merged nodes, and hence the merge operation is likely to have a small effect on the quality of approximate query answering. Another interpretation is that the nodes in $\mathcal{M}$ introduce redundancy in the synopsis if their centroids are similar, and hence it is advantageous to collapse them in a single node.

Based on the previous discussion, we can quantify the effect of $\texttt{merge}(\mathcal{M})$ by computing a metric of distance between $p_{\tilde{r}}$ and the points $\{p_r \mid r \in \mathcal{M}\}$. There are several well known metrics for this task from the clustering literature, such as, the diameter of the group, the radius, or the maximum inter-point distance to name a few. In our work, we employ the radius metric which is defined as follows:

$$radius(\mathcal{M}) = \frac{\sum_{r \in \mathcal{M}} ||p_r - p_{\tilde{r}}||_2}{|\mathcal{M}|}$$

Here, $||x||_2$ denotes the L-2 norm of vector x. The radius metric measures the average distance of points from the center $p_{\tilde{r}}$. We have chosen this metric as it is common in the clustering literature, and there are several clustering algorithms that are already optimized for its use. We note, however, that it does not form a core component of our technique, and it can be substituted with other metrics.

The multi-dimensional space of join and value counts involves a large number of dimensions, given that it includes a separate coordinate for each value ν in the database. This property makes the computation of *radius* look impractical. We observe, however, that the points are likely to be sparse, given that $p_{\tilde{r}}$ will have non-zero coordinates only for the values corresponding to the attributes of R. Moreover, as we explain in the next section, the construction algorithm pre-compresses the value domains to a smaller set of values, thus making the computation of *radius* viable.

4.2 Lossless Merge Operations

In this section, we examine *lossless merge operations*, i.e., operations that do not affect the accuracy of the resulting synopsis. Such operations are crucial for the efficiency of the TuG construction algorithm, as they enable the fast compression of a large accurate synopsis to a much smaller equivalent summary.

Intuitively, we expect a merge operation to be lossless if $radius(\mathcal{M}) = 0$, i.e., nodes in $\mathcal{M}$ have the exact same join and value counts to every other node in the synopsis. In our work, we prove a somewhat surprising result: $\texttt{merge}(\mathcal{M})$ can be lossless even if the nodes in $\mathcal{M}$ do not match on every dimension of the underlying point space. This type of restricted similarity is captured by our novel concept of *all-but-one similarity* that we define next.

Let r and r' be two synopsis nodes for relation R. Let S be another relation that has a join relationship to R. We say that r and r' are similar with respect to S if $p_r[s] = p_{r'}[s]$ for each synopsis node s of S. Essentially, the tuples in r and r' have the same average number of join results when joined with the nodes of S, and in that sense they cannot be distinguished based solely on this join relationship. Similarly, we say that r and r' are similar with respect to an attribute A of R if

$p_r[\nu] = p_{r'}[\nu]$ for each value ν in the domain of A. We define all-but-one similarity by extending this idea to specific subsets of the relations that join with R and its attributes.

DEFINITION 4.1. *Nodes r and r' are called* all-but-one similar *if either they are similar with respect to all the attributes of R and all the joining relations except either a single attribute or a single relation.*

Hence, r and r' are all-but-one similar either if they only have different average join counts to the nodes of a single relation, or if they only have a different value distribution for a single attribute of R.

EXAMPLE 4.1.: Figure 7(a) shows an example synopsis that contains all-but-one similar nodes. More concretely, nodes a_2 and a_6 are all-but-one similar since their points differ only for the nodes of B. At the same time, a_2 and a_1 are also all-but-one similar, as their points differ only for the nodes of C. ∎

As hinted earlier, all-but-one similarity captures the redundancy of statistical information in a TuG synopsis. We formalize this property as follows:

THEOREM 4.1. *Consider a* TuG $\mathcal{TG}$ *and a set $\mathcal{M}$ of all-but-one similar nodes. Let $\mathcal{TG}$' be the summary that results from* `merge`$(\mathcal{M})$. *It holds that $Aggr(Q(\mathcal{TG})) = Aggr(Q(\mathcal{TG}'))$ for any query $(Aggr, Q)$.*

To prove the theorem, we show that the embeddings of a query Q over $\mathcal{TG}$ can be mapped to a set of equivalent embeddings over $\mathcal{TG}$'. The details of the proof can be found in the appendix.

The intuition behind Theorem 4.1 can be illustrated with a simple example. Consider a relation R that has a single attribute A and joins to a single relation S. Let r and r' be two synopsis nodes of R that have the same joins to S but different value distributions for A. By definition, r and r' are all-but-one similar, and hence a merge of the two nodes leads to a synopsis with the same accuracy. We observe that the different value distributions for r and r' cannot be correlated to the join relationships of the two nodes, since the joins to S are identical. Hence, the operation `merge`$(\{r, r'\})$ will not result in the loss of a join-to-value correlation that was captured by the separate nodes r and r'.

EXAMPLE 4.2.: Figure 7 shows the application of Theorem 4.1 on a sample synopsis. Looking at Figure 7(a) again, we observe that nodes a_2 and a_6 are all-but-one similar, as they differ only in their joins to B. A similar observation holds for nodes a_4 and a_5. We can thus perform two merge operations, as shown in Figure 7(b), without compromising the accuracy of the summary. After these operations, no nodes are all-but-one similar.

Similarly, we observe that nodes a_1 and a_2 of the original summary $\mathcal{TG}$ are all-but-one similar because they differ only in their joins to C, and the same holds for a_3 and a_4, and for a_5 and a_6. By performing the three corresponding merges, we obtain the first summary of Figure 7(c). At this point, the merged nodes a_{12} and a_{34} are identified as all-but-one similar, because they differ only in their joins to B. It is possible, therefore, to perform another lossless `merge` and derive the second synopsis of Figure 7(c). ∎

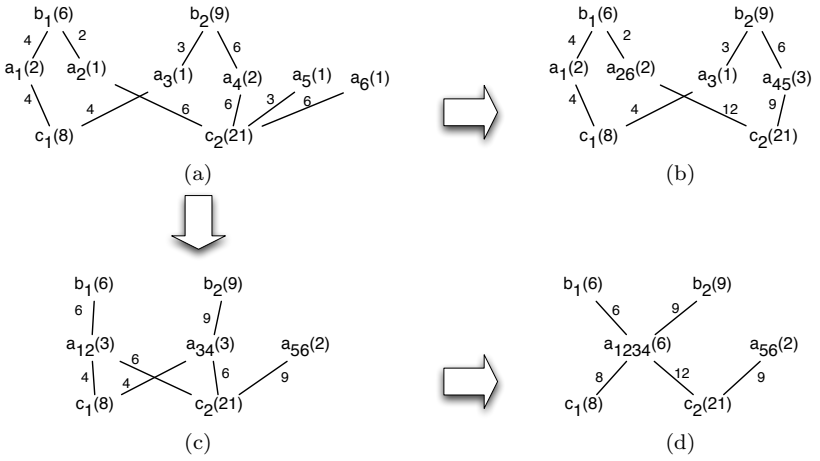


Fig. 7. An example of applying all-but-one similar merges: (a) Starting summary $\mathcal{TG}$, (b) $\mathcal{TG}$ after merging A with respect to C (c) $\mathcal{TG}$ after merging A with respect to B, and (d) then with respect to C.

The previous example highlights an interesting property of all-but-one similarity, namely, that the order in which lossless merge operations are applied can affect the resulting synopsis. We formalize this as follows. We say that a synopsis $\mathcal{TG}'$ is the *all-but-one minimal* synopsis for a synopsis $\mathcal{TG}$ if $\mathcal{TG}'$ is derived from $\mathcal{TG}$ through a sequence of lossless merge operations, and $\mathcal{TG}'$ does not contain any all-but-one similar nodes.

THEOREM 4.2. *The all-but-one minimal synopsis is not unique.*

The proof is straightforward and uses the counter-example of Figure 7. The intuition of the theorem is that merge operations affect the implied multi-dimensional space of join ratios and thus impact the existence of all-but-one similarity. This is an interesting property of our framework and at the same time a significant challenge, as it becomes necessary to select carefully the order in which such operations are applied. We revisit this issue in Section 5.1, where we introduce a heuristic for determining the order in which lossless merge operations are performed.

All-but-one similarity is a novel feature of our model that provides a unified (in terms of joins and values) characterization of statistical redundancy in a TuG synopsis. It is interesting to note that XML summarization models have also employed similar definitions of similarity, albeit with stricter conditions. In TreeSketches [Polyzotis et al. 2004], for instance, similarity is defined in terms of *all* the neighbors of a node in the *data graph*, which is considerably more restrictive compared to the *except-one* subset of *schema neighbors* used here. As we show in our experimental study, the greater flexibility of all-but-one similarity achieves a more aggressive compression of statistical information, and is key in the scalable construction of accurate TuG summaries. Transferring our results to the XML domain is an interesting direction for future work.

5. TUG CONSTRUCTION

In this section, we introduce the TuGBuild algorithm for constructing an accurate TuG synopsis within a specific storage budget. The proposed algorithm operates in four stages. In the first stage, it performs a pre-compression of numerical attributes in tight ranges in order to reduce the complexity of the value domain and "smooth" the value distribution. The second stage initializes a synopsis with a fine-grained partitioning (one tuple per partition) and exact value summaries, and proceeds to apply merge operations on all-but-one similar nodes (Section 4.2). The end result is the generation of a coarser partitioning that maintains the accuracy of the original synopsis. The third stage compresses the partitioning further by employing `merge` operations of low radius, so that the compression in size does not sacrifice the accuracy of the synopsis. The final stage substitutes the detailed value summaries with compressed single-dimensional histograms. Throughout this process, TuGBuild relies on efficient disk-based structures and scalable algorithms in order to handle large data sets under limited memory resources. Our experimental results (Section 6) verify that our techniques scale well in practice and are hence of interest for other graph-based summarization methods, e.g., in the XML domain.

The following sections describe the construction algorithm in more detail. We first examine the key problem of identifying effective `merge` operations, and then discuss the specifics of the TuGBuild algorithm.

5.1 Identifying Merge Operations

The TuGBuild algorithm relies on merge operations of low radius in order to construct a concise accurate synopsis. An important problem therefore is identifying such operations efficiently, in order to ensure good performance of the construction process.

Clearly, it is possible to identify good merge operations by computing a clustering of the synopsis nodes in the underlying space of join and value counts. Given that the clustering is computed over a metric space, we can use existing techniques that have very good scalability properties, e.g., BIRCH [Zhang et al. 1996] or subspace clustering [Agrawal et al. 1998]. One complicating factor, however, is that merge operations change the underlying space and may thus affect the validity of the identified clusters. For instance, assume that the clustering reveals a cluster $\mathcal{M}_1$ of R nodes and another cluster $\mathcal{M}_2$ of S nodes, such that the nodes in $\mathcal{M}_1$ join with the nodes in $\mathcal{M}_2$. The operation $\texttt{merge}(\mathcal{M}_1)$ will change the multi-dimensional space (since the coordinates of the nodes in $\mathcal{M}_1$ will be substituted with the single coordinate of the merged node), which changes in effect the value of $radius(\mathcal{M}_2)$. Moreover, the new multi-dimensional space may render other nodes similar and hence reveal more interesting clusters. On the other hand, assume that $\mathcal{M}_3$ is another cluster of nodes from relation R. Since the schema graph does not contain self-joins, it is trivial to show that $radius(\mathcal{M}_3)$ remains unaffected by $\texttt{merge}(\mathcal{M}_1)$. In other words, the radius values of the identified clusters remain valid if merge operations are confined to relations that are not neighbors in the schema graph. Based on this observation, our first design choice is to perform the merge operations in rounds, where in each round we focus on the nodes of a single relation.

The next step is to define a policy for selecting the relation of the current round.

Intuitively, it is beneficial to cluster on relation R if it will yield few clusters that are tight, i.e., it will offer a good compression to error ratio. We quantify the latter with a heuristic metric termed the clustering ratio of R and denoted as $CV(R, \mathcal{T}G)$. The proposed metric adopts the idea of subspace clustering [Agrawal et al. 1998], where clusters in low dimensional spaces are used to compute clusters in higher dimensions, and it examines the clustering potential of the nodes of R with respect to every attribute and every joining relation. More concretely, let A be an attribute of R and let ν be a value in the corresponding value domain. (In the context of the construction algorithm, ν is a value in the compressed domain of A if the latter is a numerical attribute.) Let $\mathcal{M}_\nu = \{r \mid p_r[\nu] > 0\}$ be the set of synopsis nodes for which the ν coordinate is positive, and let $\tilde{r}$ be the node that results from $\texttt{merge}(\mathcal{M}_\nu)$. We define the clustering ratio of R with respect to ν, denoted as $CV(R, \mathcal{T}G, \nu)$, as follows:

$$CV(R, \mathcal{T}G, \nu) = \frac{|\mathcal{M}_\nu| - 1}{\sum_{r \in \mathcal{M}_\nu} (p_r[\nu] - p_{\tilde{r}}[\nu])^2}$$

The numerator is indicative of the compression that is achieved by substituting the nodes in $\mathcal{M}_\nu$ with a single node, while the denominator is indicative of the error in the value counts for the coordinate corresponding to ν. (Note that this error is directly linked to the radius metric defined in Section 4). Essentially, the metric captures the amount of redundancy in the nodes of R with respect to the particular value ν. We define a similar metric $CV(R, \mathcal{T}G, s)$ with respect to a node s of a joining relation S, and compute $CV(R, \mathcal{T}G)$ as the average across all these per-value and per-node metrics. Intuitively, we expect a high $CV(R, \mathcal{T}G)$ value if the nodes of R can be separated in groups of similar join and value counts, implying that merge operations on R are likely to compress the synopsis effectively. The clustering policy then chooses the relation R that has the highest clustering ratio for the current round.

Clearly, $CV(R, \mathcal{T}G)$ is only a heuristic metric that may overestimate the potential of R. The metric, however, is used to guide the selection of relations for clustering and not to compute the actual clusters. We also observe that the metric depends on the current synopsis $\mathcal{T}G$, as the points p_r and $p_{\tilde{r}}$ that are used in the computation are defined over the current partitioning. This implies that the metric has to be recomputed after each round.

5.2 Construction Algorithm

We now proceed to describe the specifics of the TuGBuild construction algorithm, shown in Figure 8. TuGBuild receives as input the relational database D and a storage budget for the summary, and returns a TuG synopsis $\mathcal{T}G$. The storage budget is specified as two components: a budget B_S for the graph structure of the synopsis, and a budget B_V for the value summaries. Note that, ideally, the construction algorithm should be provided with a single storage budget and then select automatically the appropriate ratio between structure and values. The two budgets are related, however, through an intricate trade-off, which complicates the automatic detection of the right rationing. More concretely, a small storage budget implies few summary nodes, and hence few value summaries that share a larger

Procedure TuGBuild(D, B_S, B_V)
Input: Database D; join budget B_S; value budget B_v
Output: TuG $\mathcal{TG}$
Parameters: n_{PV}: number of compressed value ranges per attribute;

begin
1. Compress values of each numerical attribute A in at most n_{PV} ranges
2. Initialize $\mathcal{TG}$ with one node per tuple and with the compressed value ranges
3. **for** each relation R **do**
4. Compute the clustering ratio $CV(\mathcal{TG}, R, S)$ for each joining relation S
5. Compute the clustering ratio $CV(\mathcal{TG}, R, A)$ for each attribute A
6. **done**
7. TugCompressLossless($\mathcal{TG}$)
8. TugCompressJoinGraph($\mathcal{TG}, B_S$)
9. TugCompressValueGraph($\mathcal{TG}, B_V$)
10. **return** $\mathcal{TG}$

end

Fig. 8. Algorithm TuGBuild

value budget. Hence, the error of the value approximation is likely to be small and the error of structural approximation high. By increasing the structure budget, the trade-off moves in the opposite direction. We choose the simpler approach of dividing the storage budget a-priori in order to focus on the core problems of TuG construction. Developing a construction algorithm that automatically rations a single budget is an interesting direction for future work on TuG synopses.

As mentioned earlier, the construction algorithm operates in four stages: The first stage (line 1) compresses the value domains of numerical attributes in tight ranges; The second stage (lines 2–7) initializes a detailed synopsis and compresses it with lossless merges; The third stage (line 8) further compresses the synopsis with lossy merge operations to meet the structure budget B_S; The final stage (line 9) compresses the value summaries within the value budget B_V. The following sections describe each stage in more detail.

5.2.1 *Pre-Compression of Numerical Attributes.* The first stage transforms each numerical value domain to a small collection of tight numerical ranges[3]. The intuition is that the values in each range are close in the underlying domain and can thus be treated as equivalent in the summarization process. Our empirical results with this method have shown that the pre-compression introduces a small error in the approximation of values, but in return it increases the similarity of tuples with respect to their value distribution. Moreover, we have observed that tuples already exhibit similarities in their join relationships, and hence the increased value similarity leads to a large number of all-but-one similar nodes. By merging these nodes in the subsequent stage, the algorithm is able to reduce substantially the size of the synopsis while maintaining a low error of approximation.

The maximum number of ranges is controlled by the parameter n_{PV} of the con-

[3]Categorical attributes are not compressed using this method, as the notion of value proximity is not meaningful for categorical domains.

Procedure TugCompressLossless($\mathcal{TG}$)
Input: An TuG synopsis $\mathcal{TG}$
Output: A compressed version of $\mathcal{TG}$ after merging all-but-one similar nodes
begin
1. Initialize a hash table *Clusters*
2. $Clustered \leftarrow \emptyset$
3. **while** there is change or *Clustered* does not contain all relations **do**
4. Compute $CV(\mathcal{TG}, R, \mathcal{N})$ for each relation R and except-one subset $\mathcal{N}$ of $neighbors(R)$
5. $(R_m, \mathcal{N}_m) \leftarrow argmax_{R,\mathcal{N}}\{CV(\mathcal{TG}, R, \mathcal{N})\}$
6. **for each** node r of R_m **do**
7. Compute p_r taking into account only the coordinates corresponding to $\mathcal{N}_m$
8. Add r to $Clusters[p_r]$
9. **done**
10. TugMergeClusters($\mathcal{TG}, R_m, \mathcal{N}_m, Clusters$)
11. TugUpdateCV($\mathcal{TG}, R_m$)
12. $Clustered \leftarrow Clustered \cup \{R_m\}$
13. **done**
14. **end**

Fig. 9. Algorithm TugCompressLossless.

struction algorithm. The ranges are determined based on the max-diff heuristic on the area source parameter [Poosala et al. 1996]. The heuristic places values in the same range if they have similar frequency and they are spaced uniformly within the range. The end result is essentially a very detailed histogram that approximates well both the frequency and the spread of values.

5.2.2 *Building a Reference* TuG. The second stage generates a reference TuG summary that is much smaller than the original data D, and at the same time preserves the key statistical correlations between and across the distribution of joins and values. This is a key step in making the construction algorithm scalable, as it reduces the size of the input for the subsequent stages of the build process.

The reference TuG is initialized with a fine-grained partitioning of one tuple per synopsis node and with value summaries that record the exact frequency of values in each node. (The values of numerical attributes are the ranges computed in the first stage of the algorithm.) After completing the initialization steps, TuGBuild invokes algorithm TugCompressLossless to identify and merge all-but-one similar nodes. The pseudo-code for this algorithm is shown in Figure 9. TugCompressLossless works in iterations, where in each iteration it merges all-but-one similar nodes for one specific relation. The relation is picked based on its clustering ratio CV, except that the metric is computed on except-one subsets of joining relations and attributes. More concretely, the algorithm considers all possible pairs $(R, \mathcal{N})$, where R is a relation and $\mathcal{N}$ is a set of schema neighbors of R that excludes either one joining relation or one attribute. The clustering ratio is computed taking into account only the neighbors in $\mathcal{N}$, and the algorithm selects the pair $(R_m, \mathcal{N}_m)$ with the highest metric [4]. Essentially, the top pair $(R_m, \mathcal{N}_m)$

[4]The clustering ratio for a set $\mathcal{N}$ can be computed efficiently if $CV(\mathcal{TG}, R)$ is stored as per-relation

Procedure TugCompressJoinGraph($\mathcal{TG}, B_S$)
Input: An TuG synopsis $\mathcal{TG}$; a join budget B_S.
1. $ct_J \leftarrow ct_J^0$
2. Initialize a set *Clusters* of clusters
3. Mark all relations in the schema as *available*
4. **while** (join space of $\mathcal{TG}$) $> B_S$ **do**
5. **while** not all relations are examined OR no change **do**
6. $R_m \leftarrow \underset{available\ R}{\text{argmax}}\ (CV(\mathcal{TG}, R))$
7. Compute a clustering of R_m in *Clusters* so that each cluster has radius at most ct_J
8. TugMergeClusters($\mathcal{TG}, R_m, neighbors(R_m)$, *Clusters*)
9. TugUpdateCV($\mathcal{TG}$)
10. **if** percent of merged nodes $>$ *MERGE_THRESHOLD* **then**
11. Mark neighbors of R_m as *available*
12. **else**
13. Mark R_m as *unavailable*
14. **end if**
15. **if** nodes were merged **then**
16. TugCompressLossless($\mathcal{TG}$)
17. **end if**
18. **done**
19. $ct_J \leftarrow \alpha_J \cdot ct_J$ /** Increase clustering threshold **/
20. **done**

Fig. 10. Algorithm TugCompressJoinGraph.

specifies the relation that is likely to have a high number of all-but-one similar nodes with respect to the specific subset of neighbors. To identify the all-but-one similar nodes, TugCompressLossless examines each node r in R_m (line 17) and computes the point p_r taking into account only the dimensions that correspond to $\mathcal{N}_m$. The point is used to assign r to a cluster of nodes that have the same coordinates and that, by definition, are all-but-one similar with respect to $\mathcal{N}_m$. We note that the clusters hash table *Clusters* is expected to have a manageable memory footprint, since it only stores node ids. In the case that it grows too large, TugCompressLossless spills it to disk and consolidates the spilled partitions at the end of the scan (a la partitioned hash join).

Once the clusters have been identified, TugCompressLossless performs a merge operation for each cluster (algorithm TugMergeClusters) and recomputes the CV metrics for R_m and its neighbors (algorithm TugUpdateCV). (We discuss these algorithms in more detail in Section 5.3.) TugCompressLossless repeats the loop until every relation has been examined at least once and it is not possible to identify new clusters. Note that TugCompressLossless may select the same pair $(R, \mathcal{N})$ multiple times, as merges on the neighbors of R may increase the overlap of R nodes in terms of their joining nodes and thus render them all-but-one similar again.

sums.

5.2.3 *Compressing Join Information.* The third stage of the construction algorithm is realized by algorithm TugCompressJoinGraph, shown in Figure 10. The goal of the algorithm is to compress the structure of the synopsis graph down to B_S space units while limiting the error that is introduced. An important issue, therefore, is to select merge operations that offer a good trade-off between compression and error.

The selection of merge operations in TugCompressJoinGraph proceeds in two nested loops. The outer loop (lines 14–31) controls a *clustering threshold* ct_J that determines the maximum radius metric of the selected merge operations (Section 4.1). Recall that the radius metric is directly correlated with the error resulting from the operation and is thus a good indicator for the effect of the merge on synopsis accuracy. The idea is to start with a low threshold for the initial merges, and to gradually increase it, if necessary, in order to compress the join graph further. Given a specific clustering threshold, the inner loop (lines 15–29) repeatedly performs merge operations of the specified threshold until no such merges can be identified. The clustering process in the inner loop follows the methodology that we outlined in Section 5.1. More concretely, the algorithm selects the relation that maximizes the CV metric, and computes a clustering of its nodes such that each cluster has radius less than ct_J. Subsequently, TugMergeClusters is invoked in order to perform the corresponding merge operations. To identify clusters for the specific value of threshold ct_J, TugCompressJoinGraph employs a modified version of the well known BIRCH algorithm [Zhang et al. 1996]. Typically, BIRCH generates an effective clustering in two passes over the data, and thus enables our algorithm to scale to large data sets.

At this point, it is interesting to examine how the compression strategy relates to the assumptions of the estimation framework. Recall that the second stage initializes the synopsis using a very fine-grained partitioning of one tuple per synopsis node. Clearly, this starting synopsis satisfies trivially the assumptions of the estimation framework and is thus highly accurate. The second stage then proceeds to compress the synopsis with lossless merges which essentially preserve the same assumptions. Hence, the synopsis that forms the input of the third stage employs a tuple-to-node partitioning that is compatible with the independence assumptions of the estimation framework. By applying merge operations of low radius, the algorithm essentially tries to minimize the deviation from this partitioning, or equivalently, the deviation from the independence assumptions. This strong link between the build process and approximate query answering is key in the construction of accurate synopses.

On a final note, we discuss two heuristics employed by TugCompressJoinGraph in order to improve the effectiveness of compression. The first heuristic attempts to maximize the useful work that is performed in each iteration, by selecting the clustered relation R_m from a set of candidates that have the potential to cluster well. This set initially contains all relations in the schema. A relation R is removed from the set if it is selected in an iteration but its clustering is not effective. The idea is to avoid re-selecting R until a clustering of one of its neighboring relations increases the similarity of its nodes. This simple heuristic works well in practice and

[4]The second pass is necessary in order to label each data point with the cluster in which it belongs.

Procedure TugCompressValueGraph($\mathcal{TG}, B_V$)
Input: A TuG summary $\mathcal{TG}$ with detailed value information; a value space budget B_V.
Output: The TuG summary $\mathcal{TG}$ with concise value summaries at each node.
begin
1. Initialize sets *Clusters* and *bestHists*
2. $ct_v \leftarrow ct_v^0$; $bestHists \leftarrow \emptyset$; $bestErr \leftarrow \infty$;; $finished \leftarrow FALSE$
3. **while** NOT $finished$ **do**
4. $currentHists \leftarrow \emptyset$
5. **for** each relation R and related attribute A **do**
6. Compute a clustering of R with respect to A so that each cluster has radius at most ct_v
7. **for each** cluster **do**
8. Create a new histogram H for the distribution of A in the nodes of the cluster
9. $currentHists \leftarrow currentHists \cup \{H\}$
10. **done**
11. **done**
12. Allocate B_V units of space among histograms in *currentHists*
13. *currentError* $\leftarrow$ estimation error of *currentHists*
14. **if** *currentError* $<$ *bestError* **then**
15. $bestHists \leftarrow currentHists$; $uphillMoves \leftarrow 0$
16. **else if** $uphillMoves < MaxUphillMoves$ **then**
17. $uphillMoves++$
18. **else**
19. $finished \leftarrow TRUE$
20. **end if**
21. $ct_v \leftarrow \alpha_V \cdot ct_v$
22. **done**
23. Substitute value summaries with histograms in *bestHists*
end

Fig. 11. Algorithm TugCompressValueGraph.

provides good guidance on the issuance of calls to the (costly) clustering module. Given that the error of clustering is always controlled by parameter ct_J, we measure the effectiveness of the clustering on R by comparing the fraction of merged nodes to a threshold that we empirically set to 5%. (Hence, R is removed from the candidate list if at least 95% of its nodes are not merged.)

The second heuristic attempts to limit the introduction of error in the synopsis by alternating between lossy and lossless merge operations. More specifically, the merge operations on the selected relation R_m increase the similarity of the neighboring nodes, and may in fact cause these nodes to become all-but-one similar. TugCompressJoinGraph attempts to take advantage of this side effect and invokes TugCompressLossless to perform lossless merges on any such nodes. By virtue of Theorem 4.1, these merges result in a smaller TuG that generates precisely the same estimates. Thus, the idea is to compress further the synopsis while maintaining the same amount of error.

5.2.4 *Compressing Value Information.* The third and final stage of the construction process is realized by algorithm TugCompressValueGraph, shown in Figure 11. The algorithm receives as input a TuG summary and a storage budget B_V, and replaces the detailed value information with concise value summaries that employ at most B_V space units.

To make effective use of the allotted space budget B_V, TugCompressValueGraph creates a distinct value summary per *group of nodes* that have similar value characteristics. More concretely, consider a relation R and a related attribute A. TugCompressValueGraph computes a clustering of R based solely on the coordinates that correspond to A, thus identifying clusters of nodes that have similar value distributions for A. For each such cluster, the algorithm creates a single value summary that summarizes the union of the corresponding value distributions, and shares it among all the nodes in the cluster.

The selection of the value-based clusters is performed adaptively, using a threshold ct_V that controls the similarity of nodes within each cluster. The selection of ct_V presents an interesting trade-off: a low threshold implies higher similarity and hence more clusters, but also more summaries that share the fixed space budget B_V. TuGBuild explores this trade-off with an iterative strategy (lines 3–22), that initializes ct_V to a low value ct_V^0 and increases it gradually in order to discover more effective solutions. For a specific threshold, TuGBuild computes the clustering for each relation R and related attribute A using ct_V to control the radius of the generated clusters (line 6). (We employ again BIRCH to compute the clustering as in algorithm TugCompressJoinGraph.) Having identified the clusters, the algorithm builds the corresponding value summaries and computes an error metric for the accuracy of the overall value approximation. If the current error constitutes an improvement over the previous iteration, then the threshold is increased and another iteration is performed. (To escape local minima, we allow the search to take a pre-determined number of uphill moves before giving up.) Overall, the goal is to achieve a good trade-off between the similarity of value distributions in each cluster and the storage allocated to each histogram.

Several details of this stage depend on the approximation methods that are used to implement the value-summaries. Two key issues are the distribution of the alloted space budget B_V to the value summaries that correspond to a specific threshold ct_V, and the computation of the approximation error. In this paper, we rely on histograms as the main value summarization mechanism and thus adopt existing techniques on histogram construction [Deshpande et al. 2001]. In particular, we employ a greedy space allocation strategy that relies on marginal gains [Deshpande et al. 2001], and we measure the approximation error based on the mean squared error over all histograms.

5.3 Implementation Details

We conclude the description of the construction algorithm with a discussion on the implementation of specific modules.

5.3.1 *Storing and Manipulating the* TuG *synopsis.* To enable the scalability of the build process to large data sets, the constructed TuG is maintained in a disk-resident B-tree structure that can be manipulated efficiently using limited memory. More precisely, the keys of the B-tree are triples (R, r, S), where R is the name of a relation, r is a node of R, and S is the name of a schema neighbor of R. The B-tree entry associated with a key (R, r, S) is a list of pairs $(s, jcount(r, s))$, where s is a node of S and (r, s) is an edge in the synopsis. (We assume that the list is sorted based on the target node s.) We often use el to denote the edge list associated with

Procedure TugUpdateCV($\mathcal{TG}, R_m$)
Input: A TuG synopsis $\mathcal{TG}$; a relation R_m whose partitions have been merged.
1. $affected \leftarrow \emptyset$
2. **for each** (R_m, r, S, el) in $BTreeGet(\mathcal{TG}, R_m, *, *)$ **do**
3. Process each edge $(s, jcount(s, r))$ in el and compute $CV(\mathcal{TG}, S, r)$
4. Incorporate $CV(\mathcal{TG}, S, r)$ in $CV(\mathcal{TG}, S)$
5. Add S to $affected$
6. **done**
7. **for each** S in $affected$ **do**
8. **for each** (S, s, R, el) in $BTreeGet(\mathcal{TG}, S, *, *)$ **do**
9. **if** $R \neq R_m$ **then continue**
10. Process each edge $(r, jcount(r, s))$ in el and compute $CV(R_m, s)$
11. Incorporate $CV(\mathcal{TG}, R_m, s)$ in $CV(\mathcal{TG}, R_m)$
12. **done**
13. **done**

Fig. 12. Algorithm TugUpdateCV.

a key (R, r, S) and denote the complete entry as (R, r, S, el). We define similar keys and entries for attributes and values. Hence, given an attribute A, the entry for (R, r, A) is a list of pairs $(\nu, f(r, \nu))$ where $f(r, \nu)$ is the frequency of the value in the tuples of the node. Recall that ν is either a compressed value range if A is a numerical attribute, or an actual value if A is a categorical attribute. To unify our notation, we henceforth treat each value as a special node in the synopsis graph and define $jcount(r, \nu) = f(r, \nu)$.

A node r is physically represented as a pair $(id, tcount(r))$, where id is a unique numerical id. The same representation is used for every node s that appears in a list el. TuGBuild accesses the information in the B-tree through a generic *BTreeGet* method that receives a (possibly partial) specification of a search key and returns a stream of entries that match it. For instance, $BTreeGet(\mathcal{TG}, R, r, S)$ will return all entries that match the specific key (R, r, S). Similarly, $BTreeGet(\mathcal{TG}, R, *, *)$ will return all entries (R, r, S, el) that match the partial key $(R, *, *)$. We use the notation $(*, *, *)$ for the partial key that matches any entry, and hence $BTreeGet(\mathcal{TG}, *, *, *)$ essentially performs a complete scan of the (R, r, S, el) entries stored in the B-Tree.

The suggested organization clearly stores redundant information, as an edge (r, s) is repeated in the edge list of both r and s, which in turn increases the disk space that is required to store the B-tree. We have chosen this redundant representation as it facilitates the operations performed by TuGBuild and thus leads to a significant speed-up in construction time. Our view is that the benefit in construction time outweighs the increased storage cost, since disk space is generally an ample resource in modern systems.

We now discuss how the construction algorithm uses this physical organization to compute the *CV* metrics and also to perform merge operations.

Computing and Updating CV. The algorithm computes *CV* for all relations by performing a single scan of the B-tree entries. Essentially, the scan encounters as a group all the edges between a node s and nodes of a relation R, and can thus compute $CV(\mathcal{TG}, R, s)$ in a streaming fashion. Obviously, the $CV(\mathcal{TG}, R)$ metric

Procedure TugMergeClusters($\mathcal{TG}, R_m, \mathcal{N}_m$, *Clusters*)
Input: Synopsis $\mathcal{TG}$; Relation R_m that is merged wrt neighbor-set $\mathcal{N}_m$; A set of clusters *Clusters*.
begin
1. Initialize a hash-table *updateLog*
   /** Iterate over *Clusters* and merge each cluster in a new node **/
2. **for each** cluster $\{r_1, \ldots, r_m\}$ in *Clusters* **do**
3. **if** $m = 1$ **then continue**
4. $r_{\text{new}} \leftarrow$ new tuple partition in $\mathcal{TG}$
5. $tcount(r_{\text{new}}) \leftarrow \sum_{r \in b}(tcount(r))$
6. Define $e_i = \{(S, s, jcount(r_i, s)) \mid S \in \mathcal{N}_m \wedge\ (s, jcount(r_i, s)) \in BTreeGet(\mathcal{TG}, R_m, r_i, S)\}$
7. Merge lists $e_1, \ldots, e_m$ based on $(S, s, *)$
8. **for each** $(S, s, jcount(r_i, s))$ in the merged lists **do**
9. **if** $(S, s, jcount(r_i, s))$ is the first for S **then** $list \leftarrow \emptyset$
10. **if** $(S, s, jcount(r_i, s))$ is the first for s **then** $jcount(r_{\text{new}}, s) \leftarrow 0$
11. $jcount(r_{\text{new}}, s) \leftarrow jcount(r_{\text{new}}, s) + jcount(r_i, s)$
12. Append (r_i, r_{new}) to $updateLog[S, s]$
13. **if** $(S, s, jcount(r_i, s))$ is the last entry for s **then** append $(s, jcount(r_{\text{new}}, s))$ to $list$
14. **if** $(S, s, jcount(r_i, s))$ is the last entry for S **then** $BTreePut(\mathcal{TG}, R_m, r_{\text{new}}, S, list)$
15. **done**
16. Remove partitions $r_1, \ldots, r_m$ from $\mathcal{TG}$
17. **done**
   /** Update the edge lists of the neighbors of the new nodes **/
18. **for each** (S, s) in keys of *updateLog* **do**
19. $(S, s, R_m, el) \leftarrow BTreeGet(\mathcal{TG}, S, s, R_m)$
20. **for each** $(r, jcount(r, s)) \in el$ **do**
21. **if** $\exists r_{\text{new}} : (r, r_{\text{new}}) \in updateLog[S, s]$ **then**
22. Remove $(r, jcount(s, r))$ from el
23. Insert $(r_{\text{new}}, jcount(s, r_{\text{new}}))$ in el if it does not exist
24. **end if**
25. **done**
26. Sort el
27. $BTreePut(\mathcal{TG}, S, s, R_m, el)$
28. **done**
end

Fig. 13. Algorithm TugMergeClusters

can be computed by aggregating the individual metrics $CV(\mathcal{TG}, R, s)$.

The updating of the CV metrics is performed with the algorithm TugUpdateCV shown in Figure 12. TugUpdateCV is called after the clustering of a relation R_m in order to recompute the CV metrics of R_m and its neighbors. To perform this efficiently, the algorithm uses a similar method as the initialization of the CV metric. More precisely, for each affected relation R, it performs a scan of the corresponding B-tree entries and computes $CV(\mathcal{TG}, R, s)$ on-the-fly for each relevant node s. The final metric $CV(\mathcal{TG}, R)$ results by aggregating the individual metrics.

Performing Merge Operations. Merge operations are realized by algorithm TugMergeClusters shown in Figure 13. TugMergeClusters receives as in-

put a set *Clusters* that contains the identified clusters, and outputs a modified synopsis $\mathcal{TG}$ where the nodes in each cluster are substituted with a single node.

The algorithm works in two stages. In the first stage, it iterates over each cluster in *Clusters* and performs the corresponding merge operation. An important issue is the computation of the edge list for the new node $\tilde{r}$ that results from each merge, since the merged partitions may have common neighbors and this requires a consolidation of the corresponding edge counts. TugMergeClusters takes advantage of the B-tree organization of $\mathcal{TG}$ in order to perform this consolidation efficiently. More concretely, assume without loss of generality that the cluster comprises nodes $r_1, \ldots, r_m$. Recall that the edge entries $(s, jcount(r_i, s))$ for each r_i $(1 \leq i \leq m)$ are essentially sorted first on the schema neighbor S and then on the partition s. Hence, it is possible to perform a multi-way merge among all the lists in order to discover the neighbors that are shared among the merged partitions. (Note that this merge requires m open cursors on the B-tree, one for each merged partition r_i.) TugMergeClusters employs precisely this idea, and computes the edge list of $\tilde{r}$ by performing a single pass over the merged list of edges (lines 6–13). In addition, the algorithm records the edge substitutions that need to be performed on the neighbors of each r_i $(1 \leq i \leq m)$. More specifically, given a neighbor s of some node r_i, it becomes necessary to substitute (s, r_i) with $(s, \tilde{r})$ in the edge list of s in order to maintain the referential integrity of the synopsis. These substitutions are recorded in the hash table *updateLog* and are processed in the second stage of the algorithm. In this stage, TugMergeClusters considers each pair (S, s) for which $updateLog[S, s] \neq \emptyset$, and reads in memory the edge list returned by $BTreeGet(S, s, R_m)$. (Since these are the joining nodes to a single partition, we expect the list to be relatively short in size and thus to fit in main memory.) Each marked edge $(r_i, jcount(r_i, s))$ is then substituted with $(\tilde{r}, jcount(\tilde{r}, s))$, and the new list is written back to the B-Tree after it is sorted.

5.3.2 *Applying BIRCH on High-Dimensional Spaces.* As noted in Section 5.2, the construction algorithm employs BIRCH [Zhang et al. 1996] in order to identify merge operations of a specific radius. BIRCH has nice scalability properties, but it is well suited for data of low dimensionality. In this section, we discuss how we adopted BIRCH to handle the high dimensionality of our problem.

BIRCH represents the clustering of the data in a tree structure called the CF-Tree. The leaf nodes of the CF-Tree store the identified clusters, and the intermediate nodes store statistics for the union of clusters in the respective sub-tree. These statistics are used by BIRCH to enforce certain properties of the clustering, including the maximum radius of the identified clusters. One such statistic is the *linear sum*, defined as the sum of the point vectors in the clusters of the sub-tree. Given that the size of the linear sum is equal to the dimensionality of the space, it becomes clear that storing this statistic increases substantially the memory requirements of BIRCH when the data have is high-dimensional. This is precisely the case with the TuG construction process, as the multi-dimensional space includes one dimension per synopsis node and per distinct value.

To address this issue, we have modified BIRCH to use count-min sketches [Cormode and Muthukrishnan 2005] in order to store the large linear sums of the internal CF-Tree nodes. A count-min sketch is a fixed-size randomized data structure that

approximates the contents of a vector. Using count-min sketches, it is possible to estimate the statistics that BIRCH computes from the original linear sums, and thus the clustering algorithm itself does not require any modifications. The gain, of course, is that we reduce the space overhead of the CF-Tree, since the sketches have fixed-size and are much smaller than the original linear sum vectors. We note that we do not apply this modification to leaf clusters, as the latter have typically a lower number of non-zero coordinates and can thus be stored exactly with a manageable space overhead.

6. EXPERIMENTAL STUDY

In this section, we present the results of an extensive empirical study that we have conducted in order to evaluate the performance of the proposed TuG framework. Overall, our results on real-life and synthetic data sets verify the effectiveness of TuG summaries and demonstrate their benefits over previously proposed (table- and schema-level) summarization techniques.

6.1 Experimental Methodology

This section describes the techniques that we have used in our study, the data sets and related workloads, and the metrics for evaluating the performance of summarization techniques.

Techniques. We base our experimental study on the following summarization techniques:

- TuGs. We have completed a prototype implementation of the TuG framework that we describe in this paper. For TuG construction, our prototype uses the modified BIRCH algorithm described in Section 5.3.

- Join Synopses: We compare the performance of TuGs against the Join Synopses technique of Acharya et al. [Acharya et al. 1999]. A Join Synopsis (also known as a Join Sample) is essentially a uniform random sample of the join of a specific table along key/foreign-key relationships. As we discuss in Section 7, however, this technique can only be used on schemata that do not contain many-to-many relationships and hence we use it only on a subset of our experiments. We note that we store the samples in normalized form [Acharya et al. 1999], in order to remove redundant tuples and thus reduce storage.

- Multi-dimensional Wavelets. For schemata where Join Synopses cannot be applied, we compare the performance of TuGs against the wavelet-based techniques of Chakraborti et al. [Chakrabarti et al. 2000][5]. The key idea is to create one multi-dimensional wavelet summary per table, and to evaluate the relational operators of a query (e.g., select, project, join) entirely in the wavelet domain. Wavelet summaries are table-level synopses and can thus be applied to any type of database schema.

- Histograms. We use single-dimensional histograms as the baseline summarization technique of our study. We have opted to use the histogram-based statistics of a

[5]We are grateful to the authors of the paper for providing us with their source code.

commercial system, henceforth referred to as system X[6]. The system is set-up with detailed statistics and indices on the join and value attributes of all tables, and we use the EXPLAIN command to obtain the estimate computed by the optimizer. To the best of our knowledge, system X employs the max-diff [Poosala et al. 1996] heuristic for the creation of histograms.

Overall, our study uses two schema-level (TuGs, Join Synopses) and two table-level (Wavelets, Histograms) techniques. Histograms serve as our baseline technique and are not expected to perform well, as they do not capture any correlations across value distributions. Wavelets and Join Synopses, on the other hand, are essentially "multi-dimensional" techniques that capture the combined distribution of joins and values. They are, therefore, the main competitors to the proposed TuG synopses.

	TPCH	**IMDB**
Number of relations	8	8
Tuples in largest relation	6,000,000	2,702,114
Tuples in smallest relation	5	68,975
Size of text files	1GB	139 MB

Table I. Data Set Characteristics

Data Sets. We use two data sets in our study: (a) IMDB, a real-life data set that contains information on movies, and (b) a skewed version of the well-known TPC-H data set[7]. The characteristics of the two data sets are shown in Table I. IMDB contains many-to-many relationships between its central movies table and the remaining tables (actors, movie genres, and producers), while TPC-H contains only many-to-one relationships that originate from the central LineItem table. It should be noted that both data sets use key/foreign-key joins to encode these relationships. Based on these characteristics, we apply TuGs and histograms on both data sets, Join Synopses on TPC-H only, and Wavelets on IMDB only.

In accordance with our data model, we retain in each data set only numerical and categorical value attributes (e.g., string attributes are discarded). Moreover, to be able to examine easily the storage requirements of each summarization technique, we encode categorical values as integers.

	TPCH	**IMDB**
Number of join predicates	4-8	4-6
Number of selection predicates	1-7	1-5
Avg. value of $COUNT$ for positive queries	634,557	49,590

Table II. Workload characteristics

Query Loads. We evaluate the accuracy of each summarization technique using synthetic workloads of randomly generated queries. Each query is generated based

[6]We do not reveal the identify of the system, as the software license of the vendor prohibits the publication of experimental results using the particular system.
[7]We have used Vivek Narasaya's TPC-H generator with z=1 for generating the data.

on a randomly chosen template that defines a meaningful set of join predicates on the underlying schema. The templates that we use are listed in Section C of the Appendix. Once the template is selected, selection predicates are generated for a randomly chosen subset of the table attributes. Each predicate is also created based on a template that defines meaningful predicates on the chosen attribute. Given a date attribute, for instance, the generated predicate is a random range whose width is randomly chosen from the following options: 1 month, 6 months, and 12 months. For predicates that do not have clear semantics, we default to a random range predicate that covers 10% of the underlying domain. Our query generator ensures that each query has at least one selection predicate.

We focus our experiments on the $COUNT$ aggregate, due to its significance in query optimization and also because it is supported by all techniques (but we also examine the performance of TuGs with other aggregates). We classify queries as positive if they generate at least one result tuple (i.e., $COUNT > 0$), and negative otherwise. We generate a workload of 350 queries for each of the two categories. Table II summarizes the characteristics of the workloads for the two data sets of our study.

Evaluation Metric. We measure the performance of a summarization technique through the error of approximation. Given a true answer a and an estimate $\widehat{a}$, we use the absolute error $AE(a, \widehat{a}) = |\widehat{a}|$ for negative queries ($a = 0$), and the absolute relative error $ARE(a, \widehat{a}) = |a - \widehat{a}|/\max(a, sn)$ for positive queries. Parameter sn is a sanity bound that avoids the artificial high errors of queries with small answers. Following previous studies in summarization, we set sn to the 10-th percentile of true query results.

For a given synopsis and error metric, we report the *Cumulative Frequency Distribution* (CFD) of the metric over the queries of the workload. A point (x, y) in the CFD indicates that y% of the queries in the workload have an error that is less than or equal to x for the particular synopsis. (The interpretation of x depends of course on the choice of AE or ARE.) Using this comparator, a synopsis A is more effective than a synopsis B if the CFD of A dominates the CFD of B, i.e., a larger percentage of queries has a lower error. We have found this approach to yield more interesting comparisons compared to using a single statistic (such as, an average).

6.2 Experimental Results

We present now some of the experiments that we have conducted in our study. The specific subset captures the main traits of the wide range of results we obtained by varying the data, the query workloads, and the model parameters.

6.2.1 *Effectiveness of Construction Algorithm.* In the first set of experiments, we evaluate the effectiveness of the TuGBuild algorithm. We focus on three important aspects of the proposed algorithm: the use of metric CV, all-but-one similarity, and total construction time. In all the experiments, we set the parameters of the algorithm as follows: $ct_J^0 = ct_V^0 = 1.5$ and $\alpha_J = \alpha_V = 5\%$. Our empirical results have shown that the algorithm's performance is mostly sensitive to the starting thresholds, with values in the range $[1, 2]$ working adequately well in our

experimental set-up.

CV **Metric.** As described in Section 5, TuGBuild processes relations in decreasing order of their *CV* metric in order to minimize the error to compression ratio of the clustering process. To evaluate the effectiveness of this heuristic, we consider the second stage of the algorithm, and we compare the sequence determined by *CV* to a large number of randomly generated sequences that also lead to all-but-one minimal synopses. Given that in all cases the synopsis is a lossless compression of the starting summary, we measure the effectiveness of a given sequence as the total number of nodes in the final summary. Note that a specific random sequence may involve the same relation more than once, and hence the length of the sequence is not necessarily bounded by the number of relations in the schema. To be able to evaluate a large number of random sequences in a reasonable amount of time, we use a scaled down version of the IMDB data set that corresponds to a sample of the movies.

Random Orderings Percentiles					
min	**25%**	**50%**	**75%**	**max**	**CV**
747	1,090	1,236	1,489	1,825	780

Table III. Size distribution for random reference summaries.

Table III shows percentile information for the sizes of 1326 random reference TuGs on the scaled down version of the IMDB data set. The size of the *CV*-based reference TuG is shown in the last column. Note that the sequence corresponding to *CV* is among the random sequences that were tested. As our results indicate, the sequence of clustered relations affects significantly the size of the reference TuG. The largest random TuG, for instance, is more than twice as big as the smallest random TuG (1,825 nodes vs. 747 nodes). We observe that the *CV* metric is very effective in guiding the clustering process, and results in a small reference synopsis that is very close to the minimum size achieved by random orderings (780 nodes vs. 747 nodes for the minimum).

All But One Similarity. As we have discussed earlier, TuGBuild relies on the novel concept of all-but-one similarity in order to perform lossless merges on the data graph G_D. Here, we evaluate the impact of this choice by comparing it against the use of *complete similarity*. We focus again on the second stage of the construction algorithm, and we examine the case where the algorithm employs complete similarity to identify the lossless merges.

Table IV shows the number of nodes in the synopsis computed by the second stage for the two cases of similarity. To put the measurements in context, we also

	Data Graph	**Complete Similarity**	**All-but-one Similarity**
TPC-H	8M	4.4M	33K
IMDB	4.7M	4.5M	65K

Table IV. umber of nodes in the output synopsis of the second stage when full and all-but-one similarity are used to identify lossless merge operations.

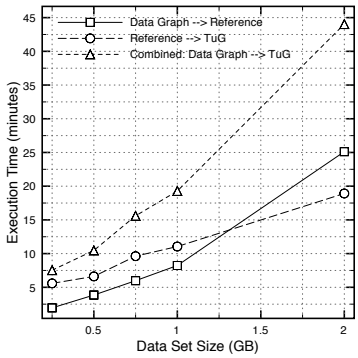


Fig. 14. Execution time of first three stages of TuGBuild as a function of data set size.

list the number of tuples in the data graph. Our results demonstrate that all-but-one similarity enables a substantially higher compression rate, outperforming complete similarity by two orders of magnitude in both data sets. For the TPCH data set, for instance, all-but-one similarity generates a reference summary of 33 thousand nodes, compared to 4.4 million nodes for full similarity. This level of compression is a key factor for the efficiency of the TuGBuild algorithm, as it reduces significantly the size of the input for the third stage of the build process.

TuG **Construction Efficiency.** This experiment examines the efficiency of the TuG construction process. We focus on the first three stages of construction, namely, building the reference synopsis and compressing the join graph, which are independent of the specific type of value summaries used. In the experiments that follow, we measure the execution time of these two components when the TuGBuild algorithm is executed on data sets of varying size. We employ the TPC-H data set and vary its size as 250MB, 500MB, 750MB, 1GB, and 2GB. (In all cases, the structural join budget B_S is set to 50KB.) All measurements are taken on an otherwise unloaded 3GHz Pentium 4 machine.

Figure 14 depicts the execution time of the first three construction stages as a function of the data set size. The results demonstrate that the construction process scales gracefully as the data set size grows. This can be attributed to the carefully tuned storage and manipulation of the synopsis data structure (see also Section 5.3) and to the identification of merge operations through highly efficient clustering algorithms. The plot also indicates that the cost of building the reference summary grows at a faster rate than the cost of compressing the join graph to 50KB. This can be explained as follows. Recall that all-but-one similar merges have a snowballing effect, i.e., the all-but-one merges of one relation can cause the nodes of neighbor relations to become all-but-one similar and so on. A large data set pronounces this effect as the starting data graph contains more tuples and thus there are more chances to find except-one similar nodes after each iteration. This leads TugCompressLossless to perform a higher number of iterations (since the stopping condition of the outer loop is not satisfied), which causes the running time to grow. Still, there is an upside to this effect: the compression ratio of

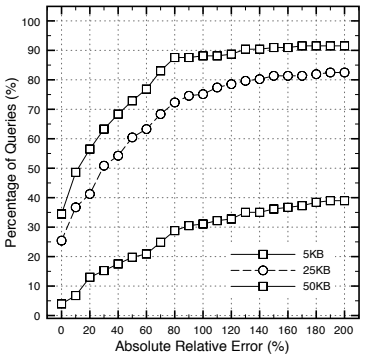


Fig. 15. Approximation error for positive queries on TPC-H vs. synopsis size.

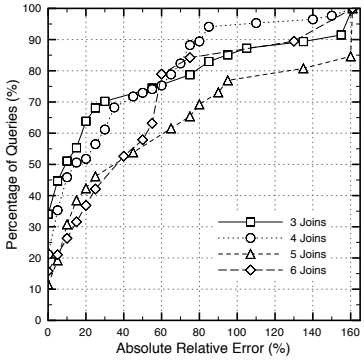


Fig. 16. CFD of estimation error for positive queries over TPC-H, varying number of join predicates.

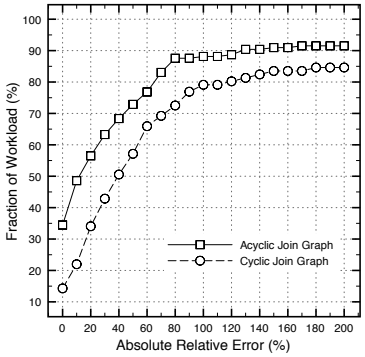


Fig. 17. CFD of estimation error for positive queries over TPC-H, with and without cyclic join graphs.

the reference synopsis is higher which limits the work of the second stage. This offsetting is demonstrated by the curve for the combined execution time of the two stages, which exhibits an almost linear scale-up with the size of the data set.

6.2.2 *Sensitivity Analysis.* In this set of experiments, we perform a sensitivity analysis of the proposed TuG synopses. We focus on the following factors that can affect the performance of TuGs: the total space allocated to the synopsis, and the complexity of the workload. In all experiments that follow, we use the synthetic TPC-H data set, and allocate an equal amount of storage to structural and value summarization. Unless otherwise noted, we use a default overall space budget of 50KB. We report results with workloads of positive queries only, as our negative workloads generated similar results.

Effect of Synopsis Size. The first experiment evaluates the effect of synopsis size on the quality of approximation. We build three different synopses with total size

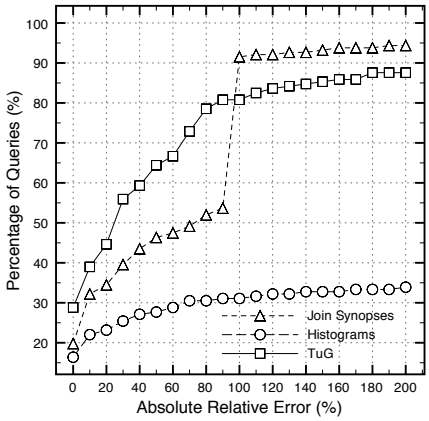
5KB, 25KB, and 50KB respectively, and measure the error of approximation of each synopsis on the test workload. The CFDs of the error are graphed in Figure 15. The results show a clear trend of diminishing errors as the synopsis size grows, and thus demonstrate that TuGBuild is able to take advantage of the additional storage in order to improve the accuracy of the synopsis. As an example, the 5KB synopsis enables less than 20% of estimation error for 15% of the workload, compared to 40% of the workload for the 25KB synopsis and 55% of the workload for the 50KB synopsis respectively.

The results also reveal a trend of diminishing returns as the storage increases. This effect can be attributed to the compression strategy of TuGBuild which is based on clustering. Recall that TuGBuild has to increase the merging radius as the storage decreases, which in turn implies that the clustering tends to group more diverse tuples in the same nodes. This heterogeneity does not affect greatly the quality of approximation as long as the clustering continues to capture the dominant correlations that exist in the data, and this is the case where an increase in the storage budget does not cause a proportional improvement in the quality of approximation. There is a tipping point, however, where the increased radius causes the computed clusters to violate the dominant correlations, and this has an adverse effect on the quality of approximation. This is the case where a small increase in storage can improve greatly the estimation error. The trend of diminishing returns is evidence that TuGBuild favors the dominant correlations in the data, and refines the clustering with respect to the less pronounced dependencies only if there is additional storage.

Effect of Workload Complexity. In this part of our study, we evaluate the accuracy of TuGs as we vary the complexity of the workload. We characterize the latter in terms of the following parameters: the number of joins in the queries, the existence of cycles in the join graph, and the type of the aggregate.

Figure 16 shows the CFD of the estimation error as we vary the number of join predicates in the queries of the test workload. As expected, the results indicate that the estimation error generally grows with the number of join predicates. Nonetheless, the accuracy of the TuG estimates remains within practical limits for the purpose of selectivity estimation. As an example, the TuG synopsis enables an estimation error of less than 40% for more than half of the workload in each case. This result reflects a significant level of accuracy given the complexity of our experimental setting: 1GB of skewed TPC-H data, and queries with 3–6 join predicates and several selection predicates.

Figure 17 depicts the CFD of the estimation error for randomly generated workloads with and without cycles in the join graph. Similar to the previous experiment, we observe that the estimation error increases with the complexity of the workload, but the overall accuracy remains reasonable for the purpose of selectivity estimation over such complex queries. The decrease in accuracy is clearly more noticeable than the previous case, due to the fact that a cyclic joining relationship corresponds to a complex correlation in the underlying data. (Essentially, a cycle can be seen as introducing one more join predicate with the additional limitation that the joining tuples are already joined with other predicates.)



(a)

	Percentiles				
Method	**0%**	**25%**	**50%**	**75%**	**100%**
TuG	0	0.1	8.1	104.9	238,766
Histograms	2.0	1,493.8	14,412	120,491	5,866,480
Join Synop.	0	0	0	0	0

(b)

Fig. 18. Estimation error of TuG*s* vs. other techniques on TPC-H: (a) Absolute relative error on positive queries, (b) Absolute error on negative queries.

6.2.3 TuG*s vs. Existing Techniques.* In this set of experiments, we compare our proposed TuG synopses against existing summarization techniques for relational data. Based on the applicability of different techniques, we split our experiments in two groups: for TPC-H, we compare TuGs against Histograms and Join Synopses, while for IMDB, where Join Synopses cannot be applied, we compare TuGs against Histograms and multi-dimensional Wavelets. In all cases, the available storage budget is set equal to the size of the histograms created by System X (approximately 30KB for TPC-H and 20KB for IMDB respectively).

Results for TPC-H. Figure 18(a) shows the Relative Error CFD for histograms, Join Synopses, and TuGs for a workload of positive queries over the TPC-H data set. Overall, we observe that TuG*s* provide an accurate summarization of the underlying data, enabling an estimation error of less than 30% for half of the queries in the workload. This level of accuracy is very effective if one considers the limited space budget and the complexity of queries in our workloads (3–7 join predicates and up to 7 selection predicates per query). Comparing across the different techniques, we note that TuGs remain more accurate for a larger part of the workload. As an example, if we fix the estimation error at 30% or less, we observe that TuG*s* enable this level of accuracy for 181 queries in the workload (55% of the total number of queries) compared to 128 queries for Join Synopses (39% of the total). (As expected, histograms perform consistently worse than both TuGs and Join Synopses.) Essentially, Join Synopses are susceptible to high estimation errors

when the sampling rate is low compared to the complexity of the data distribution. TuGs, on the other hand, summarize the structure of the complete data graph and can thus capture more effectively the key statistical traits of the data. It is interesting to note that Join Synopses perform better than TuGs for errors greater than 100%. This is an artifact of the estimation algorithm of Join Synopses, which returns an estimate of 0 tuples (100% of error) when the sample does not yield any results for the query.

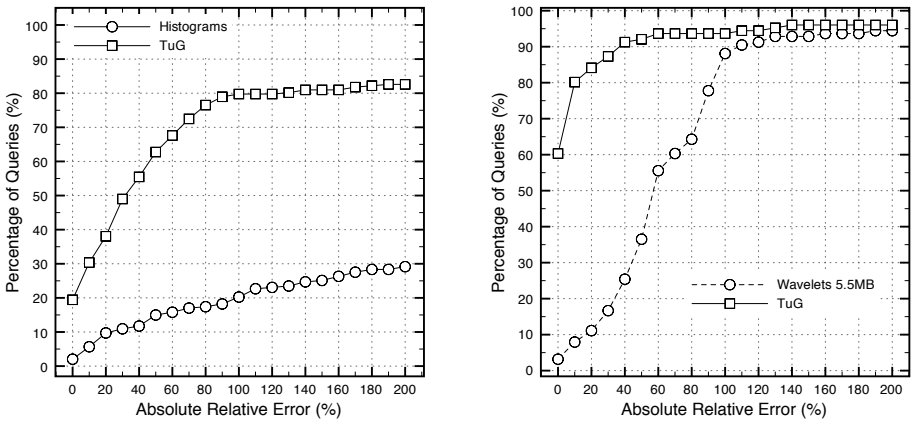
Figure 18(b) shows the estimation error of the three techniques for a workload of negative queries over TPC-H. For each technique, we report the percentiles of the absolute error metric over the complete workload. The clear winner in this case is Join Synopses, which by definition compute a perfect estimate of 0 tuples for any negative query. Our TuG synopses continue to perform well, yielding an absolute error of less than 9 tuples for 50% of the tested queries. We observe that TuGs may generate estimates of high absolute error (up to 238K tuples), but such high errors occur only for a few outlier queries in the workload.

Results for IMDB. The experiments on the IMDB data set compare the performance of TuG*s* against against Histograms and multi-dimensional Wavelets. Since it was not practical to construct Wavelet synopses on the full IMDB data set[8], we evaluate the performance of Wavelet synopses on a scaled-down version of IMDB. For the smaller data set, we retained 20% of the wavelet coefficients for each table resulting in a Wavelet summary of 5.5MB in size. We have experimented with Wavelet summaries of smaller size, but we found that the accuracy of estimation deteriorated quickly below this large space budget.

Figure 19(a) shows the CFD of relative estimation error for Histograms and TuGs, for positive queries over the IMDB data set and a space budget of 20KB. Figure 19(b) shows the same metric for Wavelets and TuGs, for the scaled-down version of IMDB. In both cases, TuGs outperform the competitor techniques by a large margin. For the full IMDB data set, TuGs enable an estimation error of less than 30% for half of the tested queries, whereas the error of the Histogram-based estimates is more than 140%. A similar picture appears in the scaled-down IMDB data set, where TuGs yield more accurate estimates than multi-dimensional Wavelets. As an example, TuG estimates have less than 40% of estimation error for 90% of the workload, while Wavelets can provide this level of accuracy for only 25% of the tested queries. As we discuss in Section 7, the reason for the bad performance of Wavelet summaries (and to some extent, of Histograms) is the presence of key/foreign-key joins in the underlying schema. This type of joins essentially results in frequency matrices that include the keys of relations, and which are difficult to summarize effectively due to their sparseness.

Table 19(c) shows the estimation error of the three techniques for a workload of negative queries. TuGs continue to outperform the other two techniques, yielding low-count estimates that are close to the true selectivity. We observe that Wavelets enable competitive estimates to TuGs, while Histogram-based estimates are again very inaccurate for negative workloads.

[8]The construction process was not finished within 12 hours.



(a) (b)

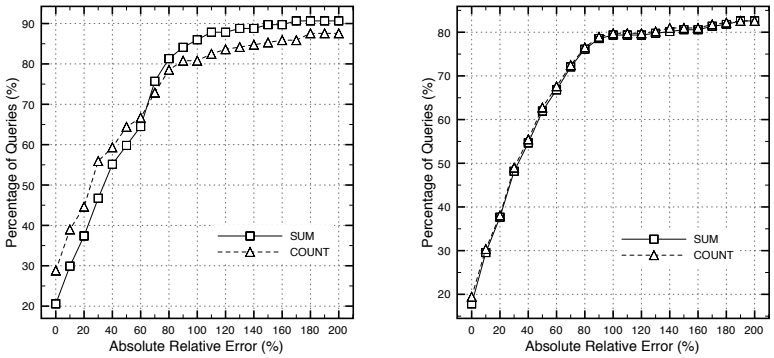
	Percentiles - Full IMDB				
Method	**0%**	**25%**	**50%**	**75%**	**100%**
TuG	0	0.1	0.8	5.8	382.2
Histograms	3	258.2	1081.4	4583.1	20218.2
	Percentiles - Scaled Down IMDB				
TuG	0	0.1	0.3	1.1	12.7
Wavelets	0	0.1	0.4	2.0	23.7

(c)

Fig. 19. Estimation error of TuG*s* vs. other techniques on IMDB: (a) Absolute relative error on positive queries for histograms and TuG*s*, (b) Absolute relative error on positive queries for wavelets and TuG*s* (scaled-down data set), (c) Absolute error on negative queries.

6.2.4 *Estimation of Other Aggregates.* Our final experiment evaluates the performance of TuG*s* when we vary the query aggregate. We focus the comparison on $COUNT$ and SUM, since SUM represents the most complex aggregate in our framework and thus represents a difficult case for TuG synopses. We employ the same synopses as the previous section, and we generate a workload of SUM queries by substituting the $COUNT$ aggregate with a summation on a numerical attribute (CUSTOMER.C_ACCTBAL on TPC-H and MOVIES.YEAR on IMDB respectively).

Figure 20 depicts the CFD of the estimation error for randomly generated workloads that compute a $COUNT$ and a SUM aggregate respectively. The results indicate that the estimation error for the SUM aggregate is somewhat increased compared to the $COUNT$ aggregate. This is expected, given that the SUM estimator involves the approximate value distribution of an attribute in combination with an estimate of $COUNT$ on the nodes of the respective relation (Section 3.2). Still, the estimation error remains within reasonable bounds for the purpose of approximate query answering. It is interesting to observe that the estimation error for SUM and $COUNT$ is almost identical in the case of the IMDB data set. For this particular data set, the TuGBuild algorithm computes a very fine clustering of movies tuples, which essentially preserves very well the value distribution of the



(a) (b)

Fig. 20. Estimation error for SUM and $COUNT$ queries: (a) TPC-H, and (b) IMDB.

year attribute. Hence, the year histogram contributes little to no error to the SUM aggregate, and most of the error comes from the count estimate on the nodes of the movies relation.

7. RELATED WORK

Relational Techniques. Relational data synopses have been investigated from the early years of database development due to their key role in query optimization. Our review of prior work uses a classification of the developed summarization techniques in *table-level* synopses and *schema-level* synopses.

Table-level synopses, such as, histograms [Aboulnaga and Chaudhuri 1999; Deshpande et al. 2001; Poosala et al. 1996], wavelets [Chakrabarti et al. 2000; Matias et al. 1998], sketches [Alon et al. 1996; Dobra et al. 2002], and table samples [Ganguly et al. 1996; Lipton et al. 1993] approximate the joint frequency distribution of values that appear in a *single* table. (Hence, to generate an approximate answer it may be necessary to combine information from several synopses.) Histograms and wavelets summarize the frequency matrix of the relation, and are most effective when the matrix contains relatively few contiguous regions that comprise similar frequencies. These techniques therefore do not perform well for key/foreign-key joins, where the inclusion of a key attribute results in a frequency matrix with widely dispersed non-zero frequencies. Similar arguments can be made for independent table samples, which have been shown to be ineffective for key/foreign-key join queries [Chaudhuri et al. 1999]. Sketch-based techniques [Alon et al. 1996; Dobra et al. 2002] have been originally developed for approximate query answering over streaming data, and in principle they can be applied as single-pass summaries over finite data sets. These techniques, however, assume a-priori knowledge of the query, while queries over finite data tend to be ad-hoc and thus difficult to know beforehand.

Schema-level synopses, such as, Join Synopses [Acharya et al. 1999] (also known as Join Samples) and Probabilistic Relational Models (PRMs) [Getoor et al. 2001], approximate the joint distribution of both *joins and values* on a *subset of tables*. As a result, approximate query answering employs a single schema-level synopsis

that covers the query. Join Synopses [Acharya et al. 1999] store uniform random samples of table joins, and compute approximate answers by scaling the answers of the query over the sample. The sample of a table join is computed by first gathering a sample of a "source" table, and then joining the sample with other tables along chains of many-to-one relationships. Hence, the total size of the join sample always equals the size of the source sample. Due to the many-to-one nature of the involved joins, it can be shown that the result is a random sample of the result of the join. PRMs [Getoor et al. 2001] employ a Bayesian network to approximates the joint frequency distribution across several joining tables. For a single relation, the Bayesian network summarizes the joint distribution of values by capturing any conditional independence that exists among attributes. This is extended to several relations by including special join indicator attributes, thus allowing the distribution of an attribute to depend on joining tuples from other relations. The authors propose a heuristic construction algorithm that greedily adds and removes binary dependencies in the Bayesian network based on a goodness metric. The aforementioned summarization methods can be applied only to queries that contain a well defined "central" relation with emanating foreign-key dependencies, i.e., queries with chains of many-to-one joins. For queries that involve many-to-many relationships, these techniques require the use of an independence assumption, similar to the one applied in our work, so that the many-to-many branches can be treated as separate selectivity factors. The issue, however, is that the application of independence is not accounted for in the respective construction algorithm, and can thus have an arbitrarily large contribution to the estimation error.

The TuG synopses that we describe in this paper fall in the second category of schema-level summaries. TuG*s* employ a general graph-summarization framework that can accommodate complex join relationships such as many-to-many and cyclic joins. Handling these complex relationships involves the application of localized independence assumptions on partitions of the original database, but the key observation is that these assumptions are taken into account in the construction algorithm. Thus, the compression of information in the synopsis is compatible with the assumptions of the approximation framework, which leads to accurate estimates. Overall, TuG*s* extend the score of approximate query answering to a larger class of real-world schemata and query loads compared to previous techniques.

XML Techniques. Graph-based summarization has been applied successfully in the construction of XML synopses [Polyzotis and Garofalakis 2006; Polyzotis et al. 2004; Zhang et al. 2006]. The general formulation of the problem is as follows: Construct a bounded-size synopsis of an XML tree that can support approximate query answering for XML queries. Several studies have proposed synopsis models that are based on a graph that describes a partitioning of the XML elements, and this has been the inspiration for the TuG model. There are, however, some important differences and challenges that arise in the application of this idea to relational data. XML data is tree-structured, whereas a relational data set is essentially an undirected graph. The same distinction carries to the supported query models, and hence it is not clear how to extend XML techniques to handle the general graph traversals that are encoded in a relational query. Another important difference is that XML techniques base their compression on complete similarity of the XML

elements, whereas TuG*s* introduce the novel concept of all-but-one similarity that allows for more aggressive compression of statistical information. As shown in the TuG study, all-but-one similarity can form the basis for the scalable construction of graph-based synopses, and thus there is the opportunity to investigate the transfer of this technique to the XML domain.

XML techniques and TuG*s* deal with a specific variant of the general problem of *graph compression*. Graph compression aims to derive a concise representation of an input graph that preserves or minimizes specific features [Feder and Motwani 1991; Gilbert 2004]. In the case of TuGs and XML techniques, the features that are preserved are the main traits of the data distribution that are relevant to approximate query answering. To the best of our knowledge, this particular variant has not been examined outside of the database literature.

Comparison to the Original Study. Our work builds on the initial TuG study [Spiegel and Polyzotis 2006] and extends it in several important directions. First, we develop a systematic approximate query answering framework that extends the application of TuG*s* to other commonly used aggregates beyond the $COUNT$ aggregate considered in the first study. Our framework includes a formalization of the independence assumptions that are applied in TuG approximation, and also provides an efficient approximate query answering algorithm for the class of tree-join queries. These two features are entirely novel compared to the original study. Second, we present a thorough discussion of the TuG construction algorithm, detailing each stage of the process and analyzing its complexity. Our discussion also touches on the physical data structures that are used to store and manipulate the working synopsis during construction. These details are valuable in understanding the inner workings of the construction algorithm and in effect the key factors that affect its efficiency. Moreover, our detailed presentation can serve as a reference for the implementation of the construction algorithm. Third, we provide the proof for the key result of lossless merge operations based on all-but-one similarity. This result and its proof are of particular interest, as all-but-one similarity is a novel feature of TuG*s* that can be potentially transferred to summarization techniques in the XML domain. Finally, we expand significantly the experimental study of TuG*s*. Our study explores the scalability of the construction algorithm, and presents a sensitivity analysis of TuG accuracy as we vary the complexity of the workload and the available storage. These important experiments were not part of the original study.

8. CONCLUSIONS AND FUTURE WORK

This article describes a new class of relational summaries, termed TuG synopses, that rely on a graph-based model in order to capture accurately the key statistical traits of the underlying data distribution. We detail the TuG model and its properties, and describe efficient algorithms for approximate query processing over concise TuG summaries. We investigate compression operations for TuG summaries and develop an efficient algorithm for constructing accurate TuG synopses for a limited space budget. Our experimental results verify the effectiveness of our approach and demonstrate its benefits over existing summarization techniques.

There are several interesting directions for future research on this topic.

Extending the query model. It will be interesting to investigate the extension of

the query model along two dimensions. The first concerns the class of GROUP-BY queries. We note that support for GROUP-BY can be worked in the current framework, by taking advantage of the fact that each embedding of the query graph carries with it histograms that can approximate the joint value distribution. Hence, it is conceivable to compute the groups and the respective aggregates for each embedding by applying histogram-based techniques [Ioannidis and Poosala 1999]. The effectiveness of this scheme is questionable, however, since it involves a separate group-by computation per embedding. The second direction is the extension of the approximation framework to set-valued answers. Currently, the TuG framework applies a unidirectional transformation from tuples to synopsis nodes. Deriving approximate set-valued answers will require the inverse transformation, so that each embedding can be expanded to a set of tuples.

Improving the efficiency of construction. Our experimental results have demonstrated that the construction algorithm exhibits an almost linear scale-up with respect to the size of the database. However, the total construction time can still be prohibitively high for certain application domains. It is interesting therefore to investigate more efficient construction techniques. Another important criterion should be simplicity and ease of deployment, as the current algorithm involves a substantial implementation effort. We believe that the link between TuG construction and clustering will prove valuable in this direction.

Theoretical study of TuG *model.* It will be interesting to embark on a theoretical study of the TuG model and its properties. One important question concerns the construction of optimal TuG synopses. Given the strong link between TuG construction and clustering, it is likely that the problem of optimal TuG construction is computationally hard. Still, the result on all-but-one similarity may offer an interesting way to deal with the hardness of the problem. A related question is the derivation of error bounds for the estimates generated by a TuG synopsis. Such bounds can improve the appeal of TuG*s* in practice, but will most likely necessitate changes to the construction algorithm and the synopsis model.

Workload- and Update-aware TuG *synopses.* In the workload-aware variant of the problem, the construction algorithm takes into account information on the expected workload in order to guide the compression of statistical information. This leads to synopses whose estimation is optimized for the expected queries. The second direction is the development of update-aware synopses, i.e., synopses that can be maintained dynamically and incrementally with data updates. Both directions are interesting extensions of the current model which assumes a static data set as the input to the construction process.

REFERENCES

Aboulnaga, A. and Chaudhuri, S. 1999. Self-Tuning Histograms: Building histograms without looking at data. In *Proceedings of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*.

Acharya, S., Gibbons, P. B., Poosala, V., and Ramaswamy, S. 1999. Join Synopses for Approximate Query Answering. In *Proceedings of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*. 275–286.

Agrawal, R., Gehrke, J., Gunopulos, D., and Raghavan, P. 1998. Automatic subspace clus-

tering of high dimensional data for data mining applications. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 94–105.

Alon, N., Matias, Y., and Szegedy, M. 1996. The Space Complexity of Approximating the Frequency Moments. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*. 20–29.

Anonymous. Details removed due to double-blind reviewing.

Chakrabarti, K., Garofalakis, M., Rastogi, R., and Shim, K. 2000. Approximate Query Processing Using Wavelets. In *Proceedings of the 26th Intl. Conf. on Very Large Data Bases*. 111–122.

Chaudhuri, S., Motwani, R., and Narasayya, V. R. 1999. On Random Sampling over Joins. In *Proceedings of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*. 263–274.

Cormode, G. and Muthukrishnan, S. 2005. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms 55,* 1, 58–75.

Deshpande, A., Garofalakis, M., and Rastogi, R. 2001. Independence is Good: Dependency-Based Histogram Synopses for High-Dimensional Data'. In *Proceedings of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*.

Dobra, A., Garofalakis, M., Gehrke, J., and Rastogi, R. 2002. Processing Complex Aggregate Queries over Data Streams. In *Proceedings of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*. 61–72.

Feder, T. and Motwani, R. 1991. Clique partitions, graph compression and speeding-up algorithms. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, 123–133.

Ganguly, S., Gibbons, P. B., Matias, Y., and Silberschatz, A. 1996. Bifocal Sampling for Skew-Resistant Join Size Estimation. In *Proceedings of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*. 271–281.

Getoor, L., Taskar, B., and Koller, D. 2001. Selectivity Estimation using Probabilistic Models. In *Proceedings of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*. 461 – 472.

Gilbert, A. C. 2004. Compressing network graphs. In *Proceedings of the LinkKDD workshop at the 10th ACM Conference on KDD*.

Ioannidis, Y. E. and Poosala, V. 1999. Histogram-Based Approximation of Set-Valued Query Answers. In *Proceedings of the 25th Intl. Conf. on Very Large Data Bases*. 174–185.

Lipton, R. J., Naughton, J. F., Schneider, D. A., and Seshadri, S. 1993. Efficient sampling strategies for relational database operations. *Theor. Comput. Sci. 116,* 1 & 2, 195–226.

Matias, Y., Vitter, J. S., and Wang, M. 1998. Wavelet-Based Histograms for Selectivity Estimation. In *Proceedings of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*. 448–459.

Polyzotis, N., Garofalakis, M., and Ioannidis, Y. 2004. Approximate xml query answers. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. 263–274.

Polyzotis, N. and Garofalakis, M. N. 2006. XCluster Synopses for Structured XML Content. In *Proceedings of the 22nd Intl. Conf. on Data Engineering*, L. Liu, A. Reuter, K.-Y. Whang, and J. Zhang, Eds. IEEE Computer Society, 63.

Poosala, V., Ioannidis, Y. E., Haas, P. J., and Shekita, E. J. 1996. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*. 294 – 305.

¡¡¡¡¡¡¡ .mine

Spiegel, J. and Polyzotis, N. 2006. Graph-based synopses for relational selectivity estimation. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 205–216.

======= ¿¿¿¿¿¿¿ .r1042

Zhang, N., Ozsu, M. T., Aboulnaga, A., and Ilyas, I. F. 2006. Xseed: Accurate and fast cardinality estimation for xpath queries. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*. 61.

Zhang, T., Ramakrishnan, R., and Livny, M. 1996. BIRCH: An Efficient Data Clustering Method for Very Large Databases. In *Proceedings of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*.

Symbol	Interpretation
$n_{\mathcal{S}}$	Number of nodes in $\mathcal{T}G(= \lvert T_{\mathcal{S}} \rvert)$
$e_{\mathcal{S}}$	Number of edges in $\mathcal{T}G$
$d_{\mathcal{S}}$	Max degree of a node in $\mathcal{T}G$
n_{PV}	Max number of compressed value ranges for each attribute
n_{Sc}	Max number of schema neighbors for a relation

Table V. Notation for Complexity Analysis of TuG construction.

A. COMPLEXITY ANALYSIS OF TUG CONSTRUCTION

For ease of reference, the notation used in our analysis is summarized in Table 8. To simplify exposition, we ignore the time- and space-complexity of invoking the clustering module, since they depend on the specific choice of the clustering algorithm. (Recall that in our work we employ BIRCH [Zhang et al. 1996] as the clustering algorithm.)

Complexity Analysis of TugCompressLossless. We now consider the time and space complexity of the TugCompressLossless algorithm. Our presentation examines a single iteration of the algorithm, since the number of iterations depends on the structure of the input graph and is thus difficult to derive analytically. To simplify our analysis, we assume that the space complexity of the B-tree data structure is $O(1)$, since it is stored on disk and accessed through a fixed-size buffer pool, and that a single B-tree look-up has complexity $O(log(n_{\mathcal{S}}))$.

We start by analyzing TugMergeClusters which is invoked as part of each iteration. The merge operation of each cluster involves several steps that can be analyzed as follows: the list merge involves opening cursors to the B-tree and reading the edge lists of the m nodes and thus has complexity $O(mlog(n_{\mathcal{S}}) + e' \cdot log(m))$, where e' is the total number of edges for the merged nodes and the $log(m)$ factor comes from the priority queue that is needed to pick the list with the least edge at each step; the loop in lines 6–13 is executed $O(e')$ times with a complexity of $O(1)$ per iteration; inserting the edge information for r_{new} has a complexity of $O(n_{\text{Sc}}log(n_{\mathcal{S}}) + e')$, since it involves a look-up and the copy of the edge list for each neighbor S; finally, removing the entries for nodes $r_1, \ldots, r_m$ has complexity $O(mlog(n_{\mathcal{S}}) + e')$, since each removal requires a look-up plus a traversal of the edge list. This brings the total complexity of a single merge to $O(mlog(n_{\mathcal{S}}) + e' \cdot log(m) + n_{\text{Sc}}log(n_{\mathcal{S}}))$. Since $n_{\text{Sc}} = O(e')$, we can simplify the complexity to $O(mlog(n_{\mathcal{S}}) + e' \cdot log(m) + e'log(n_{\mathcal{S}}))$ and further to $O(mlog(n_{\mathcal{S}}) + e'log(n_{\mathcal{S}}))$ since $log(m) = O(log(n_{\mathcal{S}}))$. It follows that the time-complexity across all iterations of the loop in lines 6–13 is $O(n_{\mathcal{S}} \cdot log(n_{\mathcal{S}}) + e_{\mathcal{S}}log(n_{\mathcal{S}})) = O(e_{\mathcal{S}} \cdot log(n_{\mathcal{S}}))$ since we expect that $n_{\mathcal{S}} = O(e_{\mathcal{S}})$. For the second stage of the algorithm, observe that reading (and writing) the edge list of a node involves a B-tree look-up plus a copy, with a total complexity of $O(log(n_{\mathcal{S}}) + e'')$, where $e'' = O(n_{\mathcal{S}})$ denotes the number of edges for the specific node. Performing the substitutions has linear complexity $O(e'')$, and the final sorting of the list has complexity $O(e''log(e'')) = O(e''log(n_{\mathcal{S}}))$. Given that there are $O(n_{\mathcal{S}})$ keys in *updateLog*, this brings the total complexity to $O(n_{\mathcal{S}}log(n_{\mathcal{S}}) + e_{\mathcal{S}} + e_{\mathcal{S}}log(n_{\mathcal{S}})) = O(e_{\mathcal{S}}log(n_{\mathcal{S}}))$. Overall, the time complexity of TugMergeClusters is $O(e_{\mathcal{S}}log(n_{\mathcal{S}}))$.

We are now ready to derive the complexity of a single iteration of TugCompressLossless. The clustering step performs a single pass over the edge lists of R_m nodes, with a total complexity of $O(e_{\mathcal{S}})$. As discussed previously, the invocation of TugMergeClusters has a total complexity of $O(e_{\mathcal{S}} log(n_{\mathcal{S}}))$. Finally, the complexity of updating the clusterability values is $O(e_{\mathcal{S}})$, since it requires a single pass over the edges in the B-tree. Overall, the time complexity of a single iteration is $O(e_{\mathcal{S}} log(n_{\mathcal{S}}))$.

We now consider the space complexity of the algorithm. We begin again with algorithm TugMergeClusters. We note that: *updateLog* records $O(e_{\mathcal{S}})$ substitutions; jv has length $O(d_{\mathcal{S}})$; *Clusters* stores $O(n_{\mathcal{S}})$ node-ids; and, TugMergeClusters has complexity $O(e_{\mathcal{S}})$, as described previously. This brings the space complexity of TugMergeClusters to $O(e_{\mathcal{S}})$.

Moving to TugCompressLossless, it is straightforward to verify that: *list* will record at most e' edges; *updateLog* will be updated with one substitution for each edge of the merged nodes; and *el* will have at most $e_{\mathcal{S}}$ edges. Taking into account the $O(e_{\mathcal{S}})$ space complexity of the call to TugMergeClusters, this brings the total space complexity to $O(e_{\mathcal{S}})$.

Complexity Analysis of TugCompressJoinGraph**.** We analyze the complexity of TugCompressJoinGraph focusing again on a single iteration of the algorithm. The total number of iterations is non-trivial to derive analytically, since it depends on the structure of the join graph, the series of lossless merge operations that lead to the initial summary $\mathcal{TG}$, and the size of the space budget.

At an abstract level, each iteration consists of the following steps: performing the merge operations; updating the CV metrics; and performing an optional call on TugCompressLossless. We consider each step in turn. The complexity of performing the merge operations remains $O(e_{\mathcal{S}} log(n_{\mathcal{S}}))$, as shown earlier. The complexity of updating the CV metrics is $O(e_{\mathcal{S}})$. Thus, the total complexity of performing lossy merges with respect to one relation is $O(e_{\mathcal{S}} log(n_{\mathcal{S}}))$. We note that a single iteration of TugCompressJoinGraph may comprise several iterations of TugCompressLossless, each one with complexity $O(e_{\mathcal{S}} log(n_{\mathcal{S}}))$. The specific number of iterations is not trivial to analyze, however, as it depends again on the distribution of joins in the data and the lossy merges performed in previous iterations.

The algorithm inherits the space complexity of the algorithms that it invokes, which, based on our previous analysis, is $O(e_{\mathcal{S}})$. Overall, it follows that the total space complexity is $O(e_{\mathcal{S}})$.

Complexity Analysis of TugCompressValueGraph**.** As in the previous stages of the algorithm, we analyze the complexity of the algorithm in terms of a single iteration. The total number of iterations is highly dependent on the statistical characteristics of the data (e.g., the distribution of values and joins among tuples), but also on the budget B_S that determines the compression of the join graph and thus the value distribution at each tuple partition. We also ignore the time- and space-complexity of the clustering module, as they depend on the particular choice of the clustering algorithm.

A single iteration of the algorithm involves a clustering step, the initialization of histograms, and the allocation of the available space among the different histograms.

The initialization of histograms has complexity $O(n_{\mathcal{S}})$ as in the worst case each cluster comprises a single node partition. Finally, the histogram construction algorithm pre-computes the split points for all histograms, sorts them, and iteratively selects the split point with the least increase in error. As shown in [Deshpande et al. 2001], the complexity of this algorithm is $O(n_{\mathcal{S}} n_{PV} log(n_{PV}) + B_V log(n_{\mathcal{S}}))$. Thus, the total complexity of one iteration is $O(n_{\mathcal{S}} n_{PV} log(n_{PV}) + B_V log(n_{\mathcal{S}}))$.

The space complexity of one iteration is $O(n_{\mathcal{S}} n_V)$ as the space allocation algorithm records $O(n_{\mathcal{S}})$ histogram structures and n_V possible split-points per histogram.

B. PROOFS OF THEORETICAL RESULTS

We present the detailed proof for the lossless merge of all-but-one similar nodes (Theorem 4.1). To simplify exposition, we assume that the set of merged nodes contains precisely two nodes, termed u and v. The extension to the merge of several nodes is straightforward.

In what follows, we use w to denote the new (merged) node in summary $\mathcal{T}G'$. We consider the case that u and v are similar with respect to all relations and attributes except a single relation S. The case of similarity except one attribute is handled similarly. To simplify our presentation, we assume that u and v have the same neighbors in S albeit with different ratios. If u does not have a specific neighbor s that v has, then we assume that the edge (u, s) exists and $jcount(u, s) = 0$. We assume a similar transformation for v. This assumption ensures that u and v have exactly the same neighbors in the synopsis graph and the same values, since u and v are similar with respect to all other relations and attributes.

Let Q be a complex query and let $Q_w(\mathcal{T}G')$ be the set of embeddings in $\mathcal{T}G'$ that contain w. Define $Q_u(\mathcal{T}G)$ and $Q_v(\mathcal{T}G)$ similarly. Since u and v have the same neighbors, there is a 1-1 correspondence between each embedding e_v in $Q_v(\mathcal{T}G)$ and an embedding $e_u \in Q_u(\mathcal{T}G)$ that substitutes v with u. Similarly, for each embedding $e_w \in Q_w(\mathcal{T}G')$, we can define a corresponding embedding e_u in $Q_u(\mathcal{T}G)$ and a corresponding embedding e_v in $Q_v(\mathcal{T}G)$. We denote these corresponding embeddings in $Q_u(\mathcal{T}G)$ and $Q_v(\mathcal{T}G)$ as e_w/u and e_w/v respectively.

Our proof is structured as follows. First, we show that the contribution of an embedding e_w for an aggregate X is the same as the combined contribution of the corresponding embeddings e_u and e_v. The lossless property of the merge then follows as a direct consequence. The equivalent contribution follows immediately if e_w does not map any query edge to a synopsis edge (w, s) for some node s of the except-one relation S, since w, u, and v have the same average join counts and values to every other relation and attribute. Hence, we only consider the case where e_w maps a query edge to an edge (w, s). This implies that the edges (u, s) and (v, s) are used in the embeddings e_w/u and e_w/v respectively. Note that, by definition of our query model, there can be only one such edge (w, s) since the query model specifies that each relation name appears exactly once.

LEMMA B.1. *$COUNT(e_u) + COUNT(e_v) = COUNT(e_w)$ if $e_u = e_w/u$ and $e_v = e_w/v$.*

Proof:

Let $\mathcal{N}_{uv}$ denote the set of neighbors of u and v in the two embeddings excluding s. Due to all-but-one similarity, u and v have the same join ratios for nodes in $\mathcal{N}_{uv}$, i.e., $p_r[u] = p_v[u]$ for $r \in \mathcal{N}_{uv}$.

Following the model of Section 3.2, we can express the count of an embedding as the product of three factors: a product of the node counts, a product of the join probabilities, and a product of the selection probabilities. For embedding e_u, we express the corresponding count aggregate as follows:

$$COUNT(e_u) = NProd \cdot tcount(u)tcount(s) \cdot SProd \cdot JProd \cdot \prod_{r \in \mathcal{N}_{uv}} Prob(u \bowtie r) \cdot Prob(u \bowtie s)$$

Here, *NProd* denotes the product of node counts for all nodes in the embedding except u and s, $SProd$ denotes the product of selection probabilities, and *RProd* denotes the product of join ratios for all edges that are not adjacent to u. Since v differs from u only on the join probability to s, we can express the corresponding aggregate $COUNT(e_v)$ as follows:

$$COUNT(e_v) = NProd \cdot tcount(v)tcount(s) \cdot SProd \cdot RProd \cdot \prod_{r \in \mathcal{N}_{uv}} Prob(v \bowtie r) \cdot Prob(v \bowtie s)$$

We note that, due to all-but-one similarity, it holds that $p_w[r] = p_u[r] = p_v[r]$ for $r \in \mathcal{N}_{uv}$, and hence it follows directly that $Prob(w \bowtie r) = Prob(u \bowtie r) = Prob(v \bowtie r)$. By summing up the two aggregates and performing standard algebraic manipulations, we arrive at the desired result:

$$\begin{aligned}
COUNT(e_v) + COUNT(e_w) &= NProd \cdot Sprod \cdot RProd \cdot \prod_{r \in \mathcal{N}_{uv}} Prob(w, r) \cdot \\
&\quad (tcount(v)tcount(s)Prob(v \bowtie s) + tcount(u)tcount(s)Prob(u \bowtie s)) \\
&= NProd \cdot Sprod \cdot Rprod \cdot \prod_{r \in \mathcal{N}_{uv}} Prob(w \bowtie r) \cdot (jcount(v, s) + jcount(u, s)) \\
&= NProd \cdot Sprod \cdot Rprod \cdot \prod_{r \in \mathcal{N}_{uv}} Prob(w \bowtie r) \cdot jcount(w, s) \\
&= NProd \cdot tcount(w) \cdot Sprod \cdot Rprod \cdot \prod_{r \in \mathcal{N}_{uv}} Prob(w \bowtie r) \cdot \cdot Prob(w \bowtie s) \\
&= COUNT(e_w)
\end{aligned}$$

∎

LEMMA B.2. *$SUM(e_u) + SUM(e_v) = SUM(e_w)$ if $e_u = e_w/u$ and $e_v = e_w/v$.*

Proof: Let A be the attribute on which the SUM aggregate is computed. As mentioned in Section 3.2, $SUM(e_w) = \sum_\nu \nu Prob(\sigma_{A=\nu}(w))COUNT(e_w)$. Since u and v are all-but-one similar with respect to all value attributes, it holds that $p_u[\nu] = p_v[\nu] = p_w[\nu]$ for all values ν, and hence $Prob(\sigma_{A=\nu}(w)) = Prob(\sigma_{A=\nu}(v)) = Prob(\sigma_{A=\nu}(u))$. By Lemma B.1, it holds that $COUNT(e_w) = COUNT(e_v) + COUNT(e_u)$. We complete the proof as follows:

$$
\begin{aligned}
SUM(e_w) &= \sum_{\nu} SUM(e_w, \nu) \\
&= \sum_{\nu} COUNT(e_w) \cdot \nu \cdot Prob(\sigma_{A=\nu}(w)) \\
&= \sum_{\nu} (COUNT(e_v) + COUNT(e_u)) \cdot \nu \cdot Prob(\sigma_{A=\nu}(w)) \\
&= \sum_{\nu} COUNT(e_v) \cdot \nu \cdot Prob(\sigma_{A=\nu}(v)) + \sum_{\nu} COUNT(e_u) \cdot \nu \cdot Prob(\sigma_{A=\nu}(u)) \\
&= SUM(e_u) + SUM(e_v)
\end{aligned}
$$

∎

LEMMA B.3. $\min(MIN(e_v), MIN(e_u)) = MIN(e_w)$ *if* $e_u = e_w/u$ *and* $e_v = e_w/v$.

Proof: Assume that the aggregated attribute A appears in the relation that corresponds to w. The lemma follows immediately since w summarizes the distribution resulting from the union of $values(u, A)$ and $values(v, A)$. In the case that A is not relevant to w, then the result is obvious since e_w, e_v, and e_u record the exact same value summary for this particular attribute. ∎

LEMMA B.4. $\max(MAX(e_v), MAX(e_u)) = MAX(e_w)$ *if* $e_u = e_w/u$ *and* $e_v = e_w/v$.

Proof: Similar as the previous lemma. ∎

THEOREM B.1. *Consider a* TuG $\mathcal{TG}$ *and two all-but-one similar nodes* u *and* v. *Let* $\mathcal{TG}$*' be the summary that results from* `merge`$(\{u, v\})$, *and let* w *be the new node. Given a query* $(Q, Aggr)$, *it holds that* $Aggr(Q(\mathcal{TG})) = Aggr(Q(\mathcal{TG}'))$.

Proof: We partition $Q(\mathcal{TG}')$ in two subsets: $Q_w(\mathcal{TG}')$ that contains the embeddings with w, and $Q_{\bar{w}}(\mathcal{TG}')$ that contains the remaining embeddings. Similarly, we partition $Q(\mathcal{TG})$ in $Q_v(\mathcal{TG})$, $Q_u(\mathcal{TG})$, and $Q_{\bar{uv}}(\mathcal{TG})$. Since w represents the merge of u and v, it holds that $Q_{\bar{w}}(\mathcal{TG}') = Q_{\bar{uv}}(\mathcal{TG})$. Moreover, as we have stated earlier, each embedding $e_w \in Q_w(\mathcal{TG}')$ corresponds to distinct embeddings $e_u = e_w/u$ and $e_v = e_w/v$ respectively in $Q(\mathcal{TG})$.

We prove the theorem for each aggregate in turn.

COUNT. We express the computation on $\mathcal{TG}'$ as follows:

$$
COUNT(Q(\mathcal{TG}')) = \sum_{e_w \in Q_w(\mathcal{TG}')} COUNT(e_w) + \sum_{e \in Q_{\bar{w}}(\mathcal{TG}')} COUNT(e)
$$

Clearly, the second summand is unchanged in $\mathcal{TG}$ since $Q_{\bar{w}}(\mathcal{TG}') = Q_{\bar{uv}}(\mathcal{TG})$. For the first summand, we employ Lemma B.1 which asserts that $COUNT(e_w) = COUNT(e_w/u) + COUNT(e_w/v)$. The theorem follows.

SUM. We follow a similar approach and express the computation of *SUM* as two separate components:

$$SUM(Q(\mathcal{T}G')) = \sum_{e_w \in Q_w(\mathcal{T}G')} SUM(e_w) + \sum_{e \in Q_{\bar{w}}(\mathcal{T}G')} SUM(e)$$

Clearly, the computation on $Q_{\bar{w}}$ remains unchanged on synopsis $\mathcal{T}G$. For the computation on $Q_w(\mathcal{T}G')$, we employ Lemma B.2 and follow a similar methodology as the previous case.

AVG. This follows from our previous results on SUM and $COUNT$.

MIN. We can express the computation of this aggregate as follows:

$$MIN(Q(\mathcal{T}G'), A) = \min(\min_{e_w \in Q_w(\mathcal{T}G')}(MIN(e_w)), \min_{e \in Q_{\bar{w}}(\mathcal{T}G')}(MIN(e)))$$

From this point onward we proceed as in the previous cases, using Lemma B.3 to handle the computation on $Q_w(\mathcal{T}G')$.

MAX. We handle this exactly as the previous case, using the operator max and Lemma B.4. ∎

C. TEMPLATES FOR THE GENERATED WORKLOADS

The following tables show the join templates that were used for the generation of the test workloads.

TPC-H
LINEITEM⋈ ORDERS⋈ PART⋈ PARTSUPP
CUSTOMER⋈ LINEITEM⋈ ORDERS⋈ PART⋈ PARTSUPP
CUSTOMER⋈ LINEITEM⋈ NATION⋈ ORDERS⋈ PART⋈ PARTSUPP⋈ REGION
LINEITEM⋈ ORDERS⋈ PARTSUPP⋈ SUPPLIER
CUSTOMER⋈ LINEITEM⋈ ORDERS⋈ PARTSUPP⋈ SUPPLIER
LINEITEM⋈ ORDERS⋈ PART⋈ PARTSUPP⋈ SUPPLIER
CUSTOMER⋈ LINEITEM⋈ ORDERS⋈ PART⋈ PARTSUPP⋈ SUPPLIER

IMDB
ACTORS⋈ CAST⋈ MGENRES⋈ MOVIES⋈ PRODUCED_BY⋈ PRODUCERS
ACTORS⋈ CAST⋈ MOVIES⋈ PRODUCED_BY⋈ PRODUCERS
ACTORS⋈ CAST⋈ DIRECTED_BY⋈ DIRECTORS⋈ MOVIES
MGENRES⋈ MOVIES⋈ PRODUCED_BY⋈ PRODUCERS
ACTORS⋈ CAST⋈ MGENRES⋈ MOVIES
DIRECTED_BY⋈ DIRECTORS⋈ MGENRES⋈ MOVIES
ACTORS⋈ CAST⋈ DIRECTED_BY⋈ DIRECTORS⋈ MGENRES⋈ MOVIES