Towards Correcting Input Data Errors Probabilistically Using Integrity Constraints

Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu
Department of Computer Science and Engineering
University of Washington
Seattle, WA
{nodira,magda,suciu}@cs.washington.edu

ABSTRACT

Mobile and pervasive applications frequently rely on devices such as RFID antennas or sensors (light, temperature, motion) to provide them information about the physical world. These devices, however, are unreliable. They produce streams of information where portions of data may be missing, duplicated, or erroneous. Current state of the art is to correct errors locally (*e.g.*, range constraints for temperature readings) or use spatial/temporal correlations (*e.g.*, smoothing temperature readings). However, errors are often apparent only in a global setting, *e.g.*, missed readings of objects that are known to be present, or exit readings from a parking garage without matching entry readings.

In this paper, we present StreamClean, a system for correcting input data errors automatically using application defined global integrity constraints. Because it is frequently impossible to make corrections with certainty, we propose a probabilistic approach, where the system assigns to each input tuple the probability that it is correct.

We show that StreamClean handles a large class of input data errors, and corrects them sufficiently fast to keep-up with input rates of many mobile and pervasive applications. We also show that the probabilities assigned by StreamClean correspond to a user's intuitive notion of correctness.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems *query processing*

General Terms: Algorithms, Design, Languages

Keywords: Stream processing, probabilistic databases, data cleaning, entropy maximization

1. INTRODUCTION

Many mobile and pervasive applications provide useful services to users by continuously collecting and processing information from the physical world. For example, an application may use Radio Frequency Identification (RFID) information from geographically distributed antennas to enable users to track their tagged equipment or manage their tagged inventory [18, 22, 29]. Another application may track cars that enter, exit, and park in a garage in order to monitor overall utilization [23], or possibly inform drivers of currently available parking spots.

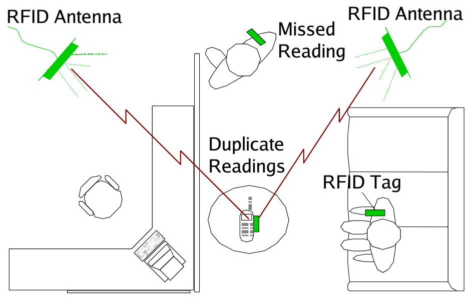


Figure 1: Example of input errors in an RFID-based equipment tracking application.

Mobile services that need to continuously process information "streaming in" from data sources can efficiently be implemented on top of what are called *stream processing engines (SPEs)* [2, 12], (*a.k.a.*, data-stream management systems [1, 26], or continuous query processors [10]).

The challenge in building such services is that, in real deployments, failures are likely to occur: SPE nodes can crash, communication between SPE nodes can be interrupted, data sources can crash, become disconnected, or they can produce incorrect input data. Although the service quality depends on the overall system reliability, the data sources are often the most brittle components [22]. As illustrated in Figure 1, in an RFID-based equipment-tracking application, an antenna can easily fail to detect a tag causing an input tuple to be missing, or two antennas can detect the same tag, resulting in erroneous duplicate inputs. As a result, even an otherwise fault-tolerant equipment-tracking application can misinform users about the location of their items.

In this paper, we propose StreamClean, a system for enhancing a stream-processing engine with the capability of detecting and handling input data errors. Previous work on SPE reliability focused on crash-failures of processing nodes and network failures [4, 21, 27], ignoring data source failures. Some SPEs manage the quality of input data by correcting input streams with per-input filters [22] or leveraging spatial and temporal correlations to smooth the data [16]. Other errors require that the user specify the algorithm necessary to clean the data [22]. In contrast to previous proposals, StreamClean handles various types of local and global errors without requiring users to indicate the data cleaning algorithm. Instead, users specify a set of global application-specific integrity constraints over the input data. A constraint can, for example, specify that "at any time t, each car that entered the garage at time $t_e < t$ must either be parked or must have exited the garage at some time

$t_x \in [t_e, t]$". StreamClean verifies such constraints at runtime. Constraint violations indicate errors.

To recover from input errors, StreamClean uses a probabilistic approach. It "cleans" the input data by inserting missing tuples when necessary and, for groups of conflicting tuples, assigning to each one the probability that it is correct. To compute these probabilities, StreamClean uses a non-linear optimization method, where the objective is to find a probability assignment that is the most uniform possible (*i.e.*, that maximizes entropy) while satisfying all integrity constraints. Other approaches are possible, but beyond our scope here.

In this paper, we show preliminary results demonstrating the feasibility of the approach. We show that StreamClean handles a large class of input data errors. We also show that using application-defined integrity constraints to detect errors and using entropy maximization to assign probabilities to tuples yields useful and intuitive probabilistic results. Finally, we demonstrate the feasibility of resolving constraint violations using non-linear programming by showing that large systems of equations can be quickly solved. Using a publicly available program [8] for Matlab [25], we find that the median time to solve a system with 100 equations and up to 20 variables per equation is below 175 msec. The overall time increases linearly with the size of the system but the pace of the increase is slow: a factor of 4 increase in system size yields only a factor 2.4 increase in processing time.

2. RELATED WORK

The most closely related project to ours is the Extensible Sensor stream Processing (ESP) framework [22], part of the HiFi project [18]. ESP allows a user to specify, in the form of declarative queries, the sequence of algorithms that the system should use to clean the data. This approach works well in many scenarios where the user can predict the types of errors that will occur and the appropriate algorithms to correct them. In StreamClean, we investigate a different approach, where the user only *states the properties that should be true about the data*. Additionally, because it is frequently impossible to make corrections with certainty, instead of attempting to completely clean the data, we propose a probabilistic approach. EPS proposes a five-stage data cleaning pipeline. Within this taxonomy, StreamClean fits well at the advanced cleaning stages called "Arbitrate" and "Virtualize", leveraging simple low-level cleaning mechanisms that average measurements within a short time-window and across a group of sensors covering the same area.

Deshpande *et al.*, [16] propose to handle input errors by building a probabilistic model of the spatial and temporal correlations between values produced by different sensors. The model then serves to predict missing values and identify outliers. The main challenge of the approach lies in selecting, building, and maintaining an appropriate model. StreamClean explores an alternate approach where a user directly specifies the input-data properties relevant to a given deployment and application.

Bertossi and Chomicki [6] describe a framework in which queries can be answered over inconsistent databases: the database violates some integrity constraints, but the user still wants to evaluate queries over the database. Bertossi and Chomicki define a tuple to be a *certain* answer to a query if it appears as an answer to the query on all possible minimal "repairs" of the database. Later, Andritsos *et al.*, [3] extend this approach to a probabilistic framework, in which the repairing tuples are associated some probabilities, thus reducing the problem to query evaluation on a probabilistic database as studied in [13, 14]. Our approach follows this line of research, in that the input streams can be viewed as a database violating the constraints given by the users. We consider, however, constraints that are much more complex than [3] (which only consider key constraints), as we have found them to be needed in stream-processing applications.

Probabilistic databases have a long history and have mostly focused on the data model and query evaluation approaches. Cavallo and Pittarelli [9] describe a model in which all tuples in the same table are disjoint (exclusive); Barbara *et al.*, [5] consider independent/disjoint tuples, but restrict the queries; Fuhr [19] describes a "complete" data model and gives an exponential space evaluation algorithm; Lakshmanan describes a model in which probabilities are approximated by intervals; Dalvi [14], and later in [13], shows that select-project-join queries are either efficiently computable (and give an explicit algorithm to compute them) or are #P-hard. Widom describes a probabilistic system called Trio [30] where probabilities and data lineage become first class citizens; the design space for probabilistic data models for Trio is described in [15]. This line of work is complementary to ours. We do not consider query evaluation in this paper, and the probabilistic data model that we describe here is unique, and quite specific to our application: the probabilistic data is the result of constraints being violated.

3. APPROACH OVERVIEW

In this section, we present a motivating scenario and the high-level overview of StreamClean.

3.1 Motivation and Goals

Imagine an RFID-based book tracking system deployed in a library [22]. Such a deployment is similar to that shown in Figure 1, except that RFID tags are attached to books and antennas are deployed on shelves and near tables. Every time a tag is in the vicinity of an antenna, the antenna detects the tag and produces an event. If a tag remains near an antenna, the antenna produces periodic events indicating the presence of the tag. With this deployment, librarians can see the location of each book at any time. The challenge is that RFID antennas are not reliable [17, 22]: they can fail to read a nearby tag or detect a tag that is relatively far away and also detected by another antenna.[1] Antenna errors can cause users to receive erroneous information. If an antenna fails to detect a nearby book, and the book is misshelved, a librarian will not be notified about the problem. If multiple antennas detect the same book, a librarian may simultaneously receive information that the book is on the correct shelf and an alert that the book is misshelved. Such erroneous information can be at best annoying to users, and at worst discourage them from using the system.

Ideally, we would like the system to correct all input data errors. However, it is often not possible to correct data source errors with certainty. For instance, if two antennas detect the same book, it is not always clear which antenna is correct. We thus propose to correct input data *probabilistically*. When errors occur on input streams, we would like the results sent to applications to be annotated with the probability that they are correct. The system could, for example, indicate that there is a 20% chance the book is on the correct shelf, a 75% chance it is misshelved, and a 5% chance that it is lying on one of the tables. With this approach, an application can take into consideration the probabilities when processing the

[1]To provide at least some form of fault-tolerance, the process polling antennas usually polls them multiple times in a short time-period and produces one input tuple for each distinct tag detected within that period. Even with this setup, antennas sometimes fail to detect tags or multiple antennas detect the same tag.

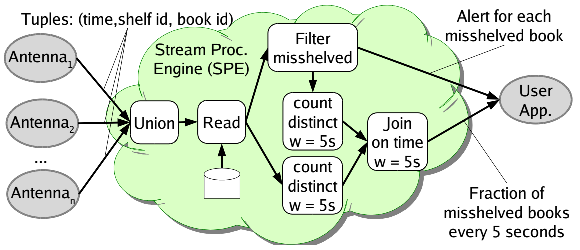


Figure 2: An SPE, showing a query diagram.

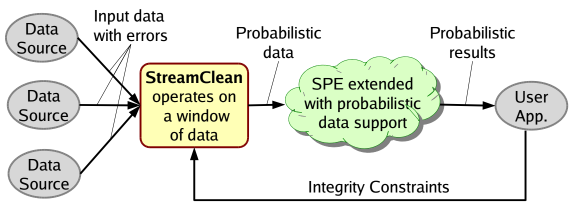


Figure 3: An SPE extended with StreamClean.

data. Whether it exposes these probabilities to the user depends on the application.

Such a system requires four pieces of functionality:

1. A mechanism to detect input data errors.
2. An algorithm to correct input data by assigning probabilities reflecting some *intuitive* notion of correctness.
3. Added capability in stream processing engines to process probabilistic data.
4. Modification of applications to handle probabilistic results. For example, a GUI could display books in different shades depending on their probability of being at each of the shown locations.

In this paper, we propose, StreamClean, a system for (1) detecting and (2) correcting input data errors. We do not address (3) the problem of extending a stream processing engine with the capability of handling probabilistic data, nor (4) how to modify applications to handle probabilistic data. For (3), one approach would be to adapt techniques from probabilistic databases [11, 13, 14]. We show one example of how this would work in Section 6.1, but we leave the details of the complete adaptation for future work.

As has been widely noted, input data errors are not specific to RFID settings. They affect most sensor-based applications [22]. Although we use the library scenario throughout this paper, the approach that we propose is applicable to any application that collects and uses information from RFID antennas or other sensors, with the limitation that StreamClean currently handles only discrete domains.

3.2 Processing Model and Approach

In this section, we present the processing and error models, we show how StreamClean fits into the overall architecture of an SPE, and outline its main components.

In a stream processing application, geographically distributed data sources (*e.g.*, RFID antennas) produce continuous streams of information. The SPE processes these streams (filters, correlates, and aggregates them) by pushing them through a dataflow of operators, called a *query diagram* [2]. Since streams are unbounded, and users need timely information about the monitored system, operators typically perform their computations over windows of data that slide with time [1, 10, 26]. Figure 2 shows an example of a query diagram. In this example, each antenna reports every five seconds, the set of books it has detected in the past five-second interval; producing one input tuple per sighted book. The *Read* operator performs a lookup in a back-end database to associate information about the proper location of each sighted book. The *Filter* operator filters misshelved books. The *Count distinct* operators coupled with the *Join* operator compute the fraction of distinct misshelved books over all distinct sightings in the past five-second period.

As illustrated in Figure 3, we propose to insert StreamClean between the data sources and the operators running at the SPE. This enables StreamClean to detect and correct errors before any additional processing occurs. In this paper, we assume that all input streams go to a central location running a single instance of StreamClean. Distribution issues are outside the scope of this paper.

We assume that three types of errors can occur on input streams: (1) input tuples may be missing, (2) erroneous input tuples may be inserted, or (3) input tuples may have erroneous values in some of their attributes (*e.g.*, a wrong temperature reading). Because these errors can be caused by software bugs, crashes, or environmental conditions, they can affect either all the tuples on a stream, a burst of tuples, or individual input tuples.

StreamClean uses application-defined constraints to detect and recover from input data errors. Integrity constraints can be defined over both stored relations and input streams. We discuss StreamClean's constraint language and the types of constraints it supports in Section 4.

In order to keep up with high input rates, we propose that the system accumulates input tuples for a short application-specific time-period, *d* (*e.g.*, a few seconds), without processing them. At the end of each time-period, the SPE verifies and corrects the new input before pushing it to the SPE.

StreamClean can keep additional history (older than the window *d*) for the purpose of constraint checking. However, because data streams are continuously processed by operators, StreamClean, assigns probabilities and otherwise corrects only *new data*. StreamClean does not recompute probabilities assigned to earlier tuples because these tuples have already been processed (or are being processed).

To correct input data, one approach would be to simply drop erroneous inputs and replace them with subsequent readings. The problem with this approach is that some events occur only once (*e.g.*, a book that is checked-out, a person passing-by an antenna) and some failures cause long sequences of erroneous inputs (*e.g.*, an antenna may systematically detect a book on a nearby shelf). Instead, to correct input data errors, StreamClean uses a technique based on non-linear programming. The idea is to transform integrity constraints over the data into constraints over the probabilities that individual tuples are correct. Among the probability assignments that satisfy a set of constraints, StreamClean seeks the one that provides the most uniform probability distribution. We further discuss how StreamClean corrects input streams in Section 5.

We evaluate StreamClean in Section 6, and discuss challenges and future work in Section 7.

4. INTEGRITY CONSTRAINTS

We propose a simple yet powerful constraint language that enables a user to express different types of integrity constraints over input data streams. The constraint language has a single construct:

```
FORALL INPUT1 AS I1, INPUT2 AS I2, ..., INPUTn AS In
WHERE EXPR1
CHECK EXPR2
CONFIDENCE c
```

The CHECK clause is mandatory, whereas the FORALL, WHERE, and CONFIDENCE clauses are optional. The arguments of the FORALL clause have the same syntax as a SQL FOR clause, EXPR1 is some arbitrary SQL condition, and c is a number between 0 and 1 which defaults to 1. The condition in the CHECK clause is of the form COUNT(Q) oprel k, where oprel is <, >, <=, >= and Q is a select-from-where SQL query. We only consider the COUNT aggregate function in this paper, and plan to explore other functions in future work. However, even with only the COUNT aggregate, a large class of constraints can be represented as shown below.

Because different types of constraints must be resolved differently, we classify constraints along three axes: (1) the type of dependency they express between tuples (inclusion or exclusion), (2) whether they affect tuples that occur approximately at the same point in time or at different points in time (stateful or stateless), and finally (3) whether they are hard constraints or soft constraints. Hard constraints must always be true while soft constraints are true only "most of the time". For each type of constraint, we show how a user can express it using the above construct.

4.1 Exclusion and Inclusion Constraints

StreamClean distinguishes and handles two basic types of dependency between tuples. Some tuples *exclude* each other: they cannot all be true in the same instance of the world. For example, a book cannot appear on a shelf and on a table at the same time. Other tuples, in contrast, *include* each other: if some tuples are true, then other tuples must also be true. For example, if a book is returned to the library, it must have been previously checked-out. Exclusion constraints typically come into play when errors cause input data to be either duplicated or contain erroneous values. Inclusion constraints in contrast come into play when errors cause input data to be missing or contain erroneous values.

4.1.1 Exclusion Constraints

For this type of constraint, the user must simply indicate that the presence of a tuple or a group of tuples (defined with the FORALL ... WHERE clause) *limits* the presence of other tuples (defined with the CHECK clause). To indicate an *exclusion*, the condition in the CHECK clause takes the form COUNT(Q) oprel k, where oprel is < or <= and Q is a select-from-where SQL query.

Example 1. In our language, the exclusion constraint: "A book can appear in at most one location at any point in time" is expressed as follows.

```
FORALL SIGHTINGS S
WHERE S.t = now()
CHECK COUNT(SELECT * FROM SIGHTINGS T
            WHERE T.t = S.t AND T.bid = S.bid) <= 1
```

We assume the system provides a user-defined function now() that returns the identifier of the current time-period, and that each antenna, sid, produces a tuple every time-period, t, for each book, bid, it detected in that time-period.

Example 2. The exclusion constraint: "There are at most 10 books on a shelf" is expressed as:

```
FORALL SHELVES S
CHECK COUNT(SELECT * FROM SIGHTINGS T
            WHERE S.sid = T.sid AND T.t = now()) <= 10
```

4.1.2 Inclusion Constraints

For this type of constraint, the user must simply indicate that the presence of a tuple or a group of tuples (defined with the FORALL ... WHERE clause) *implies* the presence of other tuples (defined with the CHECK clause). To indicate an *inclusion*, the condition in the CHECK clause takes the form COUNT(Q) oprel k, where oprel is > or >=.

Example 3. Consider the constraint: "Each book that is in the catalog must appear in at least one location at any time." This constraint is expressed as:

```
FORALL BOOKS B
CHECK COUNT( SELECT * FROM SIGHTINGS S
             WHERE S.bid = B.bid and S.t = now()) >= 1
```

4.2 Stateless and Stateful Constraints

A dependency can affect tuples that occur approximately at the same point in time (within the same window d) or can affect tuples that occur at different points in time. We say that a constraint is *stateless* if it applies to tuples in the most recent window d independently of what happened in the past. Otherwise, we classify the constraint as *stateful*. The above constraints were all stateless. We now show two examples of stateful constraints.

Example 4. Stateful exclusion constraint: "A checked-out book cannot appear in the library." (For simplicity, we assume that there is no more than one check-out and return for each book in StreamClean's history).

```
FORALL CHECKOUTS C
CHECK COUNT( SELECT * FROM SIGHTINGS S
             WHERE S.bid=C.bid and S.t=now()) <= 0
```

Example 5. Stateful inclusion constraint: "A returned book must have previously been checked-out."

```
FORALL RETURNS R
WHERE R.t = now()
CHECK COUNT( SELECT * FROM CHECKOUTS C
             WHERE C.bid=R.bid and C.t<R.t) >= 1
```

4.3 Hard or Soft Constraints

The CONFIDENCE clause allows the user to indicate the likelihood that a constraint holds. When omitted, the confidence defaults to 1, indicating that the constraint is hard. For any value below 1, the constraint is said to be soft, as it holds only some of the time.

Example 6.The soft constraint: "Every book seen at time t on a shelve is likely to be on the same shelve at time $t+1$." is expressed as:

```
FORALL SIGHTINGS S
WHERE S.t = now() - 1
CHECK COUNT( SELECT * FROM SIGHTINGS T
              WHERE S.bid=T.bid and T.t = now()) >= 1
CONFIDENCE 0.3
```

If a book is not detected at time t then the constraint in Example 3 will place the book equally likely on any shelve in the library. This new constraint will place the book with a higher probability on the same shelf as it was at time $t-1$.

5. RESOLVING VIOLATIONS

In this section, we present the constraint resolution algorithms. For each time-window d, StreamClean first checks all integrity constraints against the available data (window, d, of new tuples, window of old tuples, and stored relations). For each constraint violation, StreamClean inserts or deletes new input tuples as necessary, and translates the constraint into an equation over tuple probabilities (Section 5.1). StreamClean then solves the system of equations to assign probabilities to the new input tuples (Section 5.2).

```
Input: A set of arriving tuples $T$ for window $[t_1, t_2]$,
a set of user-defined integrity constraints $I$
Output: A set $C_{ME}$ of equations to represent the constraints
 1.  Let $C_{ME}$ = [] //constraint equations for this window
     //Translate inclusion constraint violations
 2.  while $\exists$ violated $C \in I$ with a $>$ or $\geq$ in the check clause
 3.    if $C$ does not contain FORALL clause
 4.      Let $y_1, y_2, ..., y_n$ = fixup_tuples$(C, null)$
 5.      Add $y_1, y_2, ..., y_n$ to $T$
 6.      Add following constraint to $C_{ME}$:
 7.        i. $\sum_{i=1}^{n} p_{y_i} \geq$ expected_count$(C)$
 8.    else //C does have FORALL clause
 9.      Let $x_1, x_2, ..., x_n$ = violating_tuples$(C)$
10.      for each $x_i$
11.        Let $y_1, y_2, ..., y_m$ = fixup_tuples$(C, x_i)$
12.        Add $y_1, y_2, ..., y_m$ to $T$
13.        Add following constraints to $C_{ME}$
14.          i. $\sum_{j=1}^{m} p_{y_j} \geq$ expected_count$(C)$
15.          ii. $p_{x_i} - \sum_{j=1}^{m} p_{y_j} = 0$
     //Translate exclusion constraint violations
16.  while $\exists$ violated $C \in I$ with a $<$ or $\leq$ in the check clause
17.      Let $G$ = violating_groups$(C)$
18.      for each group $g = \{x_1, ..., x_n\} \in G$
19.        Add following constraint to C:
20.          1. $\sum_{i=1}^{n} p_{x_i} \leq$ expected_count$(C)$
     //Remove tuples which are not in current window from $C_{ME}$
21.  for each equation $E$ in $C_{ME}$
22.      for each tuple $x$ involved in $E$ s.t. $x.t < t_1$
23.        Remove $x$ from the left-hand side of $E$ and from $T$
24.        Subtract $p_x$ from the right-hand side of $E$
```

Figure 4: Algorithm for translating constraint violations into equations.

5.1 Translating Constraints into Equations

We give the overall algorithm for translating the integrity constraints into equations in Figure 4. The algorithm presented is for hard constraints only. We present the procedure for handling soft constraints at the end of this section.

The procedure begins by processing inclusion constraints (Figure 4: line 2). Two sets of tuples are involved in these constraints, the premise tuples and the conclusion tuples[2]. The premise tuples are those provided by the `FORALL` clause, *i.e.*, tuples that appear in the input streams. The second set are the conclusion tuples, *i.e.*, tuples which should have been present in the input stream. If the constraint is broken, less than the specified number of conclusion tuples have appeared in the stream. In this case, the algorithm generates and inserts the *fix-up tuples*, the set of all possible conclusion tuples that are not present in the stream (line 5, 12). To simplify the discussion, when an inclusion constraint is violated, we assume that no conclusion tuples are present.

The algorithm also adds to the set of equations, C_{ME}, the restriction that the sum of the probabilities of the fix-up tuples must be greater than (or equal to) the `expected_count` of conclusion tuples (line 7, 14) (In reality, this is the `expected_count` minus the number of conclusion tuples already present in the stream). This is done for constraints with or without a `FORALL` clause.

If the constraint has a `FORALL` clause, another equation (line 15) is included to allow the probability of the premise tuple to be adjusted instead of only that of the fix-up tuples. It is particularly useful when the premise tuple occurs in the current time window, and the fix-up tuples should have occurred in the past. We do not adjust the probabilities of old tuples, so the only option is to remove the premise tuple.

[2]We borrow these terms from logic terminology where they are used to describe the two sides of an implication.

We now explain the generation of fix-up tuples (line 4, 11). Consider an inclusion constraint of the form:

```
FORALL INPUT1 AS I1, INPUT2 AS I2, ..., INPUTn AS In
WHERE EXPR1
CHECK COUNT( SELECT * FROM Reln_Name R
             WHERE EXPR2) >= k
```

If there is no `FORALL` clause, the system generates all possible tuples that can be constructed from the active domain of table R. It then selects only those tuples that satisfy `EXPR2`. Note that the number of fix-up tuples is typically much larger than the `expected_count`: *e.g.*, in Example 3 the system will generate one fix-up tuple for every possible location of the missing books. The system then adds the constraint that the expected number of fix-up tuples (*i.e.*, the sum of their probabilities) is $\geq$ `expected_count`.

If there is a `FORALL` clause, we first select the premise tuples that break the constraint. This is done by taking a left outer join between the premise tuples and R and then selecting the tuples where the attributes of R are all null. (*i.e.*, all premise tuples that do not join.) Second, for each violating premise tuple, we generate the fix-up tuples, by generating all possible tuples over R, and filtering on `EXPR2`. This is the same as before, except this time `EXPR2` may depend on the premise tuple's attributes.

Next, the algorithm handles exclusion constraints. It first identifies the groups of tuples that violate an exclusion constraint together (line 17). Then for each group, it adds an equation to C_{ME} stating that the tuples' probabilities must sum to less than (or equal to) the specified count (line 20). The challenging part is identifying the conflicting groups.

To identify the conflicting groups of tuples we take a join between the outer `FORALL` clause relations, and the inner `CHECK` clause relation. For example, given an exclusion constraint of the same form as the inclusion constraint above except with a $\leq$ operator, we execute the following join:

```
SELECT *
FROM (SELECT * FROM INPUT1 AS I1, ..., INPUTn AS In
      WHERE EXPR1) as S INNER JOIN Reln_Name as R
ON EXPR2
```

Any pair of tuples, across S and R, that join in this view participate in the same conflicting group. If we think of each tuple as a vertex in a graph, with edges between two tuples if they join, each strongly connected component of this graph corresponds to a conflicting group.

Soft constraints are handled similarly to hard constraints, except that the `expected_count` threshold in the equations is adjusted by the confidence factor. For inclusion (exclusion) constraints, the `expected_count` is multiplied (divided) by the confidence factor. The confidence factor effectively increases the search space and thus softens the constraint.

The final step of the translation (lines 21-24) caters for stateful constraints, by handling old tuples and their probabilities. Since the probabilities for these tuples have already been fixed, we must ensure that they are not re-adjusted. Therefore, the algorithm removes these tuples from the equations, and adjusts the equations accordingly. Similarly, tuples from stored relations are also removed from these equations.

```
Input: A set $C_{ME}$ of equations over probabilities of tuples $T$
Output: A set of tuples $T'$ with associated probabilities
Let $x_1,...,x_n$ be the tuples in $T$, all initialized to probability 1.0

Maximize $-\sum_{i=1}^{n}(p_{x_i} \cdot \log p_{x_i})$
subject to $C_{ME}$

for each tuple $x_i$ in $T$
    Add $x_i$ with its associated probability $p_{x_i}$ to $T'$.
```

Figure 5: Assigning probabilities to tuples using entropy maximization.

5.2 Assigning Probabilities to Tuples

The generated system of equations bounds the probabilities that StreamClean can assign to input tuples. The next step is to actually compute and assign these probabilities.

In most cases, multiple probability assignments are possible. For example, if two tuples exclude each other, StreamClean could assign a probability of 1.0 to one of them and 0.0 to the other one, or even assign a 0.0 probability to both tuples. A more intuitive solution, however, would be to assign a probability of 0.5 to each tuple. This intuition is captured by the maximum-entropy principle [20], which favors a solution that meets all explicitly imposed constraints, but is otherwise as uniform as possible. Mathematically, maximizing the entropy of tuple probabilities, p_x, translates into maximizing the entropy function [28], E, defined as $E(p_x) = -\sum_{i=1}^{n}(p_{x_i} \cdot \log p_{x_i})$ subject to the imposed constraints. By using entropy maximization (EM), we also make the explicit assumption that in the absence of stated constraints the probabilities of different input tuples are independent of each other. Figure 5 summarizes the approach.

EM is a well-known technique, and it has been successfully applied in many domains. For example, Markl *et al.*, use EM to estimate the selectivity of conjunctive predicates [24]. Several algorithms exist to perform this type of optimization numerically. StreamClean performs the computation using a publicly available program [8] for Matlab [25].

Once the probabilities have been computed, StreamClean sends the tuples and their assigned probabilities to the SPE and moves on to correcting the next window of input tuples.

6. EVALUATION

We study the feasibility of using entropy maximization to resolve integrity constraint violations. We evaluate the feasibility from two orthogonal perspectives, through a qualitative evaluation and a quantitative evaluation.

6.1 Qualitative Evaluation

For StreamClean to be useful, the probabilities it assigns to input tuples must correspond to what a user might expect. We evaluate whether the approach results in such intuitively correct assignments by presenting an end-to-end example. We first examine the probabilities assigned to input tuples through entropy maximization. We then examine the resulting probabilities of the output tuples of a query.

6.1.1 Input Probabilities

Consider the library scenario but with only three book shelves S_1, S_2 and S_3, no tables, and with one antenna per shelf. For each time-period, t, each antenna produces an input tuple for each book, b, that it detected within that time-period. Each input tuple takes the form (t, S_i, b).

We first consider two constraints: (1) a shelf contains no more than ten books at any given time (exclusion constraint from Section 4.1.1 Example 2); (2) each book that is in the catalog must appear in at least one location at any time (inclusion constraint from Section 4.1.2 Example 3). Suppose that we are at time $t = 5$ and we do not have a reading for a book with book ID b_{34}. Because the missing book can be on anyone of the three shelves, we expect StreamClean to assign a probability of $\frac{1}{3}$ to each of these possible sightings.

Now suppose that S_2 and S_3 appear full according to the antennas that monitor them. S_1's antenna reads only three books, b_1, b_2, b_3 (misses b_{34}), whereas S_2's antenna reads ten books, $b_4, b_5, ..., b_{13}$ (b_4-b_6 are in reality checked-out) and S_3's antenna's reads twenty books $b_{14}, b_{15}, ..., b_{33}$ (b_{14}-b_{18},b_{29}-b_{33} are actually on a nearby cart). With these readings, we expect the probability that b_{34} is on the empty shelf to be higher than the probability it is on S_2 or S_3, especially shelf S_3 that reports more books than its capacity.

Suppose a third stateful constraint specifies that checked-out books cannot appear in the library (Example 4, Section 4.2), and that books b_4, b_5 and b_6 were checked-out at time 2. Since the antenna on shelf S_2 reads the tags for the checked-out books, it may actually not be full after all. With this additional constraint, we would expect the probability that S_2 holds book b_{34} to increase.

Finally, suppose we also include the constraint (Example 1 Section 4.1.1): each object must be in at most one location at any time.

The translation from constraints to equations (Section 5.1) returns the following entropy maximization problem. $p_{t,s,b}$ is the probability that tuple (t, s, b) is correct.

$$\begin{array}{ll}
\text{Maximize } -\sum_{j=1}^{3}(\sum_{k=1}^{24} p_{5,S_j,b_k} \cdot \log p_{5,S_j,b_k}) & \\
\text{subject to} & \sum_{i=1}^{3} p_{5,S_1,b_i} + p_{5,S_1,b_{34}}) \leq 10 \text{ //constraint 1} \\
& \sum_{i=4}^{13} p_{5,S_2,b_i} + p_{5,S_2,b_{34}}) \leq 10 \text{ //constraint 1} \\
& \sum_{i=14}^{33} p_{5,S_3,b_i} + p_{5,S_3,b_{34}}) \leq 10 \text{ //constraint 1} \\
& p_{5,S_1,b_{34}} + p_{5,S_2,b_{34}} + p_{5,S_3,b_{34}} \geq 1 \text{ //constraint 2} \\
& p_{5,S_2,b_4} \leq 0 \text{ //constraint 3} \\
& p_{5,S_2,b_5} \leq 0 \text{ //constraint 3} \\
& p_{5,S_2,b_6} \leq 0 \text{ //constraint 3} \\
& p_{5,S_1,b_{34}} + p_{5,S_2,b_{34}} + p_{5,S_3,b_{34}} \leq 1 \text{ //constraint 4}
\end{array}$$

The program assigns the following probabilities.

$$\begin{array}{l}
p_{5,S_1,b_{34}} = p_{5,S_2,b_{34}} = 0.4016 \\
p_{5,S_3,b_{34}} = 0.1968 \\
p_{5,S_1,b_1} = p_{5,S_1,b_2} = p_{5,S_1,b_3} = 1.0 \\
p_{5,S_2,b_4} = p_{5,S_2,b_5} = p_{5,S_2,b_6} = 0.0 \\
p_{5,S_2,b_7} = p_{5,S_2,b_8} = ... = p_{5,S_2,b_{13}} = 1.0 \\
p_{5,S_3,b_{14}} = p_{5,S_3,b_{15}} = ... = p_{5,S_3,b_{33}} = 0.4901
\end{array}$$

As we can see, the results match our expectations. The probability that the book is on shelf S_3 is quite low as S_3 is already reading too many book tags. Since S_1 is not full, there is a high probability that the book is there. Interestingly, the probability that the book is on S_2 is also high. Although the antenna gave us the impression that the bookshelf was full, StreamClean decided that books b_4, b_5 and b_6 are checked-out, and hence are less likely to be on the shelf.

6.1.2 Query evaluation

We now examine how the probabilities assigned to input tuples propagate to the results of the query shown in Figure 2. We assume that books b_1, b_2, b_3, b_{34} should be shelved on S_1, $b_4, ..., b_{13}$ on S_2, and $b_{14}, ..., b_{23}$ on S_3. We assume that query operators implement the approach proposed by Dalvi and Suciu [14].

The filter operator does not modify tuple probabilities. The misshelved books that pass the filter have the same probabilities as the input tuples. The count operators simply compute the *expected* values of their results. Assuming the tuple probabilities from Section 6.1, the count of distinct book sightings is 20.802 books. The count of misshelved books is 5.4994. Hence for the current time-period, this query returns the result $(5, 0.26437)$ where $t = 5$ is the current time and 0.26437 is the fraction of misshelved books. In reality, 5 out of 20 (*i.e.*, 0.25) books are misshleved. Antennas reported 10 out of 33 (*i.e.*, 0.303) books, but StreamClean helped bring this value closer to the actual one.

6.2 Quantitative Evaluation

StreamClean must perform a non-linear optimization for every window of tuples it corrects. We believe that this, the entropy maximization, will be the bottleneck of our approach. The other process-intensive operation is the identification of violating tuples and groups. This latter operation, however, boils down to executing database queries where most of the data is in the window of new tuples, which can easily be kept in memory. Hence, in this section, we evaluate if entropy maximization is sufficiently fast to keep up with the data input rate of a mobile or pervasive application. We present the results of micro-benchmarking an off-the-shelf Matlab implementation [8] of the entropy maximization algorithm by Blien and Grael [7]. Our main goal is to determine if entropy maximization is a feasible approach to pursue.

We evaluate two kinds of systems. The first kind, which we call *lightly violating*, are systems of equations where the number of equations (constraint violations) is less than or equal to the number of variables (tuples). The second kind, called *heavily violating*, are those where the number of equations is greater than or equal to the number of variables.

In a typical scenario, we expect StreamClean to accumulate and correct input tuples in windows that span a few seconds, as for many pervasive or mobile applications such a delay is tolerable. Each optimization must therefore complete within that time-frame.

We can easily expect a deployment where large numbers of constraints are violated within each time-window, and each constraint involves up to a few tens of tuples. However, as the number of violated constraints increases, it becomes likely that constraints can be partitioned into smaller independent groups. We thus run experiments with 100 equations. For each equation, we vary the number of variables per equation (*i.e.*, conflicting tuples) from 1 to 20. By doing so, we increase the size of the system of equations. For each configuration, we experiment with a different total number of variables. When the number of variables increases, each variable participates in a smaller number of equations and the solution space increases. We run all experiments on a machine with a dual 2.4GHz Xeon processor with 2GB of memory.

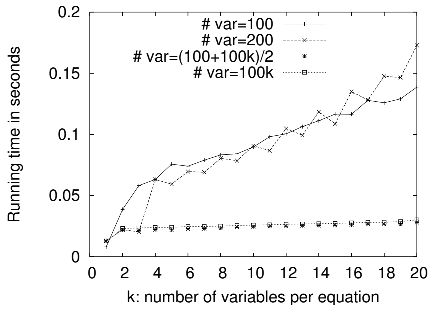
Because our goal is to determine the speed at which the equations can be solved, to simplify our experiments, we test the feasibility of entropy maximization only on exclusion constraint violations. To generate a test configuration, we assign variables to equations randomly, ensuring that a variable is not placed into the same equation twice, and that the resulting system of equations is fully connected.

Figure 6 presents the results. Each point is the median value across 50 runs of each experiment. The figure shows that the running times are reasonably fast even for large systems of equations. For example, with 20 variables per equation, 100 equations, and 2000 variables, the system is represented by a 2000 by 100 matrix of integers, yet the assignment of probabilities to tuples takes less than 175 ms.

The figures also show that, as soon as each equation contains at least a couple of variables (approximately 4 or 5) the time to solve the system of equations starts to increase linearly with the size of the system. The increase, however, is slow (low gradient). For a factor 4 increase in system size (from 100 equations with 5 variables per equation to 100 equations with 20 variables per equation), the time to solve the system of equations increases only by a factor of 2.4. Increasing the number of equations instead of the number of variables yields a similar linear increase. We do not show these results due to space constraints.

Interestingly, when the system is either so lightly violating that a large number of solutions exist or the system is so heavily violating that only few solutions exist, the system of equations is solved in a small and constant time.[3]

From these preliminary results, entropy maximization thus appears to be a feasible approach, since even relatively large systems of equations are solved within a fraction of a second, and the time to solve equations increases slowly with the size of the system.



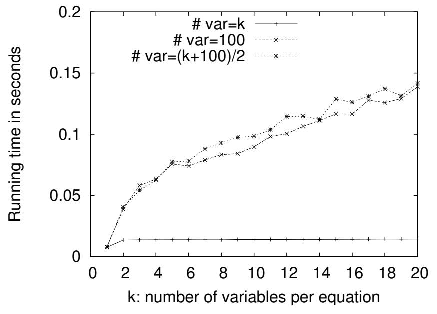


Figure 6: Time to solve a system of 100 equations with different numbers of variables per equation in a (a) lightly and (b) heavily violating system.

7. CONCLUSION

Devices, such as RFID antennas, light, motion, and temperature sensors are known to be unreliable. A fraction of data they report is erroneous, duplicated or missing. However, many mobile and pervasive applications rely on the data produced by such devices. To improve the quality-of-service of these applications, the input data must thus be cleaned before it can be processed.

We presented, StreamClean, a system for correcting input data errors probabilistically using user-defined integrity constraints. Our contributions include a simple declarative language for defining integrity constraints over multiple streams, the application of the concept of maximum entropy to the problem, and a preliminary feasibility study.

Our approach has several benefits. First, using a declarative con-

[3]A solution always exists as equations are inequalities with $\leq$ signs, and an assignment of all zero's is always feasible.

straint language simplifies the task of the user who only needs to express properties of the input data and not the algorithm necessary to clean it. Second, the language enables the expression of a wide range of constraints: local, global, static constraints, and even dynamic constraints, which define valid sequences of events created by moving objects (*e.g.*, a book that is checked-out then returned, a car that enters then exits a parking garage).

The work introduced in this paper also presents many challenges that we are currently addressing. One challenge is to integrate StreamClean with an existing stream processing engine such as Borealis [2] and experiment with a real RFID or sensor-based deployment, and real input rates. Such integration requires the modification of the SPE to handle probabilistic data. We are exploring the adaption of techniques from [14], which support arbitrarily complex SQL queries on probabilistic databases, including queries involving aggregate functions (*e.g.*, `SUM, MIN, MAX`).

We would like to extend StreamClean to produce appropriate fix-up tuples when tuple attributes come from continuous domains.

Another challenge is to expand the subset of constraints we can handle to support, for example, arbitrary aggregate operators rather than simply the `COUNT` operator. Providing such support within our framework is an open problem.

A final challenge is to investigate how applications can take advantage of the probabilistic information to provide more information to users without confusing them.

Overall, correcting their input data is an important challenge for mobile and pervasive applications, and StreamClean is a step toward automating this task.

Acknowledgments This work is partially support by Suciu's NSF CAREER Award IIS-00992955, and Khoussainova's New Zealand Federation of Graduate Women Fellowship.

8. REFERENCES

[1] Abadi et. al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), Sept. 2003.

[2] Abadi et. al. The design of the Borealis stream processing engine. In *Proc. of the CIDR Conf.*, Jan. 2005.

[3] P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases. In *Proc. of the 22nd ICDE Conf.*, 2006.

[4] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *Proc. of the 2005 SIGMOD Conf.*, June 2005.

[5] D. Barbara, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Trans. Knowl. Data Eng.*, 4(5):487–502, 1992.

[6] L. Bertossi and J. Chomicki. Query answering in inconsistent databases. In G. S. J. Chomicki and R. van der Meyden, editors, *Logics for Emerging Applications of Databases*. Springer, 2003.

[7] U. Blien and F. Grael. Entropy optimizing methods for the estimation of tables. In *I. Balderjahn, R. Mathar, and M. Schader, eds.: Classification, Data Analysis, and Data Highways (Springer Verlag, Berlin)*, 1997.

[8] M. Boss. Matlab central file exchange - entrop. `http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectI%d=5566&objectType=file`, 2004.

[9] R. Cavallo and M. Pittarelli. The theory of probabilistic databases. In *Proc. of the 13th VLDB Conf.*, pages 71–81, Sept. 1987.

[10] Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the CIDR Conf.*, Jan. 2003.

[11] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *Proc. of the 2003 SIGMOD Conf.*, pages 551–562, 2003.

[12] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the CIDR Conf.*, Jan. 2003.

[13] N. Dalvi, C. Re, and D. Suciu. Query evaluation on probabilistic databases. *IEEE Data Engineering Bulletin*, 29(1):25–31, 2006.

[14] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *Proc. of the 30th VLDB Conf.*, Toronto, Canada, Sept. 2004.

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

[16] A. Deshpande, C. Guestrin, and S. R. Madden. Using probabilistic models for data management in acquisitional environments. In *Proc. of the CIDR Conf.*, Jan. 2005.

[17] C. Floerkemeier and M. Lampe. Issues with RFID usage in ubiquitous computing applications. In *Proc. of the 2nd Pervasive Conf.*, Apr. 2004.

[18] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design considerations for high fan-in systems: The HiFi approach. In *Proc. of the CIDR Conf.*, Jan. 2005.

[19] N. Fuhr and T. Roelleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, 1997.

[20] S. Guiasu and A. Shenitzer. The principle of maximum entropy. *The Mathematical Intelligencer*, 7(1), 1985.

[21] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Proc. of the 21st ICDE Conf.*, Apr. 2005.

[22] S. Jeffery, G. Alonso, M. J. Franklin, W. Hong, , and J. Widom. Declarative support for sensor data cleaning. In *Proc. of the 4th Pervasive Conf.*, 2006.

[23] J. Liu and F. Zhao. Towards semantic services for sensor-rich information systems. In *Proc. of the Basenets Workshop*, Oct. 2005.

[24] V. Markl, N. Megiddo, M. Kutsch, T. Tran, P. Haas, and U. Srivastava. Consistently estimating the selectivity of conjuncts of predicates. In *VLDB*, pages 373–384, 2005.

[25] T. MathWorks. Matlab. `http://www.mathworks.com/products/matlab/`, 2006.

[26] Motwani et. al. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the CIDR Conf.*, Jan. 2003.

[27] M. Shah, J. Hellerstein, and E. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *Proc. of the 2004 SIGMOD Conf.*, June 2004.

[28] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27, 1948.

[29] M. L. Songini. Wal-Mart details its RFID journey. ComputerWorld.`http://www.computerworld.com/industrytopics/retail/story/0,10801,109132%,00.html`, Mar. 2006.

[30] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Proc. of the CIDR Conf.*, pages 262–276, 2005.