Making Aggregation Work in Uncertain and Probabilistic Databases

Raghotham Murthy, Robert Ikeda, Jennifer Widom
Stanford University
{rsm,rmikeda,widom}@cs.stanford.edu

Abstract

We describe how aggregation is handled in the *Trio* system for uncertain and probabilistic data. Because "exact" aggregation in uncertain databases can produce exponentially-sized results, we provide three alternatives: a *low* bound on the aggregate value, a *high* bound on the value, and the *expected* value. These variants return a single result instead of a set of possible results, and they are generally efficient to compute for both full-table and grouped aggregation queries. We provide formal definitions and semantics and a description of our open-source implementation for single-table aggregation queries. We study the performance and scalability of our algorithms through experiments over a large synthetic data set. We also provide some preliminary results on aggregations over joins.

Note to referees: *Our initial results were published as a workshop paper [17]. This paper includes the following additional material: (1) More algorithms for different aggregate variants (Section IV). (2) Proofs of the key lemmas and theorem for approximating expected-average (Section V and Appendix). (3) Performance experiments (Section VI) (4) Preliminary results on aggregation over joins (Section VII)*

Index Terms

Database Management, Query Processing

I. INTRODUCTION

Trio is a prototype database management system under development at Stanford, designed specifically for storing and querying data with *uncertainty* and *lineage* [19, 34]. Trio's query language, *TriQL*, is an adaptation and extension of SQL [32]. Previous papers and the Trio system have focused so far on *select-project-join* queries and some set operations [2, 19]. In this paper we begin tackling the problem of TriQL queries with aggregation [15].

In a typical semantics for uncertain or probabilistic data based on *possible-instances* (also called *possible-worlds*), it is well known that the result size for a query with aggregation can grow exponentially with data size [3, 6, 22]: there can be an exponential number of possible-instances, with potentially different aggregation results in each one. For example, if an uncertain relation has 10 tuples, each of which exists with probability 0.9, then an aggregate function like `SUM` returns 2^{10} values in its result (modulo duplicates). To make computation feasible, and for general usability, in Trio we decided to offer several variants to "exhaustive" aggregation. Specifically, we support variants for each aggregate function that return a single value over uncertain data, instead of a set of possible values: a function returning the lowest possible value

of the aggregate result (*low*), the highest possible value (*high*), or the expected value (*expected*), the latter of which takes confidences or probabilities into account. Note that the size of the interval between the low and high values of an aggregate can be a useful indicator of the degree of uncertainty in the data [27]. It turns out that almost all of these functions can be computed efficiently in the Trio system, and the one exception (*expected-average*) can be approximated effectively.

The paper proceeds as follows. In Section II we review the Trio data model and query semantics, and we introduce a running example. Then we cover the paper's main contributions:

- We provide formal definitions for our aggregate function variants *low*, *high*, and *expected*. Their semantics is defined in terms of *exhaustive* aggregation, which itself follows TriQL's formal query semantics (Section III).
- We briefly review the implementation of Trio's data model and query language, which is built on top of a conventional DBMS [19]. We then show how Trio's data-encoding scheme made it easy for us to implement nearly all of our aggregate function variants (Section IV).
- Our one difficult function, *expected-average*, is implemented using the approximation of *expected-sum* divided by *expected-count*. We identify a special case where we obtain the correct *expected-average* (Section V).
- We have implemented all of our algorithms in the Trio prototype DBMS [19]; they are available in the online demo as well as the open-source distribution [32]. In Section VI we report on performance experiments over a large synthetic uncertain database. Our performance experiments demonstrate (1) scalability of our algorithms in data size, (2) the fact that our algorithms are not affected by the amount of uncertainty in the data and (3) the usefulness of *low* and *high* aggregates as indicators of uncertainty in the data.
- We provide preliminary results on the problem of performing aggregations over joins (Section VII). This problem is particularly challenging because joins in uncertain databases introduce "correlated uncertainty" across tuples, which in turn can affect aggregation results.

Related work is discussed in Section VIII.

In this paper we focus on aggregation queries posed over base tables, and we do not consider lineage. Extending our techniques to Trio's full *ULDB* model including derived tables with lineage [2], and to the full TriQL query language [32], is the subject of future work; see

Section IX.

II. Data Model and Running Example

We review Trio's model for uncertain data, which is similar to most models for uncertain and probabilistic data. Our overview is brief, and it does not include aspects of Trio's *ULDB* data model (notably lineage) not relevant to this paper. For a full treatment see [2].

An uncertain relation is a multiset of *x-tuples*. Each x-tuple is comprised of one or more mutually-exclusive *alternatives*. Each alternative is a regular tuple and has an associated probabilistic confidence value in $[0, 1]$. In a single x-tuple, if Σ is the sum of the confidence values of all alternatives, then $0 \leq \Sigma \leq 1$. If $\Sigma < 1$, then the entire x-tuple may not exist; we represent this case as an additional special alternative denoted ϕ, whose confidence is $(1 - \Sigma)$.[1] Trio also supports uncertain relations with alternatives but no confidence values [2]; restricting the definitions and algorithms in this paper to ignore confidence values is straightforward and not further discussed.

As is typical, an uncertain relation in Trio represents a set of *possible-instances*. The following observations have been developed formally in [2] and elsewhere:

- Each possible-instance is a regular relation (multiset) containing one alternative tuple from each x-tuple, or none if alternative ϕ is selected. The total number of possible-instances is the product of the number of alternatives of each x-tuple.
- Each possible-instance has an associated *probability*, which is the product of the confidence values for the selected alternatives (including ϕ) in the instance. The possible-instance probabilities sum to 1.
- Each alternative a is selected in some subset of the possible-instances. The probabilities of these instances sum to the confidence of alternative a. As a special case, if an alternative has confidence 1.0 (and therefore is the only alternative in its x-tuple), then it appears in all possible-instances.
- If a possible-instance has only ϕ alternatives, then it is empty, denoted $\{\}$. There can be at most one such instance.

[1] We are using a slightly different notation from [2], for presentation purposes only. In [2], *maybe-tuples* with "?" annotations denote x-tuples that may not exist, instead of ϕ alternatives as we use here. The two representations are isomorphic.

time	(color, length)
1	(gray,20) *.5* || (black,20) *.4* || ϕ *.1*
2	(black,18) *.8* || (brown,16) *.2*
2	(brown,20) 1.0

Fig. 1. Relation `Sightings`(`time`, `color`, `length`)

All of our techniques apply to Trio's data model, and hence to most uncertain and probabilistic databases.

A. Running Example: Squirrel Sightings

We present a fabricated, highly-simplified eco-monitoring application for examples throughout the paper. (This application was inspired partially by the *Christmas Bird Count* [5], an original motivating example for Trio [34].) Human volunteers observe squirrels on the Stanford campus and record their observations. Volunteers record the *color* (species) and *length* of each squirrel sighting, along with the time of the observation.

Figure 1 shows a Trio relation `Sightings(time, color, length)`. Each x-tuple in `Sightings` represents one observation. We follow the new Trio convention of denoting the *certain* attributes separately from the alternatives: An attribute is certain if, in each x-tuple, it contains the same value in all alternatives [32]. In `Sightings`, attribute `time` is certain, i.e., we assume the time of each observation is accurate. In an observation, a volunteer may be uncertain about either the color or the length of the squirrel, or both, and may assign relative confidence values to the different possibilities. The symbol || separates the alternatives. For example, in the first listed observation, the volunteer was not sure if the 20-centimeter squirrel was black (confidence 0.5) or gray (confidence 0.4). Furthermore, he has only 0.9 confidence that the animal was a squirrel at all, resulting in a ϕ alternative with confidence 0.1. In the second listed observation, the volunteer saw either an 18-centimeter black squirrel (confidence 0.8) or a 16-centimeter brown squirrel (confidence 0.2). In the third listed observation, the volunteer was certain that he saw a 20-centimeter brown squirrel (confidence 1.0).

Relation `Sightings` has six possible-instances, labeled I_1 through I_6 in Figure 2. Each instance I_i has a probability P_i associated with it, as described above and formalized in [2]. Note that ϕ alternatives are not explicit in the instances.

$I_1 =$

time	color	length
1	gray	20
2	black	18
2	brown	20

$P_1 = 0.4$

$I_2 =$

time	color	length
1	gray	20
2	brown	16
2	brown	20

$P_2 = 0.1$

$I_3 =$

time	color	length
1	black	20
2	black	18
2	brown	20

$P_3 = 0.32$

$I_4 =$

time	color	length
1	black	20
2	brown	16
2	brown	20

$P_4 = 0.08$

$I_5 =$

time	color	length
2	black	18
2	brown	20

$P_5 = 0.08$

$I_6 =$

time	color	length
2	brown	16
2	brown	20

$P_6 = 0.02$

Fig. 2. Possible-instances of `Sightings` relation

III. AGGREGATION

We first discuss aggregate functions applied to the entire table, which we refer to as *full-table* aggregation. We extend in Sections III-B–III-C to *grouped* aggregation, i.e., queries with a GROUP BY clause. Until Section VII we restrict ourselves to aggregation queries over a single base table. Filtering predicates in the WHERE clause are permitted—they are applied before aggregation and do not affect our definitions or techniques. HAVING clauses are supported as well; further discussion appears in Section IV.

Recall the semantics of relational queries over uncertain databases as described in, e.g., [2]: Consider an uncertain relation U representing n possible-instances $I_1, \ldots, I_n$. A query Q on U produces a result relation R that represents the set of n possible answers: $Q(I_1), \ldots, Q(I_n)$. Aggregation follows the same semantics: An aggregation query on an uncertain relation produces a result whose possible-instances are the result of the aggregation applied to the possible-instances of the input relation.

More concretely, consider an aggregation query A with COUNT, SUM, AVG, MIN, or MAX applied to an uncertain relation U. (DISTINCT aggregates are not covered in this paper; they will be incorporated as future work.) The result contains exactly one x-tuple, which has one alternative a_j corresponding to each possible-instance I_j of U. Alternative a_j contains the result of aggregation query A over instance I_j. Note that over an empty relation (i.e., over the empty possible-instance, if it exists for U), SQL semantics dictates that the result of COUNT is 0, while the result of SUM,

Full:

avgLength
(19.33) *.4* || (18.67) *.1* || (19.33) *.32* || (18.67) *.08* || (19) *.08* || (18) *.02*

Grouped:

color	avgLength
black	(18) *.4* || (19) *.32* || (20) *.08* || (20) *.08* || ϕ *.12*
brown	(20) *.4* || (18) *.1* || (20) *.32* || (18) *.08* || (20) *.08* || (18) *.02*
gray	(20) *.4* || (20) *.1* || ϕ *.5*

Fig. 3. Full-table and grouped exhaustive aggregates

AVG, MIN, or MAX is NULL. The confidence associated with alternative a_j is the probability of instance I_j. Although we are not addressing lineage in this paper, here we can easily see that lineage of each alternative is the set of alternatives in its corresponding possible-instance.

In our running example, the TriQL query to find the average length of observed squirrels looks just like its SQL counterpart:

```
SELECT AVG(length) as avgLength FROM Sightings
```

The answer returned is in the first table of Figure 3. This query result has one alternative for each of the six possible-instances in Figure 2. Since the number of possible-instances doubles with each additional alternative in the input relation, the result of this *exhaustive* aggregation is exponential in the size of the input relation. Even if we merge duplicate alternatives (using Trio's MERGED option [32]), aggregations like SUM could still be exponential. Hence, we define a set of practical variants for aggregation over uncertain data.

A. Practical Variants to Exhaustive Aggregation

We provide three variants for each of the five aggregate functions. (Thus, including exhaustive aggregation, Trio supports a total of 20 aggregate functions.) We define the variants in terms of the result of the corresponding exhaustive aggregate function, which we denote $\mathcal{A}$.

Let us begin with the *low* variant. A *low* aggregate returns one x-tuple having one alternative that is the lowest non-NULL alternative in $\mathcal{A}$. Note that there can be at most one NULL in $\mathcal{A}$, corresponding to the empty possible-instance, and it can only occur for aggregates SUM, AVG, MIN, and MAX; on the empty possible-instance COUNT returns 0. If the only alternative in $\mathcal{A}$ is

NULL, then *low* returns NULL as well.[2] TriQL supports five *low* aggregate functions: LCOUNT, LSUM, LAVG, LMIN, and LMAX. For example, we can retrieve the lowest possible average squirrel length with the TriQL query SELECT LAVG(length) FROM Sightings. The result is a single x-tuple having one alternative with value 18.

Similarly, TriQL currently supports *high* aggregate functions HCOUNT, HSUM, HAVG, HMIN, and HMAX, returning the highest possible aggregate value. In our running example, if we replace LAVG in the query above with HAVG, we get one x-tuple having one alternative with value 19.33.

The third variant is the set of *expected* aggregate functions: ECOUNT, ESUM, EAVG, EMIN, and EMAX. An *expected* aggregate is the weighted average of all the non-NULL alternatives in the corresponding exhaustive result $\mathcal{A}$, where the weights are based on the alternative's confidence values as defined earlier. (Recall that confidences on result alternatives correspond to probabilities on possible-instances.) One subtlety is that we ignore the empty possible-instance for all expected aggregates, except for ECOUNT. For example, for ESUM we compute the weighted average of all possible sums, whenever a sum exists. (To do so, we must scale the confidence values of the non-NULL alternatives in $\mathcal{A}$ so they add up to 1, but implementing this computation turns out to be easy; see Section IV.) If we factored in the empty possible-instance, we would need to either use 0 for its SUM, which might nonintuitively result in LSUM $>$ ESUM, or use NULL, which cannot be combined arithmetically with other values. On the other hand, for ECOUNT we do consider the empty possible-instance, since it meaningfully contributes a count of 0 to the expected result. Note that *expected* aggregate functions typically return a value that is not an alternative of the exhaustive result $\mathcal{A}$.

Considering our running example once again, the expected average length of observed squirrels can be computed from the exhaustive avgLength result shown earlier. We get:

$$19.33 \cdot 0.4 + 18.67 \cdot 0.1 + 19.33 \cdot 0.32 + 18.67 \cdot 0.08 + 19 \cdot 0.08 + 18 \cdot 0.02 = 19.16$$

Thus, the query SELECT EAVG(length) FROM Sightings returns one x-tuple having one alternative with value 19.16. Note the following two points about our new aggregate functions:

- In all of the variants, the result for full-table aggregation has exactly one alternative. We set the confidence of that alternative to 1.0.

[2] In general, we follow the SQL semantics for NULLs, and aggregates on the empty possible-instance return NULL when the corresponding SQL aggregate returns NULL on an empty table.

- For all five aggregate functions applied to any data, the variants satisfy the desirable constraint *low* $\leq$ *expected* $\leq$ *high*.

Generally speaking, low, high and expected aggregates provide the user information about the distribution of the alternative values in the exhaustive result. More information could be gleaned from variance and higher moments. In this paper, we have not considered these additional measures. Reference [7] provides some results for computing the variance of `SUM` and `COUNT` aggregates, but they do not consider issues arising from the empty possible-instance. Their technique generalizes our method for computing `ECOUNT` and should be easily adaptable. Histograms also provide the user rich information about the distribution of the result. We have seen that our techniques can be extended to compute histograms of MIN (and MAX). In future work, we will consider computation of variance (and higher moments) and histograms for all aggregates.

B. Grouped Aggregation

Now consider queries with `GROUP BY` clauses. We first briefly review grouped aggregation queries in SQL [15]. Then we define their meaning over uncertain relations, again following TriQL semantics. Finally we extend our *low*, *high*, and *expected* variants for grouped aggregation.

In SQL, the `GROUP BY` clause contains a list of *grouping attributes*. Call this list G, and without loss of generality assume it is a single attribute. The result of a query with grouped aggregation has one tuple for each value g of G appearing in the query result prior to aggregation. Let A be the query's aggregate function. (Again without loss of generality, assume there is only one aggregate function in the `SELECT` clause, along with the grouping attribute.) The tuple in the query result for value $G = g$ contains the result of aggregate function A applied to $\sigma_{G=g}(R)$.

Now consider an uncertain relation U with possible-instances $I_1, \ldots I_n$. The result of a grouped aggregation query Q, as usual, must represent the possible answers $Q(I_1), \ldots, Q(I_n)$. Trio produces one x-tuple for each value g of G that appears in at least one possible-instance. Logically, this x-tuple contains the result of full-table aggregation on the uncertain relation $\sigma_{G=g}(U)$, followed by a correction for the empty possible-instance: As noted earlier, full-table aggregation produces a 0 alternative (for `COUNT`) or a `NULL` alternative (for the other aggregate functions) if the input relation has $\{\}$ as one of its possible-instances. In grouped aggregation, when this alternative would be produced by full-table aggregation over $\sigma_{G=g}(U)$, it is replaced

by the special ϕ alternative, representing the fact that a group for $G = g$ does not appear in all possible-instances. In the full grouped aggregation result containing one x-tuple for each possible grouping value g, correct possible-instance semantics across tuples relies on lineage. The confidence for an alternative, as usual, represents the probability of the corresponding possible-instance(s).

In our running example, to find the average length for each squirrel color, we write:

```
SELECT color, AVG(length) AS avgLength FROM Sightings GROUP BY color
```

The result is shown in the second table of Figure 3. Notice that in the result, `color` is a certain attribute. For each group, there is one alternative for each possible-instance in which the group exists. For example, "black" has 4 alternatives since it appears in 4 of the 6 possible-instances, while "brown" has 6 alternatives since it appears in all 6 of the possible-instances. When a group does not appear in all possible-instances, it also has a ϕ alternative.

Note that with Trio's `MERGED` option, we get the more readable result:

color	avgLength
black	(18) *.4* || (19) *.32* || (20) *.16* || ϕ *.12*
brown	(20) *.8* || (18) *.2*
gray	(20) *.5* || ϕ *0.5*

C. Variants for Grouped Aggregation

The *low* and *high* aggregates in a grouped aggregation are the obvious extension of the full-table case. Suppose we want low and high bounds for average squirrel lengths based on color. The TriQL query is `SELECT color, LAVG(length) AS lAvgLength, HAVG(length) as hAvgLength FROM Sightings GROUP BY color` . The result is shown in Figure 4.
Grouped aggregation queries with *expected* aggregates are also an extension of the full-table case: For every group, *expected* returns the weighted-by-confidences average of the non-ϕ alternatives in its corresponding exhaustive answer, after scaling the confidences of the non-ϕ alternatives so that they add up to 1.0. Note that here, unlike in the full-table case, for `ECOUNT` we do not factor in a count of 0 for possible-instances in which a group does not exist. As with the other aggregate functions and to correctly follow TriQL possible-instance semantics, a ϕ alternative appears in the exhaustive `COUNT` result, not a 0, when a group does not appear in some possible-instances.

color	lowAvgLength	highAvgLength
black	18	20
brown	18	20
gray	20	20

Fig. 4. Result of Low and High queries

color	expectedAvgLength
black	18.73
brown	19.6
gray	20

Fig. 5. Result of expected queries

In our running example, the expected average length of squirrels based on color is given by `SELECT color, EAVG(length) AS expectedAvgLength FROM Sightings` and the result is shown in Figure 5.

IV. Implementation

We have implemented all 20 aggregate functions in the Trio system: *exhaustive*, *low*, *high*, and *expected* for each of `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`. Each function is supported in both full-table and grouped form—typically the implementation for grouped is a fairly direct extension of the full-table version. We first briefly review how the Trio system is built on top of a conventional relational DBMS; for details see [19]. We then describe how the encoding we use for uncertain relations facilitates simple query translation and stored procedures to efficiently compute all but one (`EAVG`) of the aggregate functions.

Consider a Trio relation `T(`$A_1,\ldots,A_n$`)`. Relation `T` is stored in a conventional relational table with four additional attributes: `T_enc(xid, aid, conf, certain, `$A_1,\ldots,A_n$`)`. Each alternative of each x-tuple in `T` is stored as its own tuple in `T_enc`. The additional attributes in `T_enc` are as follows:

- `xid` identifies the x-tuple
- `aid` identifies an alternative within the x-tuple
- `conf` contains the confidence of the alternative
- `certain` is a flag to indicate whether the x-tuple has a ϕ alternative

For example, the Trio relation `Sightings(time, color, length)` from Figure 1 is encoded in the regular relational table `Sightings_enc(xid, aid, conf, certain, time, color, length)`

xid	aid	conf	certain	time	color	length
101	1	0.5	0	1	gray	20
101	2	0.4	0	1	black	20
102	3	0.8	1	2	black	18
102	4	0.2	1	2	brown	16
103	5	1.0	1	2	brown	20

Fig. 6. `Sightings_enc` is the Trio system encoding for the relation in Figure 1

shown in Figure 6. [3]

Note that ϕ alternatives are not explicitly represented. They are encoded in the `certain` attribute of each tuple: `certain=0` if there is a ϕ alternative and `certain=1` if not. Implicitly, if there is a ϕ alternative, its confidence is $1 - \Sigma$, where Σ is the sum of the confidences of the other alternatives. For example, x-tuple 101, which corresponds to the first observation in the `Sightings` table, has two tuples in `Sightings_enc`, one for each non-ϕ alternative, and `certain` is 0 since the non-ϕ confidences do not add up to 1.

Trio translates TriQL queries over uncertain relations into SQL queries over this encoding. For example, the simple TriQL query:

```
SELECT color FROM Sightings S WHERE length > 18 AND conf(S) > 0.4
```

translates nearly verbatim to a query over `Sightings_enc`, although some bookkeeping is required to construct result x-tuples and manage lineage information.

Our presentation in this paper covers queries with just one aggregation in their `SELECT` clause. The current implementation does support multiple aggregations in a single query, by creating a translated subquery for each one and joining the results. In theory, the optimizer can produce an execution plan that shares work across the multiple subqueries, but we are likely to get better performance if we do more sophisticated translation of queries with multiple aggregations ourselves, a topic of future work.

Support for `HAVING` clauses can be added by first creating the translated subquery Q for the

[3] In reality, Trio implements the table in Figure 6 as a virtual view over two tables, `Sightings_c` and `Sightings_u`, joined on `xid`: Table `Sightings_c` has one row per x-tuple, containing the certain attributes and the special column `certain`. Table `Sightings_u` has one row per alternative, containing the uncertain attributes and the special column `conf`.

TABLE I

SUMMARY OF IMPLEMENTATION METHODS FOR THE 20 AGGREGATE FUNCTIONS

	COUNT	SUM	AVG	MIN/MAX
exhaustive	stored procedure	stored procedure	stored procedure	stored procedure
low/high	translation	translation	stored procedure	translation
expected	translation	translation	approximation	stored procedure

query without the HAVING clause and then filtering the results using the HAVING predicate. This method works (both semantically and in implementation) for all 20 aggregate variants discussed in this paper. Details are omitted.

Table I summarizes the methods we used to implement the 20 different aggregate functions. In the table we have combined *low* and *high*, as well as MIN and MAX, since they are always symmetric. Recall that we are considering aggregation queries over one base table, possibly with filtering predicates. A "translation" entry in Table I indicates that we were able to implement that aggregate by a simple translation from TriQL queries to queries on the encoded table; this approach works for 10 of the 20 cases. Except for EAVG, which is approximated, the remaining 9 were implemented using efficient stored procedures. (Since Trio is currently built on top of the *Postgres* DBMS, we use PL/pgSQL [20] and SPI [29] for stored procedures.)

For all 20 aggregate functions, the full-table and grouped versions are implemented in a similar fashion. In general we focus our discussion on the full-table version, although we do illustrate some grouped cases. Note that in no cases do we actually perform the $\sigma_{G=g}(R)$ selections used in the definition of grouped aggregation.

The remainder of this section first discusses exhaustive aggregation (Section IV-A). We then cover the aggregates implemented through translation (Section IV-B), and finally the remaining aggregates implemented as stored procedures (Section IV-C). Aggregate EAVG is an important special case—an efficient algorithm to compute the exact EAVG remains an open problem. There have been several proposals for computing approximate answers, e.g., [3, 13]. In Trio, we approximate EAVG as ESUM divided by ECOUNT, with compensation for the empty possible-instance. Section V discusses this approximation.

A. Exhaustive Aggregates

In this section we describe our implementation of the exhaustive aggregates. As seen in the first row of Table I, all of them are implemented using stored procedures. As shown in [6], SUM and AVG can have different values in each possible instance and therefore have an exponentially-sized result. Aggregates COUNT, MIN, and MAX have a polynomial number of different values, although the number may still be too high to be usable in practice.

We use algorithms similar to the ones described in [6] to implement COUNT, MIN, and MAX, and will not describe them further. We did implement the exponential algorithms for SUM and AVG, primarily for experimental purposes—they usually cannot be used except for very small relations. Our "user-friendly" query processor first counts the number of possible-instances, which can be done very efficiently. If the number of possible-instances is too high (say $> 2^{32}$), we return an error message and do not permit exhaustive SUM and AVG in these cases.

When the possible-instances are few enough that we permit exhaustive SUM and AVG, our algorithm first computes and materializes, into a temporary table, all possible combinations of the alternatives in the input relation (identified by their aid's), i.e., it enumerates all possible-instances. This table is then joined back with the input table to fetch and aggregate the actual data. For example, the stored procedure to compute exhaustive full-table AVG on Sightings first populates temporary table tmp_combos_Sightings with the possible combinations of alternatives (each with a combo_id), then computes the following query:[4]

```
SELECT AVG(length) AS avgLength, PRODUCT(trio_conf) AS conf
FROM Sightings S INNER JOIN tmp_combos_Sightings T USING (aid)
GROUP BY combo_id
```

This query returns AVG values for all possible-instances along with their corresponding confidences. Simple post-processing generates the single x-tuple comprising the final result. The grouped version (by color for example) is similar to the query above, except the GROUP BY also includes the grouping attributes, and there is a final ORDER BY on the grouping attributes:

```
SELECT color, AVG(length) AS avgLength, PRODUCT(trio_conf) AS conf
FROM Sightings S INNER JOIN tmp_combos_Sightings T USING (aid)
GROUP BY combo_id, color ORDER BY color
```

[4]Aggregate function PRODUCT is similar to SUM, except it multiplies instead of adds.

Post-processing on the result of this query generates one x-tuple for each group.

B. Translation-Based Aggregates

For the aggregates implemented by translation, we automatically rewrite the TriQL query with aggregation into a SQL aggregation query over our encoded data. For these aggregates, the computation is about as efficient as aggregating over a conventional relation. We show translations for a representative sample of full-table and grouped aggregation queries over the `Sightings` relation. Note that `WHERE` and `HAVING` predicates could be added easily (since they are executed before and after the aggregation respectively) to our example queries.

For *low* (respectively *high*) aggregates, all the translations essentially involve choosing the right alternative (or ϕ) from each x-tuple in the table, in order to construct the possible-instance with the lowest (respectively highest) aggregate value across all possible-instances.

For *expected* aggregates, the two translations exploit *linearity of expectation* [23] to compute the contribution of each alternative (based on its value and its confidence) towards the expected aggregate value across all possible-instances.

For each translation, we show the TriQL query and the corresponding translation to a SQL (specifically `postgresql`) query, along with a description in English of what the query is computing. Clearly, all of these translations are much more efficient than computing the corresponding exhaustive aggregate. The most expensive translations perform a `GROUP BY` on `xid` and an `ORDER BY` on the aggregated attribute. So, they have a time complexity of $O(n \log n)$ for n tuples in the relation. The following translations cover the primary techniques; those omitted are similar. In the descriptions, we refer to *certain* x-tuples as those with no ϕ alternatives, and *uncertain* x-tuples as those where ϕ is present.

LCOUNT

- *TriQL:* `SELECT LCOUNT(*) FROM Sightings`
- *Description:* The lowest count occurs when every uncertain x-tuple is not present. Thus, the translated query simply counts the number of certain x-tuples.
- *SQL:* `SELECT COUNT(DISTINCT xid) FROM Sightings_enc WHERE certain = 1`

HCOUNT

- *TriQL:* `SELECT HCOUNT(*) FROM Sightings`

- *Description:* The highest count occurs when all uncertain x-tuples are present. Thus, the translated query simply counts the number of (certain and uncertain) x-tuples.
- *SQL:* `SELECT COUNT(DISTINCT xid) FROM Sightings_enc`

LSUM

- *TriQL:* `SELECT LSUM(length) FROM Sightings`
- *Description:* To compute the lowest SUM we sum the minimum values from all of the certain x-tuples, then further decrease the sum if possible by adding in the lowest negative value from each uncertain x-tuple that has a negative value. In the case where there are no certain x-tuples and no negative values in uncertain tuples, LSUM is the lowest value across all uncertain x-tuples, capturing the possible-instance that has that value only.
- *SQL:*

```
SELECT CASE WHEN lSumPos IS NOT NULL lSumPos ELSE lSumNeg END AS LSUM
FROM (SELECT MIN(minValue) AS lSumNeg,
             SUM(CASE WHEN certain = 1 or minVal < 0 THEN minVal
                      ELSE NULL END) AS lSumPos
      FROM (SELECT certain, MIN(length) AS minVal
            FROM Sightings_enc GROUP BY certain, xid) R) S
```

LMIN

- *TriQL:* `SELECT LMIN(length) FROM Sightings`
- *Description:* The lowest MIN is simply the least value present in the relation, since some possible-instance contains that value. Analogously, HMAX is simply the highest value present in the relation.
- *SQL:* `SELECT MIN(length) AS LMIN FROM Sightings_enc`

LMAX

- *TriQL:* `SELECT LMAX(length) FROM Sightings`
- *Description:* If the relation contains uncertain x-tuples only, then the lowest MAX is the lowest value present in the relation, since there is a possible-instance containing this value only. If the relation contains one or more certain x-tuples, then the lowest MAX is the largest of the minimum values from each certain x-tuple. HMIN is analogous.
- *SQL:*

```
SELECT (CASE WHEN cLmax IS NOT NULL THEN cLmax ELSE uLmax END) AS LMAX
```

```
FROM (SELECT MAX(certainMinAttr) AS cLmax, MIN(uncertainMinAttr) AS uLmax
      FROM (SELECT MIN(length) AS uncertainMinAttr,
                   MIN(CASE WHEN certain = 1 THEN length
                            ELSE NULL END) AS certainMinAttr
            FROM Sightings_enc GROUP BY xid) R) S
```

Grouped HMAX

- *TriQL:* `SELECT color, HMAX(length) FROM Sightings GROUP BY color`
- *Description:* Recall that without grouping, `HMAX` is simply the highest value present in the relation. As a natural extension, grouped `HMAX` simply selects the highest value present in each group.
- *SQL:* `SELECT color, MAX(length) AS HMAX FROM Sightings_enc GROUP BY color`

Grouped HSUM

- *TriQL:* `SELECT color, HSUM(length) FROM Sightings GROUP BY color`
- *Description:* First consider `HSUM` without grouping. Analogous to `LSUM`, to compute the highest `SUM` we sum the maximum values from all of the certain x-tuples, then further increase the sum if possible by adding in the highest positive value from each uncertain x-tuple that has a positive value. In the case where there are no certain x-tuples and no positive values in uncertain tuples, `HSUM` is the highest value across all uncertain x-tuples, capturing the possible-instance that has that value only. Grouped `HSUM` performs the same computation on a per-group basis. Note that a result x-tuple is certain only if its grouping value appears in a certain x-tuple with only one alternative.
- *SQL:*

```
SELECT color, CASE WHEN hSumPos IS NULL THEN hSumNeg ELSE hSumPos END AS HSUM
FROM (SELECT color, MAX(maxValue) AS hSumNeg,
             SUM(CASE WHEN certain = 1 OR maxValue > 0 THEN maxValue
                      ELSE NULL END) AS hSumPos
      FROM (SELECT color, MAX(length) AS maxValue,
                   (CASE WHEN COUNT(*)>1 THEN 0 ELSE certain END) AS certain
            FROM Sightings_enc GROUP BY color, certain, xid) R
      GROUP BY color) S
```

ECOUNT

- *TriQL:* `SELECT ECOUNT(*) FROM Sightings`

- *Description:* Recall from Section III-A that the expected count is the weighted sum of counts across all possible-instances, where weights are the probabilities of the corresponding possible-instances. In each x-tuple, the confidence value of an alternative is equal to the sum of the probabilities of the possible-instances containing that alternative [2]. Using this property and *linearity of expectation* [23] (See Appendix for a detailed analysis), we get the expected count by summing all non-ϕ confidence values in the entire relation (Recall that ϕ alternatives are not explicitly represented in the underlying database — rather they are encoded in attribute `certain`).
- *SQL:* `SELECT SUM(conf) AS ECOUNT FROM Sightings_enc`

ESUM

- *TriQL:* `SELECT ESUM(length) FROM Sightings`
- *Description:* Analogous to `ECOUNT`, we can compute the expected sum by adding up all non-ϕ alternatives, weighted by their confidence values. However, recall from Section III-A that we do not want to factor the empty possible-instance into the expected sum — without compensation we would be treating the empty possible-instance as having a sum of 0. We compensate by scaling the weighted sum according to the probability of the empty possible-instance, computed as the product of the ϕ confidence values in the uncertain x-tuples.
- *SQL:*

```
SELECT SUM(weightedlength)/(1-PRODUCT(phiconf)) AS ESUM
FROM (SELECT SUM(length* conf) AS weightedlength, 1-SUM(conf) AS phiconf
      FROM Sightings_enc GROUP BY xid) R
```

C. Stored-Procedure Aggregates

From Table I we see that the remaining implementations to discuss, aside from `EAVG` covered in the next section, are the stored procedures for `LAVG`, `HAVG`, `EMIN`, and `EMAX`, in their full-table and grouped versions.

Full-table LAVG (and HAVG)

The pseudocode for the `LAVG` stored procedure is shown in Figure 7. The algorithm essentially constructs the possible-instance that is guaranteed to have the least AVG value. We use an incremental method to construct this possible instance. We first compute the "certain" average, i.e., the average of the minimum values from each certain x-tuple (query `Q1` in Figure 7). This

```
// Query to compute the sum and count of the certain x-tuples
Q1 ← SELECT SUM(mVal) AS mSum, COUNT(*) AS mCount FROM (SELECT MIN(length) AS
mVal FROM Sightings_enc WHERE certain = 1 GROUP BY xid) R
(mSum, mCount) ← result(Q1)
// Query to retrieve the minimum alternative values from each uncertain x-tuple.
// Sort those values in non-decreasing order.
Q2 ← SELECT MIN(length) AS mVal FROM Sightings_enc WHERE certain = 0 GROUP BY
xid ORDER BY mVal
// Include alternatives that decrease the average
foreach mVal in result(Q2)
  if (mSum + mVal) / (mCount+1) < mSum / mCount
    mCount ← mCount + 1
    mSum ← mSum + mVal
  else
    break
  endif
endfor
LAVG ← mSum / mCount
```

Fig. 7. Pseudocode for LAVG stored procedure

value provides an upper bound on the low average. Then, from the uncertain x-tuples, we identify alternatives that can decrease the average by considering the minimum alternative from each x-tuple. We consider these values in ascending order (query Q2 in Figure 7) so we can stop when the average stops decreasing. The procedure for HAVG is nearly identical—it looks for alternatives that increase the average, in descending order.

Full-table EMIN (and EMAX)

The pseudocode for the EMIN stored procedure is shown in Figure 8.[5] For details on the correctness of the algorithm, see Appendix C. We consider all alternative values for the aggregated attribute, in ascending order, until we have “covered” at least one x-tuple that does not have a

[5]The pseudocode is slightly simplified for presentation—the complete version has additional bookkeeping to handle duplicate values.

```
Q ← SELECT length AS val, conf, xid FROM Sightings_enc ORDER BY length
cprob ← 1; cval ← 0; cumulative_conf ← []; found_certain_tuple ← FALSE;
foreach (val, conf, xid) in result(Q) // begin main loop
  if cprob < 1
    cprob ← cprob / (1 − cumulative_conf[xid])
  endif
  cval ← cval + cprob·conf·val
  cumulative_conf[xid] ← cumulative_conf[xid] + conf
  if (cumulative_conf[xid] = 1)
    found_certain_tuple ← TRUE
    break
  endif
  cprob ← cprob · (1 − cumulative_conf[xid])
endfor // end main loop
// Compute probability of empty possible-instance
if (found_certain_tuple = FALSE)
  prob_empty ← 1
  foreach (conf) in cumulative_conf
   prob_empty ← prob_empty · (1 − conf)
  endfor
  // Scale cval to compensate for empty possible-instance
  cval ← cval / (1 - prob_empty)
endif
EMIN ← cval
```

Fig. 8. Pseudocode for EMIN stored procedure

ϕ alternative, or we have considered all values. This stopping condition guarantees that at least one of the considered values exists in every possible-instance, i.e., no other values can be MIN in any possible-instance.

For each considered value v, we compute the total probability of the possible-instances that contain any alternative with value v, but no lower value. We compute the weighted average of the considered values to get our expected MIN, as per the *law of total expectation* [23]. Note that if no certain x-tuple was found, there is an empty possible-instance and we need to compensate for it.

In the pseudocode in Figure 8, at the end of the main loop, the variable *cval* contains the expected MIN value, not yet scaled to compensate for the empty possible-instance (if one exists). Variable *cumulative_conf* is an array that accumulates the non-ϕ confidences of all the considered x-tuples. At the end of the loop, if there is an empty possible-instance, this array is used to compute the probability of the empty possible-instance as the product of the confidence of ϕ in each x-tuple. *cval* is then scaled accordingly.

As can be seen in Figure 8, this stored procedure can be implemented with a linear scan over the result of an ORDER BY query, with extra space required to accumulate confidences until a "covering" x-tuple is found. Note that in the worst case, we require O(n) extra space for n x-tuples. The procedure for EMAX is nearly identical—it enumerates the alternative values in descending instead of ascending order.

Grouped aggregation queries with LAVG, HAVG, EMIN, and EMAX are implemented using very similar algorithms, except the computation is performed per-group instead of across the entire table.

Histograms

As an extension to our implementation, we are working on efficient algorithms for computing histograms of aggregate values. For example, an algorithm similar to Figure 8 can be used to compute the histogram for MIN. We first compute HMIN and LMIN (efficiently) to establish the range of the histogram. Based on available memory, we divide the range into subranges and create an array, *hist*, with one element per subrange. This array is used to accumulate counts for the histogram. Then, in the main loop of Figure 8, instead of updating *cval*, we increment counts corresponding to the subrange that encloses *val*. At the end of the main loop, *hist* contains the histogram for MIN. A similar algorithm can be used to compute the histogram for MAX. Here

TABLE II

TREATMENT OF EMPTY POSSIBLE-INSTANCE AND UNCERTAIN GROUPS

	COUNT	SUM, AVG, MIN, MAX	ECOUNT	ESUM, EAVG, EMIN, EMAX
all-table	empty possible-instance creates 0 alternative	empty possible-instance creates NULL alternative	incorporates 0 alternative from *exhaustive*	does not incorporate NULL alternative from *exhaustive*
grouped	group not in all possible-instances: ϕ alternative		does not incorporate ϕ alternative from *exhaustive*	

too, grouped aggregation queries with histograms can be implemented with similar algorithms. Computing histograms for other aggregates is trickier and will be considered in future work.

V. EXPECTED-AVERAGE

The only aggregate remaining to be discussed is *expected-average* (EAVG). Expected-average over uncertain data is known to be difficult to compute; [3] and [13] provide approximate algorithms. Like [3], we decided to use a fairly simple efficient approximation based on ESUM divided by ECOUNT. In some cases, discussed below, the error is guaranteed to be zero.

A subtle issue arises while computing EAVG because of the way we treat empty possible-instances for the different aggregates in their all-table and grouped forms. Table II summarizes this treatment, based on the definitions and justification in Section III-B. Specifically, since ESUM and ECOUNT are treated the same way for grouped aggregates, we can simply use ESUM/ECOUNT as our approximation. However, for all-table aggregates, ECOUNT incorporates the empty possible-instance, while ESUM does not. Fortunately we can easily compensate by further dividing ECOUNT by the probability of the non-empty possible-instances, which, recall from Section IV, can be computed efficiently. Hereafter consider all-table aggregation; the generalization to grouped is straightforward.

We consider the relative error between the exact expected-average, denoted *eavg*, and our approximation, denoted *approx*: *relative error* $= \left|\frac{eavg - approx}{eavg}\right|$. We first identify a case in which there is no error: when all x-tuples have the same total confidence, p. Situations where $p = 1$ are common, but $c < 1$ occurs frequently as well. Consider, for example, applications where data is collected from sensors and human observations. A single confidence value may be applied to the entire data set depending on the reliability of the sources. We formalize this case in the theorem below.

Theorem 1. Let U be any uncertain relation such that the confidences of all non-ϕ alternatives in each x-tuple sum to the same value. In this case *approx* = *eavg*. □

Before we provide a proof for the theorem, we need the following two lemmas, whose proofs appear in Appendix A.

Lemma 1. Let U be an uncertain relation. There exists a relation U' corresponding to U such that:

1) Every x-tuple x in U has a corresponding x-tuple x' in U': x' has exactly one non-ϕ alternative whose confidence is the sum of the confidences of the non-ϕ alternatives in U.
2) There are no other x-tuples in U'.
3) $eavg(U) = eavg(U')$
4) $approx(U) = approx(U')$ □

Lemma 2. Let U be an uncertain relation where every x-tuple has exactly one non-ϕ alternative with the same confidence value. Then $approx(U) = eavg(U)$. □

Proof of Theorem 1. The proof follows from the two lemmas above. Let p be the value summed to by the confidences of all non-ϕ alternatives in U, i.e., the confidence of ϕ in all x-tuples is $1-p$. Using Lemma 1 we construct, from U, a corresponding relation U' in which each x-tuple has only one non-ϕ alternative with confidence p. Lemma 1 tells us that U' has the same *eavg* and *approx* as U. Lemma 2 considers any relation, like U', in which all x-tuples have exactly one non-ϕ alternative with the same confidence value. It shows that such relations have no error in *approx*, so *approx*(U') = *eavg*(U'). From Lemma 1, *eavg*(U') = *eavg*(U), and *approx*(U') = *approx*(U), proving the theorem. Note that the theorem is trivial for the case where all x-tuples have confidences summing to 1.0, i.e., when there are no ϕ alternatives in U. □

When the conditions of Theorem 1 are not met, error in the approximation is certainly possible, as studied in the past [17]). We have performed a number of experiments, discussed in the technical report [18] but not reported here due to space constraints. The results suggest that with reasonable data sizes and across a wide variety of distributions, the error is quite small in practice.

VI. EXPERIMENTS

All 20 aggregate variants over base tables have been implemented in the Trio system [19]. The experiments presented here were conducted with Trio running on an Ubuntu Linux server

machine with 4 2.8GHz processors and 4GB of RAM. We measure the wall clock execution times with a warm cache, i.e., by running a query once and then averaging the times over three subsequent identical executions.

Our uncertain data set is synthesized from the `Lineitems` table of the TPC-H benchmark [31] with scale factor 1. The number of alternatives per x-tuple vary according to the experiment. Each tuple in the `Lineitems` table becomes an alternative with a randomly assigned probability. For each experiment we run the same set of full-table aggregation queries of the form `SELECT trio_agg(quantity) FROM Lineitems_expno`, where `trio_agg` is one of 15 aggregate variants described in this paper, and `Lineitems_expno` is the Trio table generated from the `Lineitems` table for the specific experiment.

The first experiment (Figures 9 and 10) measures running time as the number of x-tuples in the input table increases, while maintaining a constant number of alternatives (five) per x-tuple. Figure 9 measures the five expected aggregates, while Figure 10 measures the low aggregates. (High aggregates are similar to low.) As can be observed, all aggregates scale linearly with increased input size. Recall that `EAVG` is computed from `ESUM` and `ECOUNT`, accounting for the signficantly higher cost.

The second experiment (Figures 11 and 12) measures the running time as the number of alternatives per x-tuple increases, while maintaining a constant table size: the total number of alternatives across all x-tuples is always 100,000. Since ϕ is one of the possible values for an alternative, the probability that an x-tuple has a ϕ increases as we increase the number of alternatives per x-tuple. Thus, increasing alternatives per x-tuple increases uncertainty in both value and presence of x-tuples. From the plots, we see that increasing the uncertainty in the data has little effect on the running time. In fact, several aggregates have higher running times for the lowest values of alternatives-per-x-tuple. This behavior is a result of keeping the overall data size constant, resulting in more x-tuples (hence more `xid`'s) when there are fewer alternatives. Recall that `ESUM` and `LMAX` perform a subquery aggregate grouping on `xid`, while `LCOUNT` performs a distinct count on `xid`.

The third experiment (Figure 13) demonstrates how the low and high aggregates can be useful as indicators of the amount of uncertainty in the data. For this experiment, we maintain a constant number of x-tuples (20,000), and we vary the number of alternatives per x-tuple to increase the uncertainty, as in the previous experiment. As can be observed, the range between the low and

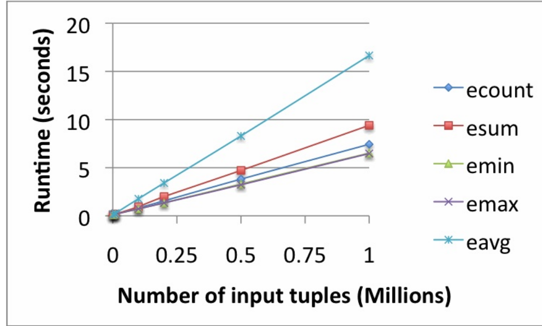

Fig. 9. Scalability in data size – expected aggs

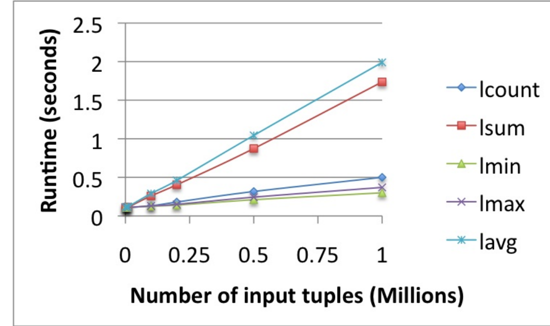

Fig. 10. Scalability in data size – low aggs

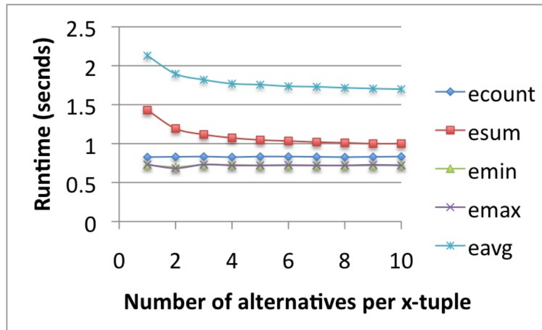

Fig. 11. Scalability in uncertainty – expected aggs

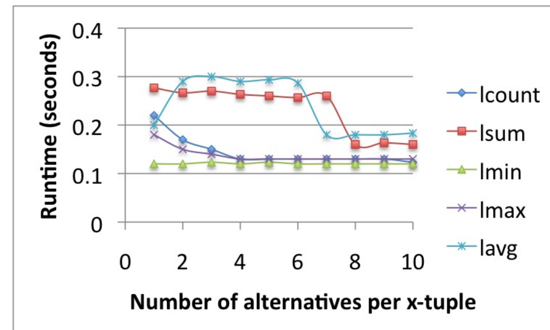

Fig. 12. Scalability in uncertainty – low aggs

high values for SUM increases as the number of alternatives per x-tuple increases.

The fourth experiment (Figure 14) shows the overhead of uncertainty while computing aggregates. For this experiment, we construct a Trio table with 400,000 tuples where each x-tuple has exactly 5 alternatives. We then run all of our full-table aggregates and compare the times against standard SQL aggregates on the underlying relational encoding. As expected, the SQL aggregates are the most efficient. The low (and hence high) aggregates have minimal overhead compared to the SQL aggregates. The expected aggregates are the most expensive since they involve multiplying values with their confidences. EAVG computation involves computation of both ESUM and ECOUNT and hence requires double the time.

We performed some preliminary experiments to evaluate the accuracy of our EAVG approximation, reported in [17]. We found that the accuracy depends on the varying combination of probabilities and alternative values contributing to the average (if Theorem 1 does not hold). Since the results were not especially compelling, we did not include them here. Completely characterizing the error in our approximation is the subject of future work.

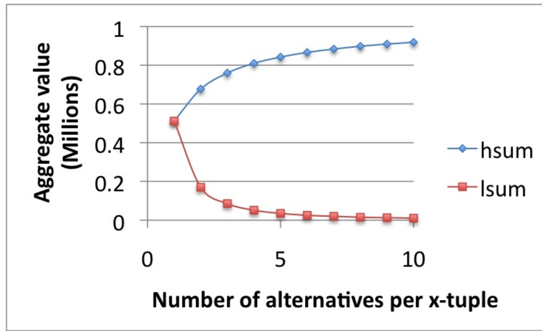

Fig. 13. *low* and *high* as uncertainty increases

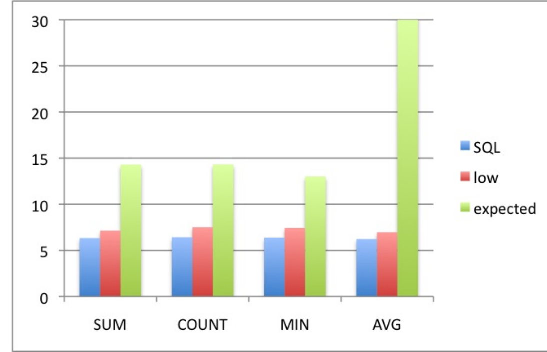

Fig. 14. Comparison of SQL Aggregates with Trio Variants

ID	$R(A)$
1	`(1)` *.5* ∥ ϕ *.5*
2	`(2)` *1.0*

ID	$S(B)$
11	`(1)` *1.0*
12	`(1)` *1.0*
13	`(1)` *.6* ∥ `(2)` *.4*

ID	$J(A, B)$
101	`(1,1)` *.5* ∥ ϕ *.5*
102	`(1,1)` *.5* ∥ ϕ *.5*
103	`(1,1)` *.3* ∥ ϕ *.7*
104	`(2,2)` *.4* ∥ ϕ *.6*

Fig. 15. Joins in Trio

VII. Aggregations over Joins

In this section we address only the case of aggregations over equijoins on base tables (i.e., the joined tables do not have lineage), as this case alone poses several challenges.

The primary reason it is difficult to aggregate over the result J of a join is that tuples in J may no longer be independent: a tuple j_1 might be present if and only if another tuple j_2 is present; conversely j_1 and j_2 may be mutually exclusive even when not alternatives of the same tuple. Such *correlations* restrict the possible-instances and therefore affect aggregation results. (Correlations arise more generally in data with lineage, posing similar problems for aggregation.)

Consider as an example $R(A)$, $S(B)$, and $J(A, B) = R \bowtie_{R.A=S.B} S$ in Figure 15. Let (i, n) denote the nth alternative of the tuple with ID i. Tuple (101,1) is derived from tuples (1,1) and (11,1), and tuple (102,1) is derived from tuples (1,1) and (12,1). Thus, both (101,1) and (102,1) exist if (1,1) exists, otherwise neither (101,1) and (102,2) exist, i.e., these two tuples in J are "mutually inclusive." Conversely, tuples (103,1) and (104,1) are mutually exclusive since they are derived from (13,1) and (13,2), which themselves are mutually exclusive as alternatives of the same tuple. In Trio query results, such correlations are captured using lineage [2].

To begin exploring aggregations over joins, we implemented all 20 aggregate variants over

base-table joins using the brute-force approach: Enumerate all possible-instances of the base tables and join them, producing the correct possible-instances of the join result (including correlations). Then compute the aggregate over the possible-instances of the join. Of course this approach is only feasible for very small databases.

Our next step was to handle cases where the correlations induced by joins do not affect the aggregation result. In practical terms, this means we can simply use our techniques from previous sections to aggregate join result J as if it were a base relation. This approach works, and is very efficient, for two important cases: *expected-count*, and *expected-sum* when J does not permit the empty possible-instance. *Expected-average* can then be approximated efficiently just as in Section V; the join does not introduce additional considerations in the approximation.

For all other aggregate variants, correlations can affect aggregations over J. An important avenue of future work is to develop efficient algorithms for aggregation not only in the presence of correlations induced by joins, but also in the presence of arbitrary lineage.

Reference [7] provides a generalization of our method for ECOUNT over one base table to compute the ECOUNT over the result of joining an arbitrary number of tables. We can use those techniques to Trio. However, [7] does not handle the case of an empty possible-instance. Recall from Section IV that for ESUM we must compensate using the probability P_ϕ of the empty possible-instance. We cannot compute P_ϕ correctly for the result of a join without considering correlations. However, the guarantee of at least one tuple in any possible-instance of the join result (which is trivial to check) ensures $P_\phi = 0$, and the same technique for ECOUNT works for ESUM.

Computing all aggregates (not just SUM and COUNT) over join results, while handling issues arising from the empty possible-instance is part of future work.

VIII. Related Work

Non-aggregation queries on probabilistic and uncertain databases have been studied extensively for several years, e.g., [1, 2, 4, 8, 10, 11, 30]. Aggregation queries have been studied more recently, e.g., [3, 6, 7, 13, 16, 22, 24, 25, 33]. Reference [3] considers computing expected aggregates for hierarchical domains, using an approximation similar to the one we have chosen for EAVG. Reference [6] gives algorithms for exhaustive aggregation over uncertain data without confidences. It also shows that COUNT, MIN, and MAX have polynomial-sized results whereas

SUM and AVG have exponentially-sized results. The work is extended in [33] to uncertain data with probabilities. Aggregations in the setting of probabilistic XML have also been studied recently [9]. Reference [7] provides methods to compute expectation and variance of SUM and COUNT aggregates.

Recent work in the area of approximate algorithms for expected aggregates includes [13, 14] which use a probabilistic stream model and provide generating-function based algorithms to compute very close approximations. However, they require multiple passes over the data, and may not improve much upon the error of our approximation in practical applications. Reference [12] uses a Monte Carlo approach to compute approximate query answers. Instead of using multiple passes over the data, [12] generates multiple instances of the same tuple based on the number of required Monte Carlo iterations. References [16, 25] are about resolving inconsistencies in the data using aggregations, rather than about the aggregate computation itself. Reference [22] uses a linear programming approach to compute aggregations and reference [24] uses a fuzzy-set approach. Top-k query processing for uncertain data also has been studied previously [21, 28].

None of the previous work we are aware of considers a full suite of aggregate variants for uncertain and probabilistic databases, including low, high, and expected for all five aggregate functions in their full-table and grouped forms. Reference [27] uses a simpler setting of "entity-based business intelligence" on inconsistent databases. They return the range of values for the aggregates similar to running low and high aggregate queries in Trio.

IX. Conclusions and Future Work

We introduced aggregate function variants for uncertain data that are much more efficient to compute than exact (exhaustive) aggregates, and are likely to be more practical from a usability perspective. Based on the data encoding scheme in the Trio system, we have implemented many of the aggregate variants, in their full-table and grouped forms over base tables, through simple query translation. The remaining aggregates are implemented as stored procedures, but they are still quite efficient—generally relying on one or two aggregate queries over regular relations, with an additional order-by in the worst case. For the one problematic aggregate, *expected-average*, we compute an approximation based on *expected-sum* over *expected-count*. We have found that the error is low in general, and is guaranteed to be zero in certain cases. Our implementation is available both in the Trio online demo and the open-source distribution. We have demonstrated

via experiments that our translation and stored-procedure based implementations are: (1) scalable in data size; and (2) not affected by the amount of uncertainty in the data. We also showed how *low* and *high* aggregates are useful as indicators of uncertainty in a dataset. Finally, we provided preliminary findings on aggregations over joins of base tables.

This paper covers a significant first step in making aggregation work in the Trio system. There are many important avenues of future work:

- We have observed through experiments that the error tends to be low in our approximation of `EAVG`; we would like to characterize the error exactly. Similar approximations have been used elsewhere, so we may be able to adapt existing results.
- `DISTINCT` aggregates are nontrivial to add, although we do not expect major obstacles.
- Much more challenging is handling aggregation queries involving Trio tables with lineage. Arbitrary non-independence among data values, as introduced by lineage, complicates aggregation considerably. Ideas from [7, 26] should be helpful in getting started.
- We have concentrated on the low, high, and expected values of aggregates. It may be useful to provide more information, such as like variance and histograms as discussed in Sections III and IV.

References

[1] D. Barbara, H. Garcia-Molina, and D. Porter. The Management of Probabilistic Data. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):487–502, 1992.

[2] O. Benjelloun, A. Das Sarma, A. Y. Halevy, and J. Widom. ULDBs: Databases with Uncertainty and Lineage. In *Proc. of Intl. Conference on Very Large Databases (VLDB)*, pages 953–964, Seoul, Korea, 2006.

[3] D. Burdick, P. Deshpande, T. S. Jayram, R. Ramakrishnan, and S. Vaithyanathan. OLAP Over Uncertain and Imprecise Data. In *Proc. of Intl. Conference on Very Large Databases (VLDB)*, pages 970–981, Trondhiem, Norway, 2005.

[4] R. Cavallo and M. Pittarelli. The Theory of Probabilistic Databases. In *Proc. of Intl. Conference on Very Large Databases (VLDB)*, pages 71–81, Brighton, England, 1987.

[5] Christmas Bird Count. Home Page at `http://www.audubon.org/bird/cbc`.

[6] A. L. P. Chen, J. Chiu, and F. S. C. Tseng. Evaluating Aggregate Operations Over Imprecise Data. *IEEE Transactions on Knowledge and Data Engineering*, 08(2):273–284, 1996.

[7] L. Chen and A. Dobra. Efficient Processing of Aggregates in Probabilistic Databases. Technical Report REP-2008-454, University of Florida, 2008.

[8] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating Probabilistic Queries Over Imprecise Data. In *Proc. of ACM-SIGMOD International Conference on Management of Data*, pages 551–562, San Diego, California, 2003.

[9] S. Cohen, B. Kimelfeld, and Y. Sagiv. Incorporating Constraints in Probabilistic XML. In *Proc. of ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 109–118, Vancouver, Canada, 2008.

[10] N. N. Dalvi and D. Suciu. Efficient Query Evaluation on Probabilistic Databases. In *Proc. of Intl. Conference on Very Large Databases (VLDB)*, pages 864–875, Toronto, Canada, 2004.

[11] D. Dey and S. Sarkar. A Probabilistic Relational Model and Algebra. *ACM Transactions on Database Systems*, 21(3):339–369, 1996.

[12] R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, and P. Haas. MCDB: A Monte Carlo Approach to Managing Uncertain Data. In *Proc. of ACM-SIGMOD International Conference on Management of Data*, pages 687–700, 2008.

[13] T. S. Jayram, S. Kale, and E. Vee. Efficient Aggregation Algorithms for Probabilistic Data. In *Proc. of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 896–905, 2007.

[14] T. S. Jayram, A. McGregor, S. Muthukrishnan, and E. Vee. Estimating Statistical Aggregates on Probabilistic Data Streams. In *Proc. of ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 243–252, Beijing, China, 2007.

[15] A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *Journal of the ACM*, 29(3):699–717, 1982.

[16] S. McClean, B. Scotney, and M. Shapcott. Aggregation of Imprecise and Uncertain Information in Databases. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):902–912, 2001.

[17] R. Murthy and J. Widom. Making Aggregation Work in Uncertain and Probabilistic Databases. In *Proc. of Workshop on Management of Uncertain Data at VLDB*, pages 76–90, Vienna, Austria, 2007.

[18] R. Murthy and J. Widom. Making Aggregation Work in Uncertain and Probabilistic Databases. Technical report, Stanford InfoLab, May 2007. Available on

http://dbpubs.stanford.edu.

[19] M. Mutsuzaki, M. Theobald, A. de Keijzer, J. Widom, P. Agrawal, O. Benjelloun, A. Das Sarma, R. Murthy, and T. Sugihara. Trio-One: Layering Uncertainty and Lineage on a Conventional DBMS (Demo). In *Proc. of Conference on Innovative Data Systems Research (CIDR)*, pages 269–274, Pacific Grove, California, 2007.

[20] PL/pgSQL - SQL Procedural Language. Manual at `http://www.postgresql.org/docs/7.4/ interactive/plpgsql.html`.

[21] C. Re, N. Dalvi, and D. Suciu. Efficient Top-k Query Evaluation on Probabilistic Data. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, pages 886–895, Istanbul, Turkey, 2007.

[22] R. Ross, V. S. Subrahmanian, and J. Grant. Aggregate Operators in Probabilistic Databases. *J. ACM*, 52(1):54–101, 2005.

[23] S. Ross. *A First Course in Probability*. Pearson Publications, 7th edition, May 2005.

[24] E. A. Rundensteiner and L. Bic. Evaluating Aggregates in Possibilistic Relational Databases. *Data Knowl. Eng.*, 7(3):239–267, 1992.

[25] B. Scotney and S. McClean. Database Aggregation of Imprecise and Uncertain Evidence. *Inf. Sci. Inf. Comput. Sci.*, 155(3-4):245–263, 2003.

[26] P. Sen and A. Deshpande. Representing and Querying Correlated Tuples in Probabilistic Databases. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, Apr. 2007.

[27] Y. Sismanis, L. Wang, A. Fuxman, P. J. Haas, and B. Reinwald. Resolution-Aware Query Answering for Business Intelligence. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, 2009.

[28] M. A. Soliman, I. F. IlyasY, and K. C.-C. Chang. Top-k Query Processing in Uncertain Databases. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, pages 896–905, Istanbul, Turkey, 2007.

[29] Server Programming Interface. Documentation at `http://www.postgresql.org/docs/8.1/ interactive/spi.html`.

[30] D. Suciu and N. N. Dalvi. Foundations of Probabilistic Answers to Queries. In *Proc. of ACM-SIGMOD International Conference on Management of Data*, pages 963–963, Baltimore, Maryland, 2005. ACM Press.

[31] Transaction Processing Council (TPC). TPC Benchmark H: Standard Specication, 2006.

Available from http://www.tpc.org/tpch.

[32] Trio Online Resources: TriQL Language Language Manual, Online Demo and Open-Source distribution. Available from http://www.infolab.stanford.edu/trio.

[33] F. S. C. Tseng, A. L. P. Chen, and W.-P. Yang. Answering Heterogeneous Database Queries with Degrees of Uncertainty. *Distributed and Parallel Databases*, 1(3):281–302, 1993.

[34] J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *Proc. of Conference on Innovative Data Systems Research (CIDR)*, pages 262–276, Pacific Grove, California, 2005.

APPENDIX

A. Proofs of the Lemmas

Proof of Lemma 1: Let U have N possible-instances. Each possible-instance i with probability c_i has COUNT k_i, SUM s_i, and AVG a_i where $a_i = \frac{s_i}{k_i}$. c_ϕ is the probability of the empty possible-instance. The table below shows the exhaustive aggregate values for SUM, COUNT, and AVG.

COUNT	$(\mathtt{k_1})\ c_1 \ \|\| \ \ldots \ \|\| \ (\mathtt{k_N})\ c_N \ \|\| \ (0)\ c_\phi$
SUM	$(\mathtt{s_1})\ c_1 \ \|\| \ \ldots \ \|\| \ (\mathtt{s_N})\ c_N \ \|\| \ \phi\ c_\phi$
AVG	$(\mathtt{a_1})\ c_1 \ \|\| \ \ldots \ \|\| \ (\mathtt{a_N})\ c_N \ \|\| \ \phi\ c_\phi$

Our definitions of *esum*, *ecount*, and *eavg* are based on the exhaustive aggregate result (refer to Section III-A) as follows:

$$esum = \frac{1}{1-c_\phi} \cdot \sum_{i=1}^{N} s_i \cdot c_i \quad (1)$$

$$ecount = \sum_{i=1}^{N} k_i \cdot c_i \quad (2)$$

$$eavg = \frac{1}{1-c_\phi} \cdot \sum_{i=1}^{N} \frac{s_i}{k_i} \cdot c_i \quad (3)$$

Our approximation for *eavg* (refer to Section V) is:

$$approx = \frac{esum}{ecount \cdot \frac{1}{1-c_\phi}} = \frac{esum}{ecount} \cdot (1-c_\phi) \quad (4)$$

Now, consider any x-tuple x in U. Let x have $h+1$ alternatives, $\mathtt{v_i}$ ($1 \leq \mathtt{i} \leq h$) with confidence p_i and the ϕ alternative with confidence p_ϕ. We show that our definitions of *eavg* and *approx*

over U do not change when all non-ϕ alternatives of x are replaced with a single alternative whose confidence $p' = \sum_{i=1}^{h} p_i = 1 - p_\phi$ and value $v' = \frac{1}{p'} \cdot \sum_{i=1}^{h} v_i \cdot p_i$.

We first partition the N possible-instances into $G = \frac{N}{h+1}$ groups of $h+1$ instances each. Note N must have $h+1$ as a factor. Each group fixes one alternative from each x- tuple in U other than x, and includes all possible alternatives of x (including ϕ). Now, consider Equation 3 for *eavg* and rewrite it by grouping together terms in the summation based on these groups.

$$eavg = \frac{1}{1-c_\phi} \cdot \sum_{j=1}^{G} \left[\frac{s_{j_\phi}}{k_{j_\phi}} \cdot c_{j_\phi} + \sum_{i=1}^{h} \frac{s_{j_i}}{k_{j_i}} \cdot c_{j_i} \right] \tag{5}$$

Each c_{j_i} is the product of the confidences of the tuples in possible-instance j_i in group j. This product can be written as the product π_j of confidences of the tuples fixed in group j times the confidence p_i of the alternative from x in possible-instance j_i. So, $c_{j_i} = \pi_j \cdot p_i$. The COUNT for each possible-instance j_i in group j is the same: $k_{j_i} = k_j$. The SUM in each possible-instance j_i in group j is the sum of the fixed tuples (σ_j) plus the alternative v_i from x: $s_{j_i} = \sigma_j + v_i$.

We now rewrite Equation 5 using the formulae above.

$$eavg = \frac{1}{1-c_\phi} \cdot \sum_{j=1}^{G} \left[\frac{s_{j_\phi}}{k_{j_\phi}} \cdot c_{j_\phi} + \sum_{i=1}^{h} \frac{\sigma_j + v_i}{k_j} \cdot \pi_j \cdot p_i \right]$$

The inner summation, Σ, can be rewriten as:

$$\Sigma \;=\; \frac{\pi_j}{k_j} \cdot \sum_{i=1}^{h} \sigma_j \cdot p_i + v_i \cdot p_i = \left(\sum_{i=1}^{h} p_i \right) \cdot \left(\sigma_j + \frac{\sum_{i=1}^{h} v_i \cdot p_i}{\sum_{i=1}^{h} p_i} \right) = \frac{(\sigma_j + v'_j)}{k_j} \cdot \pi_j \cdot p'$$

So, *eavg* is rewritten as follows:

$$\frac{1}{1-c_\phi} \cdot \sum_{j=1}^{G} \frac{s_{j_\phi}}{k_{j_\phi}} \cdot c_{j_\phi} + \frac{(\sigma_j + v'_j)}{k_j} \cdot \pi_j \cdot p' \tag{6}$$

We perform this process for every x-tuple in U, which results in the term replacement of Equation 6 for each one. We then have 2^n terms in the *eavg* summation. These 2^n terms correspond to the possible-instances of the relation U' where every x-tuple from U is replaced as above. Thus $eavg(U) = eavg(U')$. Using a very similar construction, we can show $esum(U) = esum(U')$ and $ecount(U) = ecount(U')$. Therefore $approx(U) = approx(U')$. Note that if there is no empty possible-instance, p_ϕ is zero, so c_ϕ is zero and all formulae still hold. □

Proof of Lemma 2: Let U have n x-tuples, and let p be the confidence value of the non-ϕ

alternatives in U. Each x-tuple x_i $(1 \leq i \leq n)$ has a single alternative v_i with confidence p. Applying linearity of expectation on Equations 1–4 as in Appendix B, we get:

$$esum = \frac{1}{1-c_\phi} \cdot p \cdot \sum_{i=1}^{n} v_i; ecount = p \cdot n; approx = \frac{esum}{ecount} \cdot (1-c_\phi) = \frac{1}{n}\sum_{i=1}^{n} v_i \qquad (7)$$

Let W be the set of non-empty possible-instances, W_k be the set of possible-instances with k tuples, and K be the number of tuples in a random possible-instance. Using an indicator variable I_{ij} as in Appendix B:

$$(1-c_\phi)eavg = \sum_{i \in W} \frac{s_i}{k_i} c_i = \sum_{i \in W} \left(\frac{\sum_{j=1}^{n} v_j I_{ij}}{\sum_{j=1}^{n} I_{ij}} \right) c_i = \sum_{k=1}^{n} \sum_{i \in W_k} \frac{1}{k} \sum_{j=1}^{n} v_j I_{ij} c_i = \sum_{j=1}^{N} v_j \sum_{k=1}^{n} \frac{1}{k} \sum_{i \in W_k} I_{ij} c_i$$

$$= \sum_{j=1}^{n} v_j \sum_{k=1}^{n} \frac{1}{k} E[I(x_j \neq \phi, K=k)] = \frac{1}{n} \sum_{j=1}^{n} v_j \sum_{k=1}^{n} \frac{n}{k} \binom{n-1}{k-1} p^k (1-p)^{n-k} = (1-c_\phi) \frac{1}{n} \sum_{j=1}^{n} v_j$$

Dividing each side by $(1-c_\phi)$ completes the proof. □

B. *Applying Linearity of Expectation to compute* ESUM *and* ECOUNT

Let U be an uncertain relation with n tuples, $v_i, \cdots, v_n$, with probabilities $p_1, \cdots, p_n$ respectively, and N possible-instances. Consider a possible-instance R_i of U that has r_i of the n tuples and probability P_i. The count, C_i of R_i can be written as $C_i = \sum_{k=1}^{r_i} 1$. Using an indicator variable I_{ij} that is 1 if tuple $v_j \neq \phi$ in possible-instance R_i and 0 otherwise, the sum can be rewritten as $C_i = \sum_{j=1}^{n} I_{ij}$. From our definition of ECOUNT:

$ecount = \sum_{i=1}^{N} P_i C_i = \sum_{i=1}^{N} P_i \sum_{j=1}^{n} I_{ij} = \sum_{j=1}^{n} \sum_{i=1}^{N} P_i I_{ij} = \sum_{j=1}^{n} E[I(v_j \neq \phi)] = \sum_{j=1}^{n} p_j$

where I(A) is the indicator of event A. A similar proof exists for ESUM. □

C. *Applying Law of Total Expectation to compute* EMIN

First consider S, an uncertain relation with exactly one non-ϕ alternative per x-tuple. Let the non-ϕ alternatives have values $v_1, \cdots, v_n$ and probabilities $p_1, \cdots p_n$ respectively. Now, in each possible-instance of U (except the empty possible-instance), one of the alternatives will be the minimum. The probability, m_i, of a given alternative, v_i, being the minimum is the sum of the probabilities of the possible-instances which contain v_i and no other alternatives v_j that are less than v_i. Without loss of generality, let $v_1 \leq v_2 \leq \cdots \leq v_n$. Since each of the v_i's are independent, $m_1{=}p_1$, $m_2{=}(1-p_1)p_2$, $m_3{=}(1-p_1)(1-p_2)p_3$ and so on. Generalizing,

$$m_k = \left(\prod_{j=1}^{k-1} (1-p_j) \right) p_k = \left(\prod_{j=1}^{k-2} (1-p_j) \right) (1-p_{k-1}) p_k = m_{k-1}(1-p_{k-1})p_k \qquad (8)$$

Applying the law of total expectation, the expected minimum across all possible-instances can now be computed as the total expectation of each of the alternatives being the minimum:

$$emin = \sum_{i=1}^{n} m_i v_i = m_1 v_1 + \sum_{i=2}^{n} m_{i-1}(1 - p_{i-1}) p_i v_i \tag{9}$$

This summation can be computed easily in an incremental manner by looping over the values in non-descending order. Note that if one of the v_i's has a probability $p_i = 1$, then for all $j > i$, $m_j = 0$. So, we can break out of our loop when we encounter a value with probability 1.

If x-tuples have more than one non-ϕ alternative, the v_i's are not all independent. So, in our loop, we need to accumulate probabilities of values which belong to the same x-tuple and use the cumulative probabilities to compensate for values already seen from the same x-tuple.

D. Formula for LAVG *computation*

Note that the LAVG value appears as the AVG in at least one possible-instance. So, in order to compute the LAVG directly from the Trio relation, we incrementally "construct" the possible-instance that has the lowest AVG. First consider S with one non-ϕ alternative per x-tuple. WLOG, let S have m certain tuples (i.e., confidence = 1) with values $u_1, \cdots, u_m$ and n uncertain tuples (i.e., with confidence < 1) $v_1, \cdots, v_n$. Also, let $v_1 \leq v_2 \leq \cdots \leq v_n$. Our aim is to construct the possible-instance with the least AVG. Note that all certain tuples exist in every possible-instance. So, we first compute the sum of the certain tuples as the base sum, $S_0 = \sum_{i=1}^{m} u_i$ and store the base count $N_0 = m$. S_0/N_0 is then the AVG of the possible-instance which has only the certain tuples. We now iterate over the uncertain values in non-decreasing order, each time including the value in the running sum, S_i, only if it decreases the running average. For $i = 1, \cdots n$, $S_i = S_{i-1} + v_i$ and $N_i = N_{i-1} + 1$ if $\frac{S_i}{N_{i-1}+1} < \frac{S_{i-1}}{N_{i-1}}$; otherwise, $S_i = S_{i-1}$ and $N_i = N_{i-1}$. At the end of this loop, S_n and N_n contain the sum and count of the possible-instance with the lowest possible AVG and hence LAVG = S_n/N_n. Note that if for some k, $S_k = S_{k-1}$, we can stop the iteration at k. No subsequent value, v_j, $j > k$, will decrease the running average since $v_j \leq v_i$ for $i = k + 1, \cdots, n$. This also means that if v_k is included in the sum, then each of $v_1, \cdots v_{k_1}$, should also be included in order to minimize the average.

The same technique works for uncertain relations by just choosing the lowest value alternative from each x-tuple as the v_i's.

X. Author Biographies

A. Raghotham Murthy

Raghotham Murthy is a PhD student in Computer Science at Stanford University working with Professor Jennifer Widom. His research interests are in provenance and data management. He has worked on large scale data processing systems at Yahoo and Facebook. He received his MS in Computer Science from Stanford University.

B. Robert Ikeda

Robert Ikeda is a PhD student in Computer Science at Stanford University working on the Panda project led by Professor Jennifer Widom. His research interests include data provenance and information extraction. He received a B.S. in Electrical Engineering from UC San Diego.

C. Jennifer Widom

Jennifer Widom is the Fletcher Jones Professor and Chair of the Computer Science Department at Stanford University. She received her Bachelors degree from the Indiana University School of Music in 1982 and her Computer Science Ph.D. from Cornell University in 1987. She was a Research Staff Member at the IBM Almaden Research Center before joining the Stanford faculty in 1993. Her research interests span many aspects of nontraditional data management. She is an ACM Fellow and a member of the National Academy of Engineering and the American Academy of Arts and Sciences; she received the ACM SIGMOD Edgar F. Codd Innovations Award in 2007 and was a Guggenheim Fellow in 2000; she has served on a variety of program committees, advisory boards, and editorial boards.