HAL
open science

SUM Query Processing over Probabilistic Data

Reza Akbarinia, Patrick Valduriez, Guillaume Verger

To cite this version:

Reza Akbarinia, Patrick Valduriez, Guillaume Verger. SUM Query Processing over Probabilistic Data. [Research Report] RR-7629, INRIA. 2011, pp.21. inria-00596020

HAL Id: inria-00596020
https://inria.hal.science/inria-00596020v1

Submitted on 26 May 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

SUM Query Processing over Probabilistic Data

Reza Akbarinia, Patrick Valduriez, Guillaume Verger

N° 7629

May 2011

Thème : Représentation et traitement des données et des connaissances

Rapport de recherche

ISSN 0249-6399

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche SOPHIA ANTIPOLIS - MÉDITERRANÉE

SUM Query Processing over Probabilistic Data

Reza Akbarinia[1], Patrick Valduriez[1], Guillaume Verger[2]
INRIA and LIRMM, Montpellier, France
[1]FirstName.LastName@inria.fr, [2]Verger@lirmm.fr

Thème : Représentation et traitement des données et des connaissances – Perception, cognition, interaction
Projet Zenith

Rapport de recherche n° 7629 – May 2011 - 21 pages

Abstract: SUM queries are crucial for many applications that need to deal with probabilistic data. In this report, we are interested in the queries, called ALL_SUM, that return all possible sum values and their probabilities. In general, there is no efficient solution for the problem of evaluating ALL_SUM queries. But, for many practical applications, where aggregate values are small integers or real numbers with small precision, it is possible to develop efficient solutions. In this report, based on a recursive approach, we propose a complete solution for this problem. We implemented our solution and conducted an extensive experimental evaluation over synthetic and real-world data sets; the results show its effectiveness.

Keywords: Probabilistic databases, query processing, aggregate queries.

Centre de recherche INRIA Sophia Antipolis - Méditerranée
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Traitement de requêtes SUM sur données probabilistes

Résumé: Les requêtes d'agrégation, notamment SUM, sont cruciales pour de nombreuses applications qui ont besoin de traiter des données incertaines probabilistes. Dans cet article, nous nous intéressons à un type de requête, appelé ALL_SUM, qui a pour objectif de retourner toutes les valeurs de somme possibles et leurs probabilités. En général, il n'y a pas de solution efficace au problème de l'évaluation de ces requêtes. Mais, pour de nombreuses applications pratiques, où les valeurs globales sont de petits entiers ou des nombres réels avec une petite précision, il est possible de développer des solutions efficaces. Dans ce rapport nous proposons une solution complète à ce problème. Nous avons implémenté notre solution et mené une évaluation expérimentale approfondie sur des données réelles et synthétiques; les résultats montrent son efficacité.

Mots clés: Bases de données probabilistes, traitement de requêtes, requêtes d'agrégation.

1 Introduction

Aggregate (or *aggr* for short) queries, in particular SUM queries, are crucial for many applications that need to deal with uncertain data [13][19][27]. Let us give two motivating examples from the medical and environmental domains.

Example 1: Reducing the usage of pesticides. Consider a plant monitoring application on which we are working with scientists in the field of agronomy. The objective is to observe the development of diseases and insect attacks in the agricultural farms by using sensors, aiming at using pesticides only when necessary. Sensors periodically send to a central system their data about different measures such as the plants contamination level (an integer in [0..10]), temperature, moisture level, etc. However, the data sent by sensors are not 100% certain. The main reasons for the uncertainty are the effect of climate events on sensors, e.g. rain, unreliability of the data transmission media, etc. The people from the field of agronomy with which we had discussions use some rules to define a degree of certainty for each received data. A decision support system will analyze the sent data, and trigger a pesticide treatment when the cumulative contamination since the last treatment is higher than a threshold. An important query for the decision support system is "return sum of contamination where date > x".

Example 2: Remote health monitoring. As another example, we can mention a medical center that monitors key biological parameters of remote patients at their homes, e.g. using sensors in their bodies. The sensors periodically send to the center the patients' health data, e.g. blood pressure, hydration levels, thermal signals, etc. For high availability, there are two or more sensors for each biological parameter. However, the data sent by sensors are uncertain, and the sensors that monitor the same parameter may send inconsistent values. There are approaches to estimate a confidence value for the data sent by each sensor, e.g. based on their precision [14]. According to the data sent by the sensors, the medical application computes the number of required human resourses, e.g. nurses, and equipments for each patient. Figure 1 shows an example table of this application. The table shows the number of required nurses for each patient. One important query in this application is "return the sum of required nurses".

Tuple	Patient	Required nurses	…	Probability
t_1	PID1	1	…	0.8
t_2	PID2	0	…	0.4
t_3	PID3	2	…	0.5

Figure 1. Motivating example

Possible Worlds	Prob.	SUM
$w_1=\{t_1,t_2,t_3\}$	0.16	3
$w_2=\{t_1,t_2\}$	0.16	1
$w_3=\{t_1,t_3\}$	0.24	3
$w_4=\{t_2,t_3\}$	0.04	2
$w_5=\{t_1\}$	0.24	1
$w_6=\{t_2\}$	0.04	0
$w_7=\{t_3\}$	0.06	2
$w_8=\{\}$	0.06	0

Figure 2. The possible worlds and the results of SUM query in each world, for the database of Figure 1.

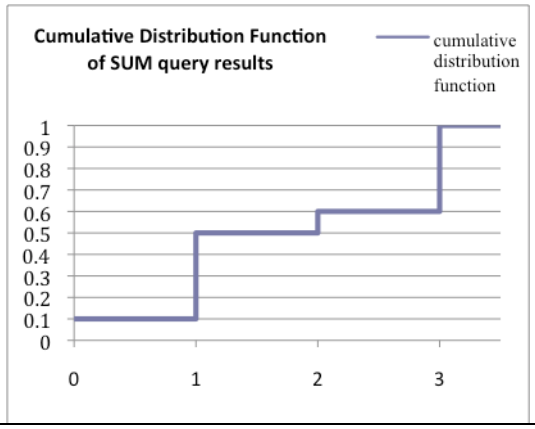

Figure 3. Cummulative disribution function for the SUM query results over database shown in Figure 1.

Based on the data in Figure 1, we show in Figure 2 the possible worlds, i.e. the possible database instances, their probabilities, and the result of the SUM query in each

world. In this example, there are 8 possible worlds and four possible sum values, i.e. 0 to 3.

In this report, we are interested in the queries that return all possible sum values and their probabilities. This kind of query, which we call ALL_SUM, is also known as sum probability distribution. For instance, the result of ALL_SUM (required nurses) for the database shown in Figure 1 is {(3, 0.40), (2, 0.10), (1, 0.40), (0, 0.10)}, i.e. for each possible SUM result, we return the result and the probability of the worlds in which this result appears. For instance, the result sum=3 appears in the worlds w_1 and w_3, so its probability is equal to $P(w_1) + P(w_3) = 0.16 + 0.24 = 0.40$.

By using the results of ALL_SUM, we can generate the probability density and cumulative distribution functions, which are important for many domains, e.g. scientific studies. Figure 3 shows the cumulative distribution function of sum values over the data shown in Figure 1.

A naïve algorithm for evaluating ALL_SUM queries is to enumerate all possible worlds, compute sum in each world, and return the possible sum values and their aggregated probability. However, this algorithm is exponential in the number of uncertain tuples.

In this report, we deal with ALL_SUM queries and propose pseudo-polynomial algorithms that are efficient in many practical applications, e.g. when the aggr attribute values are small integers or real numbers with small precision, i.e. small number of digits after decimal point. These cases cover many practical aggregate attributes, e.g. temperature, blood pressure, needed human recourses per patient in medical applications. To our knowledge, this is the first proposal of an efficient solution for returning the exact results of ALL_SUM queries.

1.1 Contributions

In this report, we propose a complete solution to the problem of evaluating SUM queries over probabilistic data:

- We first propose a new recursive approach for evaluating ALL_SUM queries, where we compute the sum probabilities in a database based on the probabilities in smaller databases.
- Based on this recursive approach, we propose a pseudo-polynomial algorithm, called DP_PSUM that efficiently evaluates ALL_SUM queries for the applications where the aggr attribute values are small integers or real numbers with small precision. For example, in the case of positive integer aggr values, the execution time of DP_PSUM is $O(n^2 \times avg)$ where n is the number of tuples and avg is the average of the aggr values.
- Based on this recursive approach, we propose an algorithm, called Q_PSUM, which is polynomial in the number of SUM results.
- We validated our algorithms through implementation over synthetic and real-world data sets; the results show the effectiveness of our solution.

The rest of the report is organized as follows. In Section 2, we present the probabilistic data models, which we consider and define formally the problem we address. In Sections 3 and 4, we describe our Q_PSUM and DP_PSUM algorithms for evaluating ALL_SUM queries under a classical model. In Sections 5 and 6, we extend our solution for a more complex model with some correlations. We also extend our solution for evaluating COUNT queries in Section 6. In Section 7, we report on our experimental validation over synthetic and real-world data sets. Section 8 discusses related work. Section 9 concludes and gives some directions for future work.

2 Porblem Definition

In this section, we first introduce the probabilistic data models that we consider. Then, we formally define the problem that we address.

2.1 Probabilistic Models

The two main models, which are frequently used in our community, are the tuple-level and attribute-level models [8]. These mod-

els, which we consider in this report, are defined as follows.

Tuple-level model. In this model, each uncertain table T has an attribute that indicates the *membership probability* (also called existence probability) of each tuple in T, i.e. the probability that the tuple appears in a possible world. In this report, the membership probability of a tuple t_i is denoted by $p(t_i)$. Thus, the probability that t_i does not appear in a random possible world is $1- p(t_i)$. The database shown in Figure 4.a is under tuple-level model.

Attribute-level model. In this model, each tuple t_i has at least one uncertain attribute, e.g. α, and the value of α in t_i is chosen by a random variable X. We assume that X has a discrete probability density function (pdf). This is a realistic assumption for many applications [8], e.g. sensor readings [11][21]. The values of α in t_i are m values $v_{i,1}, \ldots, v_{i,m}$ with probabilities $p_{i,1}, \ldots, p_{i,m}$ respectively (see Figure 4.b). Note that for each tuple we may have a different pdf.

The tuples of the database may be independent or correlated. In this report, we first present our algorithms for databases in which tuples are independent. We extend our algorithms for correlated databases in Section 6.1.

Tuple	Probability
t_1	p_1
t_2	p_2
…	…

(a)

Tuple	Possible values of aggr attribute, and their probabilities
t_1	$(v_{1,1}, p_{1,1}), (v_{1,2}, p_{1,2}), \ldots, (v_{1,m1}, p_{1,m1})$
t_2	$(v_{2,1}, p_{2,1}), (v_{2,2}, p_{2,2}), \ldots, (v_{2,m2}, p_{2,m2})$
…	…

(b)

Figure 4. Probabilistic data models; a) Tuple-level; b) Attribute-level model

2.2 Problem Definition

ALL_SUM query is defined as follows.

Definition 1: ALL_SUM query. It returns all possible sum results together with their probability. In other words, for each possible sum value, ALL_SUM returns the cumulative probability of the worlds where the value appears as a result of the query.

Let us now formally define ALL_SUM queries. Let D be a given uncertain database, W the set of its possible worlds, and $P(w)$ the probability of a possible world $w \in W$. Let Q be a given aggr query, f the aggr function stated in Q (i.e. SUM), $f(w)$ the result of executing Q in a world $w \in W$, and $V_{D,f}$ the set of all possible results of executing Q over D, i.e. $V_{D,f} = \{v \mid \exists w \in W \wedge f(w)=v\}$. The cumulative probability of having a value v as the result of Q over D, denoted as $P(v, Q, D)$, is computed as follows:

$$P(v,Q,D) = \sum_{\substack{w \in W \text{ and} \\ f(w)=v}} P(w)$$

Our objective in this report is to return the results of ALL_SUM as follows:

$$ALL_SUM\ (Q, D) = \{(v, p) \mid v \in V_{D,f} \wedge p = P(v, Q, D)\}$$

3 ALL_SUM under Tuple-level Model

In this section, we propose an efficient solution for evaluating ALL_SUM queries. We first propose a new recursive approach for computing the results of ALL_SUM. Then, using the recursive approach we propose our Q_PSUM algorithm.

We assume that the database is under the tuple-level model defined in the previous section. Our solution is extended for the attribute-level model in Section 5. We adapt our solution to process COUNT queries in Section 6.2.

3.1 Recursive approach

We develop a recursive approach that pro-

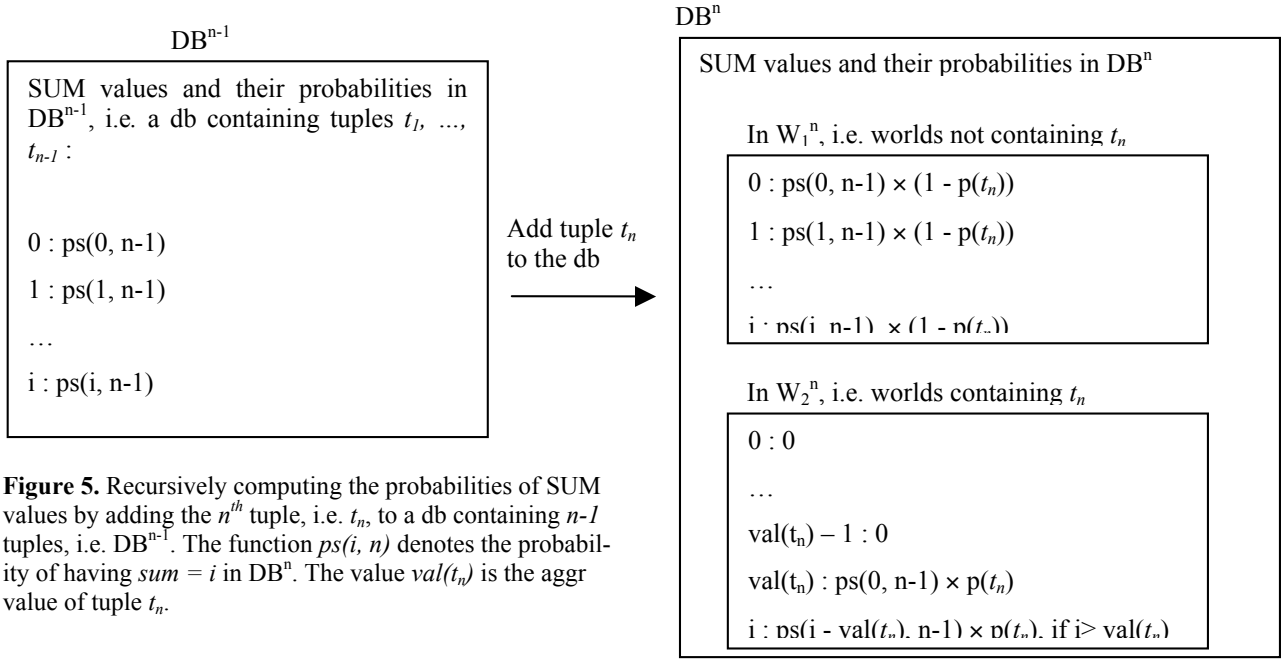

Figure 5. Recursively computing the probabilities of SUM values by adding the nth tuple, i.e. t_n, to a db containing *n-1* tuples, i.e. DB^{n-1}. The function *ps(i, n)* denotes the probability of having *sum = i* in DB^n. The value $val(t_n)$ is the aggr value of tuple t_n.

duces the results of ALL_SUM queries in a database with *n* tuples based on the results in a database with *n-1* tuples. The principle behind it is that the possible worlds of the database with *n* tuples can be constructed by adding / not adding the *n*th tuple to the possible worlds of the database with *n-1* tuples.

Let DB^n be a database involving the tuples $t_1, \ldots, t_n$, and *ps(i,n)* be the probability of having *sum = i* in DB^n, We develop a recursive approach for computing *ps(i, n)*.

3.1.1 Base

Let us first consider the recursion base. Consider DB^1, i.e. the database that involves only tuple t_1. Let $p(t_1)$ be the probability of t_1, and $val(t_1)$ be the value of t_1 in aggr attribute. In DB^1, there are two worlds: 1) $w_1=\{\}$, in which t_1 does not exist, so its probability is $(1- p(t_1))$; 2) $w_2=\{t_1\}$, in which t_1 exists, so the probability is $p(t_1)$. In w_1, we have *sum=0*, and in w_2 we have $sum=val(t_1)$. If $val(t_1) = 0$, then we always have *sum=0* because in both w_1 and w_2, sum is zero. Thus, in DB^1, *ps(i, 1)* can be written as follows:

$$ps(i,1) = \begin{cases} p(t_1) & if\ i = val(t_1)\ and\ val(t_1) \neq 0 \\ 1 - p(t_1) & if\ i = 0\ and\ val(t_1) \neq 0 \\ 1 & if\ i = val(t_1) = 0 \\ 0 & otherwise \end{cases} \quad (1)$$

3.1.2 Recursion Step

Now consider DB^{n-1} , i.e. a database involving the tuples $t_1, \ldots, t_{n-1}$. Let W^{n-1} be the set of possible worlds in DB^{n-1}.

We construct DB^n by adding t_n to DB^{n-1}. Notice that the set of possible worlds in DB^n, denoted by W^n, is constructed by adding / not adding the tuple t_n to each world of W^{n-1}. Thus, in W^n, there are two types of worlds (see Figure 5): 1) the worlds that do not contain t_n, denoted as W^n_1; 2) the worlds that contain t_n, denoted as W^n_2.

For each world $w \in W^n_1$, we have the same world in DB^{n-1}, say w'. Let $p(w)$ and $p(w')$ be the probability of worlds w and w'. The probability of w, i.e. $p(w)$, is equal to $p(w') \times (1 - p(t_n))$, because t_n does not exist in w even though it is involved in the database. Thus, in W^n_1 the sum values are the same as in DB^{n-1}, but the probability of *sum=i* in W^n_1 is equal to the probability of having *sum=i* in DB^{n-1} multiplied by the probability of non-existence of t_n. In other words, we have:

```
Algorithm PS(i, n)
Input:
  n : number of tuples;
  t1, …, tn : the tuples of the database;
  p(t) : a function that returns the probability of tuple t;
Output:
  Probability of sum=i in a database containing t1, …, tn;
Begin
 If (n > 1) then
   Return PS(i, n-1)×(1 – p(tn)) + PS(i - val(tn), n-1)×p(tn);
 Else If ((n=1) and (i=val(t1)) and (val(t1) ≠0)) then
   Return p(t1);
 Else If ((n=1) and (i=0) and (val(t1) ≠0)) then
   Return  1- p(t1);
 Else  If ((n=1) and (i=val(t1)=0)) then
   Return 1;
 Else Return 0;
End;
```

Figure 6. Recursive algorithm for computing the probability of sum=i in a database containing $t_1, \ldots, t_n$.

In W^n_1: *(probability of sum=i) = ps(i, n-1)×(1 – p(t_n))* (2)

Let us now consider W^n_2. The worlds involved in W^n_2 are constructed by adding t_n to each world of DB^{n-1}. Thus, for each sum value equal to i in DB^{n-1} we have a sum value equal to $(i + val(t_n))$ in W^n_2, where $val(t_n)$ is the value of aggr attribute in t_n. Therefore, the probability of $sum= i + val(t_n)$ in W^n_2 is equal to the probability of $sum=i$ in DB^{n-1} multiplied by the existence probability of t_n. In other words, we have:

In W^n_2: *(probability of sum=i) = ps(i - val(t_n), n-1)×p(t_n)* (3)

The probability of $sum=i$ in DB^n is equal to the probability of $sum=i$ in W^n_1 plus the probability of $sum=i$ in W^n_2. Thus, using the Equations 2 and 3, and using the base of the recursion, i.e. Equation 1, we obtain the following recursive formula for the probability of $sum=i$ in DB^n, i.e. $ps(i, n)$:

$$ps(i,n) = \begin{cases} ps(i,n-1)\times(1-p(t_n)) + ps(i-val(t_n),n-1)\times p(t_n) & if\ n>1 \\ 1-p(t_1) & if\ n=1\ and\ i=0\ and\ val(t_1)\neq 0 \\ p(t_1) & if\ n=1\ and\ i=val(t_1)\ and\ val(t_1)\neq 0 \\ 1 & if\ n=1\ and\ i=val(t_1)=0 \\ 0 & otherwise \end{cases} \quad (4)$$

Based on the above recursive formula, we can develop a recursive algorithm for computing the probability of sum=i in a database containing tuples $t_1, \ldots, t_n$ (see the pseudocode in Figure 6). However, the execution time of the algorithm is exponential in the number of uncertain tuples, i.e. due to the two recursive calls in the body of the algorithm.

3.2 Q_PSUM Algorithm

In this section, based on our recursive definition, we propose an algorithm, called Q_PSUM, whose execution time is $O(n \times N)$ where n is the number of uncertain data, and N is the number of distinct sum values.

Q_PSUM uses a list for maintaining the computed possible sum values and their probabilities. It fills the list by starting with DB^1, i.e. a database containing only t_1, and gradually adds other tuples to the database and updates the list.

Let Q be a list of pairs *(i, ps)* such that i is a possible sum value and ps its probability. The Q_PSUM algorithm proceeds as follows (see the pseudocode in Appendix C). It first initializes Q for DB^1 by using the base of the recursive definition. If $val(t_1) = 0$, then it inserts *(0, 1)* into Q, else it inserts $(0, 1 - p(t_1))$ and $(val(t_1), p(t_1))$. By inserting a pair to a list, we mean adding the pair to the end of the list.

Then, in a loop, for $j=2$ to n, the algorithm adds the tuples t_2 to t_n to the database one by one, and updates the list by using two temporary lists Q_1 and Q_2 as follows. For each tuple t_j, Q_PSUM removes the pairs involved in Q one by one from the head of the list, and for each pair $(i, ps)\in Q$, it inserts $(i, ps\times(1 - p(t_j))$ into Q_1 and $(i+val(t_j), ps\times p(t_j))$ into Q_2. Then, it merges the pairs

involved in Q_1 and Q_2, and inserts the merged results into Q.

The merging is done on the sum values of the pairs. That means that for each pair $(i, ps_1) \in Q_1$ if there is a pair $(i, ps_2) \in Q_2$, i.e. with the same sum value, then Q_PSUM removes the pairs from Q_1 and Q_2 and inserts $(i, ps1 + ps2)$ into Q. If there is $(i, ps1) \in Q_1$ but there is no pair $(i, ps2) \in Q_2$, then it simply removes the pair from Q_1 and inserts it to Q.

Let us now analyze the complexity of Q_PSUM. Let N be the number of possible (distinct) sum results. Suppose the lists are implemented using a structure such as linked list (with pointers to the haed and tail of the list). The cost of inserting a pair to the list is $O(1)$, and merging two lists is done in $O(N)$[1]. For each tuple, at most N pairs are inserted to the lists Q_1 and Q_2, and this is done in $O(N)$. The merging is done in $O(N)$. There are n tuples in the database, thus the algorithm is executed in $O(n \times N)$. The space complexity of the algorithm is $O(N)$, i.e. the space needed for the lists.

4 DP_PSUM Algorithm

In this section, using the dynamic programming technique, we propose an efficient algorithm, called DP_PSUM, designed for the applications where aggr values are integer or real numbers with small precisions. It is usually much more efficient than the Q_PSUM algorithm (as shown by the performance evaluation results in Section 4.5).

Let us assume, for the moment, that the values of aggr attribute are positive integer numbers. In Section 4.3, we adapt our algorithm for real numbers with small preci

sions, and in Section 4.4, we deal with negative integer numbers.

[1] Notice that the pairs involved in Q_1 and Q_2 are systematically orderded according to sum values, because they follow the same order as the values in Q which is initially sorted.

4.1 Basic Algorithm

Let *MaxSum* be the maximum possible sum value, e.g. for positive aggr values *MaxSum* is the sum of all values. DP_PSUM uses a 2D matrix, say PS, with $(MaxSum + 1)$ rows and n columns. DP_PSUM is executed on PS, and when it ends, each entry PS[i, j] contains the probability of $sum=i$ in a database involving tuples $t_1, \ldots, t_j$.

DP_PSUM proceeds in two steps as follows (the pseudocode is shown in Appendix C). In the first step, it initializes the first column of the matrix. This column represents the probability of sum values for a database involving only the tuple t_1. DP_PSUM initializes this column using the base of our recursive formula (described in Equation 1) as follows. If $val(t_1) = 0$, then $PS[0, 1] = 1$. Otherwise, $PS[0, 1] = (1 - p(t_1))$ and $PS[val(t_1), 1] = p(t_1)$. The other entries of the first column should be set to zero, i.e. $PS[i, 1] = 0$ if $i \neq 0$ and $i \neq val(t1)$.

In the second step, in a loop, DP_PSUM sets the values of each column j (for $j=2$ to n) by using our recursive definition (i.e. Equation 4) and based on the values in column $j-1$ as follows:

$$PS[i, j] = PS[i, j-1] \times (1 - p(t_j)) + PS[i - val(t_j), j-1] \times p(t_j)$$

Notice that if $(i < val(t_j))$, then for the positive aggr values we have $PS[i - val(t_j), j-1]=0$, i.e. because there is no possible sum value lower than zero. This is why, in the algorithm only if $(i \geq val(t_j))$, we consider $PS[i - val(t_j), j-1] \times p(t_j)$ for computing $PS[i, j]$.

Theorem 1. *DP_PSUM works correctly if the database is under the tuple-level model, and the aggr attribute values are positive integers, and their sum is less than or equal to MaxSum.*

Proof. Implied by using the recursive formula in Equation 4. □

Let us now illustrate DB_PSUM using the following example.

Example 3. Figure 7.b shows the execution of DP_PSUM over the database shown in Figure 7.a. In the first column, we set the probability of sum values for DB[1], i.e. a

database that only involves t_1. Since the aggr value of t_1 is equal to 1 (see Figure 7.a), in DB^1 there are two possible sum values, sum=1 and sum=0 with probabilities 0.3 and (1 – 0.3) = 0.7 respectively. The probabilities in other columns, i.e. in 2nd and 3rd, are computed using our recursive definition. After the execution of the algorithm, the 3rd column involves the probability of sum values in the entire database. If we compute ALL_SUM by enumerating the possible worlds, we obtain the same results.

Tuples	Aggr Attribute Value	Membership Probability
t_1	1	0.3
t_2	3	0.4
t_3	2	0.5

(a

SUM	$DB^1=\{t_1\}$	$DB^2=\{t_1,t_2\}$	$DB^3=\{t_1,t_2,t_3\}$
0	0.7	0.42	0.21
1	0.3	0.18	0.09
2	0	0	0.21
3	0	0.28	0.23
4	0	0.12	0.06
5	0	0	0.14
6	0	0	0.06

(b)

Figure 7. a) A database example in tuple-level model; b) Execution of DP_PSUM algorithm over these example

4.2 Complexity

Let us now discuss the complexity of DP_PSUM. The first step of DP_PSUM is done in *O(MaxSum),* and its second step in $O(n \times MaxSum)$. Thus, the time complexity of DP_PSUM is $O(n \times MaxSum)$, where n is the number of uncertain tuples and *MaxSum* the sum of the aggr values of all tuples. Let *avg* be the average value of aggr values of tuples, then we have $MaxSum = n \times avg$. Thus, the complexity of DP_PSUM is $O(n^2 \times avg)$ where *avg* is the average of the aggr values in the database.

Notice that if *avg* is a small number compared to n, then DP_PSUM is done in a time quadratic to the input. But, if *avg* is exponential in n, then the execution time becomes exponential. Therefore, DP_PSUM is a pseudo polynomial algorithm.

The space requirement of DP_PSUM is equivalent to a matrix of $(MaxSum + 1) \times n$, thus the space complexity is $O(n^2 \times avg)$. In Section 4.2.1, we reduce the space complexity of DP_ PSUM to $O(n \times avg)$.

4.2.1 Reducing space complexity

In the basic algorithm of DP_PSUM, for computing each column of the matrix, we use only the data that are in the precedent column. This observation gave us the idea of using two arrays instead of a matrix for computing ALL_SUM results as follows. We use two arrays of size (*MaxSum* + 1), e.g. ar_1 and ar_2. First, we initialize ar_1 using the recursion base (like the first step of the basic version). Then, for $i = 2, \ldots, n$ steps, DP_PSUM fills ar_2 using the probabilities in ar_1, based on the recursion step, then it copies the data of ar_2 into ar_1, and starts the next step. Instead of copying the data from ar_2 into ar_1, we can simply change the pointers of ar_1 to that of ar_2, and renew the memory of ar_2.

The space requirement of this version of DP_PSUM is *O(MaxSum)* which is equivalent to $O(n \times avg)$ where *avg* is the average value of aggr values.

4.3 Supporting real attribute values with small precisions

In many applications that work with real numbers, the precision of values, i.e. the number of digits after decimal point, is small. For example, in medical applications the temperature of patients requires real values with one digit of precision. DP_PSUM can be adapted to work for those applications as follows. Let *DB* be the input database, and c be the number of precision digits in the aggr values. We generate a new database DB' as follows. For each tuple t in the input database *DB*, we insert a tuple t' to DB' such that the aggr value of t', say v', is equal to the aggr value of t, say v, multiplied by 10^c, i.e. $v' = v \times 10^c$. Now, we are sure that the aggr values in DB' are integer, and we can apply the DP_PSUM algorithm on it.

Then, for each ALL_SUM result (v'_i, p) over DB', we return $(v'_i /10^c, p)$ as a result of ALL_SUM in DB.

The correctness of the above solution can be implied by the fact that, if we multiply all aggr values of a DB by a constant k, then every possible sum result is multiplied by k.

The time complexity of this version of DP_PSUM for aggr attribute values with c digits of precision is $O(n \times MaxSum \times 10^c)$ which is equivalent to $O(n^2 \times avg \times 10^c)$ where avg is the average of the aggr attribute values. The space complexity is $O(n \times avg \times 10^c)$.

4.4 Dealing with negative integer values

The basic version of DP_PSUM assumes integer values that are positive, i.e. ≥ 0. This assumption can be relaxed as follows. Let *MinNeg* be the sum of all negative aggr values. Then, the possible sum values are integer values in [*MinNeg* … *MaxSum*] where *MaxSum* is the maximum possible sum value, i.e. the sum of positive aggr values. This modification in the interval of possible sum values needs the following modifications in the data structure and the algorithm which we used in our DP_PSUM algorithm: 1) the size of the first dimension of the PS matrix should be modified to $(MaxSum + 1 + |MinNeg|)$, instead of $(MaxSum + 1)$, because it represents the number of possible sum values; 2) to cover all possible sum values, in the algorithm (see pseudocode in Appendix C) every where we have a loop "*for i=0 to MaxSum*", we replace it by "*for i=mnNeg to MaxSum*"; 3) Since the index in the matrix cannot be negative, we should shift the index of the first dimension by $|MinNeg|$. This means that everywhere we have $PS[x, j]$, we replace it by $PS[x + |MinNeg|, j]$.

4.5 Leveraging GCD

For the applications with integer aggr values, if the greatest common divisor (GDC) of the values is higher than 1, then we can significantly improve the performance of the DP_PSUM algorithm as follows. Let DB be the given database, and g be the GCD of the aggr values. We generate a new database DB' in which the tuples are the same as in DB except that the aggr values are divided by g. Then, we apply DP_PSUM on DB', and for each sum result (v'_i, p), we return $(v'_i \times g,, p)$ to the user.

In general, the above approach reduces the time and space complexities of DP_PSUM by an order of GCD, i.e. since the average of the aggr values in database DB' is divided by GCD. For example, if the aggr values in DB are {10, 20, 30}, then in the database DB' the aggr values are {1, 2, 3}, i.e. GCD=10. Since the average of aggr values is reduced by 10, the space and time complexity of the DP_SUM algorithm will be reduced by an order of 10.

4.6 Skipping Zero Points

During execution of the basic version of our DP_PSUM algorithm, there are many cells (of the matrix) with zero points, i.e. zero probability values. For example, in the first column of the matrix, there is at most 2 non-zero points, and in the 2nd at most 4, etc. Obviously, we do not need to read zero points because they cannot contribute to non-zero probability values. Thus, we avoid accessing zero points using the following approach. Let L be a list which is initially set to zero. During the execution of the algorithm we add the index of non zero points to L. At each step of the algorithm, for filling each new column, we take into account only the cells whose indices are in L.

This approach improves significantly the performance of DP_PSUM, in particular when the number of tuples is very small compared to the average of aggr values, i.e. avg. For example, if there are only two tuples and avg is equal to 10, then each column of the matrix has about 20 cells. However, there are at most 6 non-zero cells in the matrix, i.e. 2 in the first and at most 4 in the 2nd. Thus, the above approach allows us to ignore almost 70% of the cells.

5 ALL_SUM under Attribute-Level Model

Due to the significant differences between the tuple-level and the attribute-level models

it is not trivial to adapt our yet proposed algorithms for the attribute-level model.

In this section, we propose our solution for computing ALL_SUM results under the attribute-level model.

5.1 Recursive Approach

We propose a recursive approach for computing ALL_SUM under the attribute-level model. This approach is the basis for a dynamic programming algorithm which we describe next. We assume that all tuples have the same number of possible aggr values, say m. This assumption can be easily relaxed as we do in Appendix B. We also assume that, in each tuple t_i, the sum of the probabilities of possible aggr values is 1, i.e. $p_{i,1} + p_{i,2} + ... + p_{i,m} = 1$. In other words, we assume that there is no null value. However, this assumption can be relaxed as in Appendix A.

5.1.1 Recursion Base

Let us consider DB^1, i.e. a database that only involves t_1. Let $v_{1,1}, v_{1,2}, ..., v_{1,m}$ be the possible values for the aggr attribute of t_1, and $p_{1,1}, p_{1,2}, ..., p_{1,m}$ their probabilities, respectively. In this database, the possible sum values are the possible values of t_1. Thus, we have:

$$ps(i,1) = \begin{cases} p_{1,k} & if\ \exists k\ such\ that\ \ v_{1,k} = i \\ 0 & otherwise \end{cases}$$

(5)

5.1.2 Recursion Step

Now consider DB^{n-1}, i.e. a database involving the tuples $t_1, ..., t_{n-}$ Let W^{n-1} be the set of possible worlds for DB^{n-1}. Let $ps(i, n-1)$ be the probability of having $sum=i$ in DB^{n-1}, i.e. the aggregated probability of the DB^{n-1} worlds in which we have $sum=i$. Let $v_{n,1}, .. v_{n,m}$ be the possible values of t_n's aggr attribute, and $p_{n,1}, .. p_{n,m}$ their probabilities. We construct DB^n by adding t_n to DB^{n-1}. The worlds of DB^n are constructed by adding to each $w \in W^{n-1}$, each possible value of t_n. Let $W^n_k \subseteq W^n$ be the set of worlds which are constructed by adding the possible value $v_{n,k}$ of t_n to the worlds of W^{n-1}. Indeed, for each world $w \in W^{n-1}$ there is a corresponding world $w' \in W^n_k$ such that $w' = w + \{t_n\}$ where the possible aggr value of t_n is equal to $v_{n,k}$. This implies that for each possible $sum=i$ with probability p in W^{n-1}, there is a possible $sum = i + v_{n,k}$ with probability $p \times p_{n,k}$ in W^n_k. Recall that $ps(i, n-1)$ is the aggregate probability of the DB^{n-1} worlds in which $sum = i$. Then we have:

$$Probability\ of\ sum=i\ in\ W^n_k = ps(i - v_{n,k}, n-1) \times p_{n,k}; \quad for\ k=1, ..., m$$

(6)

Let $ps(i, n)$ be the probability of $sum=i$ in DB^n. Since we have $W^n = W^n_1 \cup W^n_2 \cup \ldots \cup W^n_m$, the probability of sum=i in W^n is equal to the sum of the probabilities of $sum=i$ in W^n_1, W^n_2, … and W^n_m. Thus, using Equation 6 and the recursion base of Equation 5, the probability of $sum=i$ in DB^n, i.e. $ps(i, n)$, can be stated as follows:

$$ps(i,n) = \begin{cases} \sum_{k=1}^{m} ps(i - v_{n,k}, n-1) \times p_{n,k} & if\ n > 1 \\ p_{1,k} & if\ \ n = 1\ and\ \exists k\ such\ that\ v_{1,k} = i \\ 0 & otherwise \end{cases} \quad (7)$$

5.2 Dynamic Programming Algorithm for Attribute-level Model

Now, we describe a dynamic programming algorithm, called DP_PSUM2, for computing ALL_SUM under the attribute-level model. Here, similar to the basic version of DP_PSUM algorithm, we assume integer values. However, in a way similar tothat of DP_PSUM, this assumption can be relaxed. Let PS be a 2D matrix with (MaxSum + 1) rows and n columns. At the end of DP_PSUM2 execution, PS[i,j] contains the probability of $sum=i$ in a database involving tuples $t_1, ..., t_j$.

DP_PSUM2 works in two steps. In the first step, it initializes the first column of the matrix, by using the base of the recursive definition, as follows. For each possible aggr value of tuple t_1, e.g. $v_{1,k}$, it sets the corresponding entry equal to the probability of the possible value, i.e. it sets $P[v_{1,k}, 1]=p_{1,k}$

for 1≤k≤m.

In the second step, DP_PSUM2 sets the entry values of each column j (for 2≤j≤n) by using the recursion step of the recursive definition, and based on the values yet set in precedent column. Formally, for each column j and row i it sets $PS[i, j] = \sum (PS[i - v_{1,k}, j-1] \times p_{j,k})$ for 2≤k≤m such that $i \geq v_{1,k}$.

Let us now analyze the complexity of the algorithm. Let $avg=MaxSum/n$, i.e. the average of the aggregate attribute values. The space complexity of DP_PSUM2 is exactly the same as that of DP_PSUM, i.e. $O(n^2 \times avg)$. The time complexity of DP_PSUM2 is $O(MaxSum \times n \times m)$. In other words its time complexity is $O(m \times n^2 \times avg)$.

Let us now illustrate the DB_PSUM2 algorithm using an example.

Example 4. Consider the database in Figure 8.a which is under attribute-level model. The execution of the DP_PSUM2 algorithm is shown in Figure 8.b. The first column of the matrix is filled using the probabilities of the possible aggr values of t_1. Thus, we set 0.3 and 0.7 for sum values 1 and 2 respectively. The other columns are filled by using our recursive definition. After the execution of the algorithm, the 3rd column shows the probability of all sum results for our example database.

SUM	$DB^1=\{t_1\}$	$DB^2=\{t_1,t_2\}$	$DB^3=\{t_1,t_2,t_3\}$
0	0	0	0
1	0.3	0	0
2	0.7	0	0
3	0	0.18	0.09
4	0	0.54	0.36
5	0	0.28	0.41
6	0	0	0.14

(b)

Tuples	Aggr attribute values and probabilities
t_1	(1, 0.3), (2, 0.7)
t_2	(3, 0.4), (2, 0.6)
t_3	(0, 0.5), (1, 0.5)

(a)

Figure 8. a) A database example in attribute-level model; b) Execution of DP_PSUM algorithm over these examples

6 Extensions

In this section, we first extend our algorithm to deal with correlated data with mutual exclusions, and then explain how they can be used for computing the results of COUNT aggregate queries.

6.1 ALL_SUM over Correlated Databases

Up to here, we assumed that the tuples of the database are independent. Here, we assume mutual exclusion correlations, and show how to execute over ALL_SUM algorithms over databases that contain such correlation. Two tuples t_1 and t_2 are mutually exclusive, iff they cannot appear together in any instance of the database (i.e. possible world). But, there may be instances in which none of them appear. As an example of mutual exclusive tuples we can mention the tuples which are produced by two sensors that monitor the same object at the same time. In this example, at most one of the produced tuples can be correct, so they are mutually exclusive.

Let us now discuss our approach for evaluating ALL_SUM queries over correlated databases with mutual exclusive dependencies. Our approach is based on the fact that the tuples of a correlated database can be grouped to a set of blocks such that there is no dependency between any two tuples that belong to two different blocks, and there are closure dependencies between any two tuples of each block [10]. Considering such blocks, for the given database D we generate an equivalent database D'. For each independent block b in D, we create a tuple t' in D' with an attribute A. The values of attribute A in t' are the possible values of aggregate attribute in tuples involved in b. The probability of each A's possible value, e.g. v, is equal to the event that v appears as an aggregate attribute value in the block b. The

database D' is under the attribute-level model with no dependency between the tuples, thus we can apply our ALL_SUM algorithms to evaluate ALL_SUM queries over it.

6.2 Evaluating ALL_COUNT Queries Using ALL_SUM Algorithms

We now show how we can evaluate ALL_COUNT queries, i.e. all possible counts and their probabilities, using the algorithms which we proposed for ALL_SUM. Under the attribute-level model, all tuples are assumed to exist, thus the result of a count query is always equal to the number of tuples that satisfy the query. However, under the tuple-level model, the problem of evaluating ALL_COUNT is harder because there may be $(n+1)$ possible count results (i.e. from 0 to n) with different probabilities, where n is the number of uncertain tuples. This is why we deal with ALL_COUNT only under the tuple-level model.

The problem of ALL_COUNT can be reduced to that of ALL_SUM in polynomial time as follows. Let D be the database on which we want to execute ALL_COUNT. We generate a new database D' as follows. For each tuple $t \in D$ we generate a tuple t' in D' such that t' involves only one attribute, e.g. B, with two possible values: $v_1=1$ and $v_2=0$. We set $p(v_1)$ equal to the membership probability of t. We set $p(v_2)= 1 - p(v_1)$. Now, if we apply one of our ALL_SUM algorithms over B as aggr attribute in D', the result is equivalent to applying an ALL_COUNT algorithm over the aggr attribute in D. This is proven by the following theorem.

Theorem 2. *If the database D is under the tuple-level model, then the result of ALL_SUM over the attribute B in database D′ is equivalent to the result of ALL_COUNT over the aggregate attribute of the database D.*

Proof. If the database D is under the tuple-level model, its membership probability in D is equal to the probability of value $v_1=1$ in attribute B of D'. Thus, the contribution of a tuple t to COUNT in the database D is equal to the contribution of its corresponding tuple t' to SUM in the database D'. In other words, the results of ALL_SUM over D' is equivalent to the results of ALL_COUNT over D. □

7 Experimental Validation

To validate our algorithms and investigate the impact of different parameters, we conducted a thorough experimental evaluation. In Section 7.1, we describe our experimental setup, and in Section 7.2, we report on the results of various experiments to evaluate the performance of the algorithms by varying different parameters.

7.1 Experimental Setup

We implemented our DP_PSUM and Q_PSUM algorithms in Java, and we validated them over both real-world and synthetic databases.

As real-world database, like some previous works, e.g. [17][20], we used the data collected in the International Ice Patrol (IIP) Iceberg Sightings Database (http://nsidc.org/data/g00807.html) whose data is about the iceberg evolution sightings in North America. The database contains attributes such as iceberg, number, sighting date, shape of the iceberg, number of days drifted, etc. There is an attribute that shows the confidence level about the source of sighting. In the dataset which we used, i.e. that of 2008, there are 6 confidence levels: R/V (radar and visual), VIS (visual only), RAD (radar only), SAT-LOW (low orbit satellite), SAT-MED (medium orbit satellite) and SAT-HIGH (high orbit satellite). Like in [17] and [20], we quantified these confidence levels by 0.8, 0.7, 0.6, 0.5, 0.4 and 0.3 respectively. As aggr attribute, we used the number of drifted days which contains real numbers with one digit of precision in the interval of [0… 365].

As synthetic data, we generated databases under the attribute-level model which is more complete than the tuple-level model. We generated two types of databases, Uniform and Gaussian, in which the values of attributes in tuples are generated using a

random generator with the uniform and Gaussian distributions, respectively. The default database is Uniform, and the mean (average) of the generated values is 10. Unless otherwise specified, for the Gaussian database the variance is half of the mean. The default number of attribute values in each tuple of our attribute-value model is 2.

In the experiments, we evaluated the performance of our DP_PSUM and Q_PSUM algorithms. We also compared their performance with that of the naïve algorithm that enumerates the possible worlds, computes the sum in each world, and returns the possible sum values and the aggregated probability of the worlds where they appear as the result of sum. To manage the possible sum values, we used a B-tree structure.

For the three algorithms, we measured their response

time. We conducted our experiments on a machine with a 2.66 GHz Intel Core 2 Duo processor and 2GB memory.

7.2 Performance Results

In this section, we report the results of our experiments.

Effect of the number of uncertain tuples. Using the synthetic data, Figure 9 shows the response time of the three algorithms vs. the number of uncertain tuples, i.e. *n*, and the other experimental parameters set as described in Section 7.1. The best algorithm is DP_PSUM, and the worst is the Naïve algorithm. For n>30, the response time of the Naïve algorithm is too long, such that we had to halt it. This is why we do not show its response time for n>30. The response time of DP_PSUM is at least four times lower than that of Q_PSUM (notice that the figure is in logarithmic scale).

Over the IIP database, Figure 10 shows the response time of the three algorithms, with different samples of the IIP database. In

each sample, we picked a set of *n* tuples, from the first to the *n*th tuple of the database. The results are qualitatively in accordance with those over synthetic data.

Effect of data distribution. Figure 11 shows the response time of our algorithms over the Uniform and Gaussian databases. The distribution of aggr values has no impact on the performance of DP_PSUM. But, it has a significant impact on Q_PSUM. The response time of Q_PSUM over the uniform database is more than 3 times better than over Gaussian (note that the curves are in logarithmic scale). The reason is that the distribution of the attribute values affects the number of possible SUM results. In this experiment, the number of tuples was 200.

Effect of average. We performed tests to study the effect of the average of aggregate values in the database, i.e. *avg*, on performance. Using the synthetic data, Figure 12 shows the response time of our DP_PSUM and Q_PSUM algorithms with *avg* increasing up to 50, and the other experimental parameters set as described in Section 7.1. The average of aggregate values has a linear impact on DP_PSUM, and this is in accordance with the complexity analysis done in Section 4. What was not expected is that the impact of *avg* on the performance of Q_PSUM is significant, although *avg* is not a direct parameter in the complexity of Q_PSUM (see Section 3.2). The explanation is that the time complexity of Q_PSUM depends on the number of possible SUM results, and when we increase *avg* (i.e. mean) of the aggregate values, their range become larger, thus the total number of possible sum values increases.

Effect of the number of attribute values per tuple. We tested the effect of the number of attribute values in each tuple under the attribute-level model, i.e. *m*, on performance. Figure 13 shows the response time of our algorithms with increasing *m* up to 10, and other parameters set as in Table 1. This number has a slight impact on DP_PSUM,

but a more considerable impact on Q_PSUM.

Effect of precision. We studied the effect of the precision of real numbers, i.e. the number of digits after decimal point, on the performance of the DP_PSUM algorithm. Using the synthetic data, Figure 14 shows the response time with increasing the precision

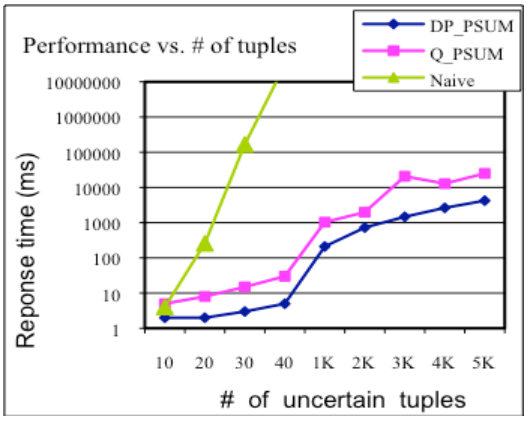

Figure 9. Response time vs. number of uncertain tuples

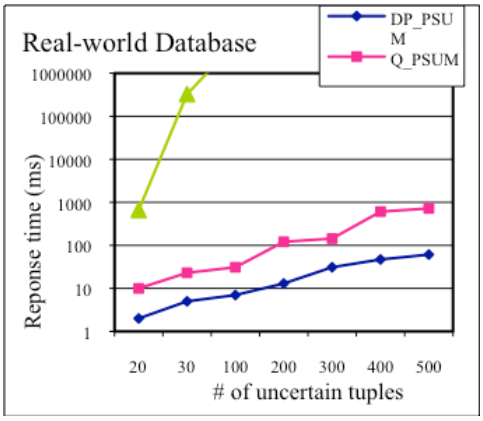

Figure 10. Performance results over real-world database

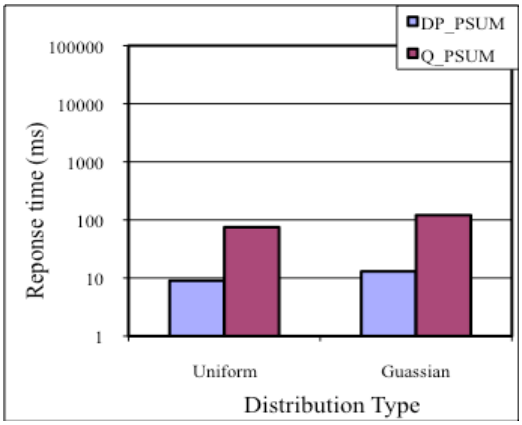

Figure 11. Performance over databases with different distribution types

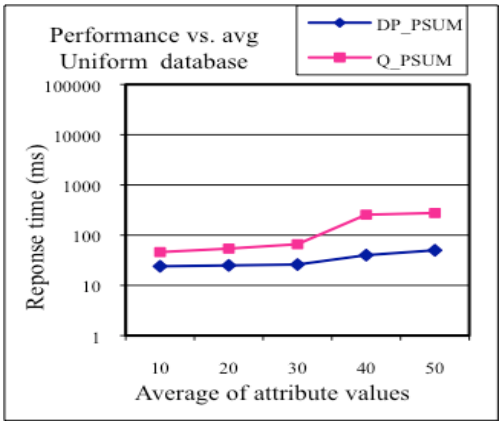

Figure 12. Effect of the average of aggregate attribute values on performance

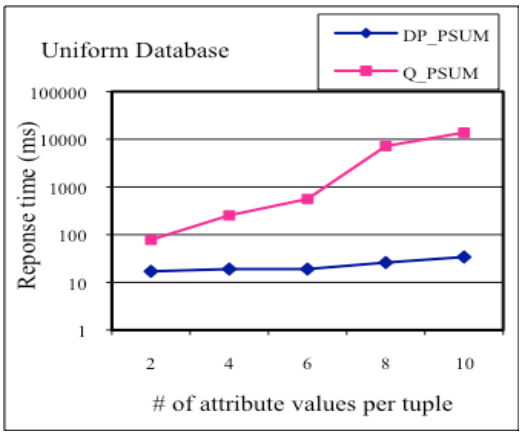

Figure 13. Effect of the number of attribute values per tuple on performance

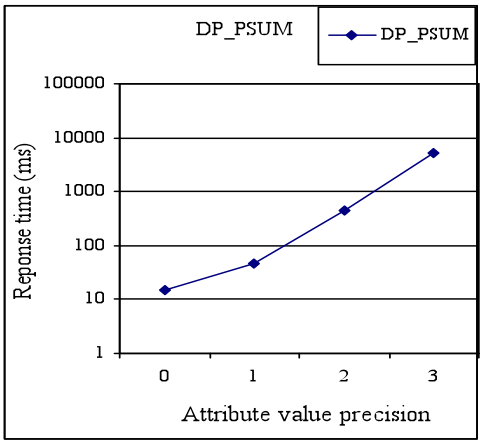

Figure 14. Effect of the precision of real aggr values on performance

of the aggr values. As shown, the precision has a significant impact on the response time of DP_PSUM, i.e. about ten times for each precision digit. This is in accordance with our theoretical analysis done in Section 4, and shows that our algorithm is not appropriate for the applications in which the aggr values are real numbers with many digits after the decimal point.

8 Related Work

In the recent years, we have been witnessing much interest in uncertain data management in many application areas such as data cleaning [2], sensor networks [11][20], information extraction [18], etc. Much research effort has been devoted to several aspects of uncertain data management, including data modeling [3][5][9][15][16][28], skyline queries[4][25], top-k queries [8][12][17][29], nearest neighbor search [30][32][34], spatial queries [33], XML documents [1][23][24], etc.

There has been some work dealing with aggregate query processing over uncertain data. Some of them were devoted to developing efficient algorithms for returning the expected value of aggregate values, e.g. [6][19]. For example in [19], the authors study the problem of computing aggregate operators on probabilistic data in an I/O efficient manner. Under their semantics, the evaluation of SUM queries is not challenging. In [9], Dalvi and Suciu consider both expected value and ALL_SUM, but they only propose an efficient approach for computing the expected value.

Approximate algorithms have been proposed for probabilistic aggregate queries, e.g. [7] and [27]. The Central Limit theorem [26] can be used to approximately estimate the distribution of sums for sufficiently large numbers of probabilistic values. However, in the current report, our objective is to return exact probability values, not approximations.

In [22] and [31], aggregate queries over uncertain data streams have been studied. For example, Kanagal et al. [22] deal with continuous queries over correlated uncertain data streams. They assume correlated data which are Markovian and structured in nature. Their probabilistic model and the assumptions, and as a result the possible algorithms, are very different from ours. For example in their model, they propose algorithms that deal with MIN/MAX queries in a time complexity that does not depend on the number of tuples. However, in our model it is not possible to develop algorithms with such a complexity.

The work in [10] studies the problem of HAVING aggregate queries with predicates. The addressed problem is related to the #KNAPSACK problem which is NP-hard. The difference between HAVING-SUM queries in [10] and our ALL_SUM queries is that in ALL_SUM we return all possible SUM values and their probabilities, but in HAVING-SUM the goal is to check a condition on an aggregate function, e.g. is it possible to have SUM equal to a given value.

Overall, for the problem which we considered in this report, i.e. returning the exact results of ALL_SUM queries, there is no efficient solution in the related work. In this report, we proposed pseudo polynomial algorithms that allow us to efficiently evaluate ALL_SUM queries in many practical cases, e.g. where the aggregate attribute values are small integers, or real numbers with limited precisions.

9 Conclusion

SUM aggr queries are critical for many applications that need to deal with uncertain data. In this report, we addressed the problem of evaluating ALL_SUM queries. After proposing a new recursive approach, we developed an algorithm, called Q_PSUM, which is polynomial in the number of SUM results. Then, we proposed a more efficient algorithm, called DP_PSUM, which is very efficient in the cases where the aggr attribute values are small integers or real numbers with small precision. We validated our algorithms through implementation and experimentation over synthetic and real-world data sets. The results show the effectiveness of our solution. The performance of DP_PSUM is usually better than that of

Q_PSUM. Only over special databases with small numbers of possible sum results and very large aggr value average, Q_PSUM is better than DP_PSUM.

References

[1] S. Abiteboul, B. Kimelfeld, Y. Sagiv and P. Senellart. On the expressiveness of probabilistic XML models. *In VLDB Journal*, 18(5), 2009.

[2] P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases. *In ICDE Conf.*, 2006.

[3] L. Antova, T. Jansen, C. Koch, D. Olteanu. Fast and Simple Relational Processing of Uncertain Data. *In ICDE Conf.*, 2008.

[4] M. J. Atallah and Y. Qi. Computing all skyline probbilities for uncertain data. *In PODS Conf.*, 2009.

[5] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: databases with uncertainty and lineage. *In VLDB Conf.*, 2006.

[6] D. Burdick, P. Deshpande, T.S. Jayram, R. Ramakrishnan, S. Vaithyanathan. OLAP Over Uncertain and Imprecise Data. *In VLDB Conf.*, 2005.

[7] G. Cormode and M. N. Garofalakis. Sketching probabilistic data streams. *In SIGMOD Conf.*, 2007.

[8] G. Cormode, F. Li, K. Yi. Semantics of Ranking Queries for Probabilistic Data and Expected Ranks. *In ICDE Conf.*, 2009.

[9] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *In VLDB Journal*, 16(4), 2007.

[10] C. Ré and D. Suciu. The trichotomy of HAVING queries on a probabilistic database. *In VLDB Journal*, 18(5), 1091-1116, 2009.

[11] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. *In VLDB Conf.*, 2004.

[12] D. Deutch and T. Milo. TOP-K projection queries for probabilistic business processes. *In ICDT Conf.*, 2009.

[13] A. Gal, M.V. Martinez, G.I. Simari and V. Subrahmanian. Aggregate Query Answering under Uncertain Schema Mappings. *In ICDE Conf.*, 2009.

[14] T. Ge, S.B. Zdonik and S. Madden. Top-k queries on uncertain data: on score distribution and typical answers. *In SIGMOD Conf.*, 2009.

[15] T. J. Green and V. Tannen. Models for Incomplete and Probabilistic Information. *In IEEE Data Eng. Bull. 29(1)*, 2006.

[16] R. Gupta, S. Sarawagi. Creating Probabilistic Databases from Information Extraction Models. *In VLDB Conf.*, 2006.

[17] M. Hua, J. Pei, W. Zhang, and X. Lin. Ranking queries on uncertain data: A probabilistic threshold approach. *In SIGMOD Conf.*, 2008.

[18] T. S. Jayram, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar information extraction system. *IEEE Data Eng. Bull.*, 29(1), 2006.

[19] T.S. Jayram, A. McGregor, S. Muthukrishnan, E. Vee. Estimating statistical aggregates on probabilistic data streams. *In PODS Conf.*, 2007.

[20] C. Jin, K. Yi, L. Chen, J. X. Yu, X. Lin. SlidingWindow Topk Queries on Uncertain Streams. *In VLDB Conf.*, 2008.

[21] B. Kanagal and A. Deshpande. Online filtering, smoothing and probabilistic modeling of streaming data. *In ICDE Conf.*, 2008.

[22] B. Kanagal and A. Deshpande. Efficient Query Evaluation over Temporally Correlated Probabilistic Streams. *In ICDE Conf.*, 2009.

[23] B. Kimelfeld, Y. Kosharovsky and Yehoshua Sagiv. Query evaluation over probabilistic XML. *In VLDB Journal*, 18(5), 2009.

[24] A. Nierman, H.V. Jagadish. ProTDB: Probabilistic Data in XML. *In VLDB Conf.*, 2002.

[25] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. *In VLDB Conf.*, 2007.

[26] G. Rempala and J. Wesolowski. Asymptotics of products of sums and U-statistics. *Electronic Communications in Probability,* vol. 7, 47–54, 2002.

[27] R.B. Ross, V.S. Subrahmanian and J. Grant. Aggregate operators in probabilistic databases. *J. ACM*, 52(1). pp. 54-101, 2005.

[28] A. D. Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. *In ICDE Conf.*, 2006.

[29] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k query processing in uncertain databases. *In ICDE Conf.*, 2007.

[30] G. Trajcevski, R. Tamassia, H. Ding, P. Scheuermann, I.F. Cruz. Continuous probabilistic nearest-neighbor queries for uncertain trajectories. *In EDBT Conf.*, 2009.

[31] T. Tran, A. McGregor, Y. Diao, L. Peng, A. Liu. Conditioning and Aggregating Uncertain Data Streams: Going Beyond Expectations. *In VLDB Conf.*, 2010.

[32] B. Yang, H. Lu, C.S. Jensen. Probabilistic threshold k nearest neighbor queries over moving objects in symbolic indoor space. *In EDBT Conf.*, 2010.

[33] M. L. Yiu, N. Mamoulis, X. Dai, Y. Tao, M. Vaitis. Efficient Evaluation of Probabilistic Advanced Spatial Queries on Existentially Uncertain Data. *IEEE Trans. Knowl. Data Eng.*, 21(1), 2009.

[34] S.M. Yuen, Y. Tao, X. Xiao, J. Pei, D. Zhang. Superseding Nearest Neighbor Search on Uncertain Spatial Databases. *IEEE Trans. Knowl. Data Eng.*, 22(7), 2010.

Appendix A: Dealing with Null Values

In classical (non probabilistic) databases, when processing SUM queries, the null (unknown) values are usually replaced by zero. Under the tuple level model, the null value has the same meaning as in classical databases. Thus, we simply replace the null values by zero without changing their probabilities.

Under the attribute level model, the null values are taken into account as follows. Let t_i be a tuple under this model, $v_{i,1}, v_{i,2}, \ldots, v_{i,m}$ the possible values for the aggr attribute of t_i, and $p_{i,1}, p_{i,2}, \ldots, p_{i,m}$ their probabilities. Let p be the sum of the probability of possible values in t_i, i.e. $p = p_{i,1} + p_{i,2} + \ldots + p_{i,m}$. If $p<1$, then there is the possibility of null value (i.e. unknown value) in tuple t_i, and the probability of the null value is $(1-p)$. We replace null values by zero as follows. If the zero value is among the possible values of t_i, i.e. there is some $v_{i,j}=0$ for $1 \leq j \leq m$, then we add $(1-p)$ to its probability, i.e. we set $p_{i,j}= p_{i,j} + 1 - p$. Otherwise, we add the zero value to the possible values of t_i, and set its probability equal to $(1-p)$.

Appendix B: Dealing with Tuples with Different Possible Aggr Values

Under the attribute level model, in our recursive approach for computing SUM we assumed that all tuples have the same number of possible aggr values. However, there may be cases where the number of possible aggr values in tuples is not the same. We deal with those cases as follows. Let t be the tuple with maximum number of possible aggr values. We set m to be equal to the number of possible values in t. For each other tuple t', let m' be the number of possible aggr values. If $m'<m$, we add $(m - m')$ new distinct values to the set of possible values of t', and we set the probability of new values to zero. Obviously, the new added values have no impact on the results of ALL_SUM queries because their probability is zero. Thus, by this method, we make the number of possible aggr values in all tuples equal to m, without impacting the results of ALL_SUM queries.

```
Algorithm DP_PSUM() {simplified version}
Input:
  n : number of tuples;
  t1, …, tn : the tuples of the database;
  MaxSum : maximum possible sum;
  p(t) : a function that returns the probability of tuple t;
Output:
  Possible sum values and their probability;
Begin
Let PS [0..MaxSum, 1..n] be a 2 dimensional matrix;
//Step 1 : initialization
   For i=1 to MaxSum do
      PS[i, 1] = 0;
   If (val(t1) = 0) Then
      PS[0, 1] = 1;
   Else begin
       PS[0, 1] = 1 - p(t1);
       PS[val(t1), 1] = p(t1) ;
   End ;
//Step 2 : filling the columns
   For j=2 to n do
      For i=0 to MaxSum do begin
         PS[i, j] = PS[i, j-1] × (1 – p(tj))
         If ( i – val(tj) ≥ 0) then
             If ( PS[i – val(tj), j - 1] > 0) then
                PS[i, j] =  PS[i, j] + PS[i – val(tj), j - 1]×p(tj);
      End;
// returning the results to the user
  For i=0 to MaxSum do
    If (PS[i, n] ≠ 0) then
       Return  i, PS[i, n];
End;
```

Figure 16. Pseudocode of DP_PSUM algorithm for tuple-level model

Appendix C: Pseudocode of AL_SUM algorithms

Figure 15 and Figure 16 show the pseudocode of the Q_PSUM and DP_PSUM algorithms.

```
Algorithm Q_PSUM()
Input:
  n : number of tuples;
  t1, …, tn : the tuples of the database;
  p(t) : a function that returns the probability of tuple t;
Output:
  Possible sum values and their probability;
Begin
//Step 1 : initialization
   Q = {};
   If (val(t1) = 0) then  Q = Q + {(0, 1)};
   Else Begin
      Q = Q + { (0, 1 - p(t1)) } ;
      Q = Q + { (val(t1), p(t1)) } ;
   End ;
//Step2 : constructing Q for DB^2 to DB^n
  For j=2 to n do Begin
      Q1 = Q2 = {};
      // construct Q1 and Q2
      For each pair (i, ps)∈Q do Begin
         Q1 = Q1 + {(i, ps×(1 – p(tj))};
         Q2 = Q2 + (i+val(tj), ps×p(tj));
      End;
      Q = Merge(Q1,  Q2); // the merge is done in such a way
     //that if exists (i, ps1)∈Q1 and (i, ps2)∈Q2 then
     // (i, ps1 + ps2) is inserted into Q.
  End;
  // returning the results to the user
 While (Q.empty() == False) do Begin
     (i, ps) = Q.removefirst();
     If (ps ≠ 0) then
```

Figure 15. Pseudocode of Q_PSUM algorithm

```
            Return  i, ps;
  End;
End;
```