Randomized Algorithms for Data Reconciliation in Wide Area Aggregate Query Processing*

Fei Xu, Christopher Jermaine
Department of Computer and Information Sciences and Engineering
University of Florida
Gainesville, FL, USA, 32611
{feixu, cjermain}@cise.ufl.edu

ABSTRACT

Many aspects of the data integration problem have been considered in the literature: how to match schemas across different data sources, how to decide when different records refer to the same entity, how to efficiently perform the required entity resolution in a batch fashion, and so on. However, what has largely been ignored is a way to efficiently deploy these existing methods in a realistic, distributed enterprise integration environment. The straightforward use of existing methods often requires that all data be shipped to a coordinator for cleaning, which is often unacceptable. We develop a set of randomized algorithms that allow efficient application of existing entity resolution methods to the answering of aggregate queries over data that have been distributed across multiple sites. Using our methods, it is possible to efficiently generate aggregate query results that account for duplicate and inconsistent values scattered across a federated system.

1. INTRODUCTION

The problem of large-scale data integration [21, 6, 25, 16, 28], is of fundamental, real-world importance. In the enterprise setting, data are often scattered across dozens or hundreds of different databases, each of which were developed independently. The dominant approach to answering queries in the enterprise integration environment has been the so-called "federated" approach [21], where an enterprise-wide query is broken up into a set or a series of individual queries that are then issued to the actual data sources. Many systems-oriented aspects of this approach are well-understood, but quite surprisingly there is a significant gap in the state-of-the-art with respect to one of the most important applications for an enterprise information integration system: statistical or analytic processing. The specific problem that we consider in this paper is answering `SUM`, `AVERAGE`, `GROUP BY`, and/or `HAVING` queries and the like in a distributed fashion.

This sort of query processing is quite different from transaction processing, and at first glance, it might seem very easy to do in a federated environment. For example, imagine the following, simple query:

```
SELECT SUM(e.SALARY)
FROM EMP e
WHERE e.DIVISION = 'MFG' or e.DIVISION = 'SALES'
```

One simple way to answer this query is to compute a set of summary statistics at each individual site, and then send them back to a coordinator for final processing. If three sites have relevant data and the individual results are ans_1, ans_2, and ans_3, then the final query result can be computed as $ans = ans_1 + ans_2 + ans_3$.

Unfortunately, in practice things might be much more difficult due to duplicated and/or inconsistent data. In our example, an employee may have recently moved from one of the company's divisions to another, and as a result there may be two records associated with that employee at two different data sites. By simply adding the local totals, the employee would be counted twice, and one of the records counted would have old or stale data. Researchers have considered methods for discovering such duplicate data [23, 5, 29, 8, 12] and for efficiently handling this problem in a centralized environment [14, 1, 7]. However, what has *not* been addressed is how to extend these methods to the real-life situation where they are most applicable: a distributed system where different sites own different subsets of the data.

The obvious way to extend these methods to a distributed environment is to ship all of the relevant data to the coordinator, who gathers all of the data and performs the reconciliation locally. However, a single aggregate query may touch most of the database records across a large number of data sites. Because so much data are relevant to computing the result, the obvious solution may require gigabytes or terabytes of communication.

Our Contributions. We propose a set of algorithms that allow computation of an aggregate function over a database table that has been distributed across many sites, where deduplication and/or reconciliation may be needed. At the most fundamental level, our goal is to allow the use of any existing technique for entity resolution and data reconciliation during the aggregate computation, but to provide a framework that renders virtually any existing method practical in a distributed environment. This is not an easy problem: in the case where duplicate data may be located any-

*Material in this paper is based upon work supported by the National Science Foundation under Grant No. 0347408.

VLDB '07, September 23-28, 2007, Vienna, Austria.

where and the method used to reconcile the data is arbitrary, it is non-obvious how to avoid shipping large volumes of data around to locate duplicate or inconsistent records.

To avoid shipping all of the data, we propose using randomization. Due to the randomness, the algorithms are not guaranteed to provide an exact answer. However, the inaccuracy of the approach is rigorously monitored and statistical accuracy guarantees are provided. These guarantees are valid regardless of the underlying matching technology that is used. While some users may be hesitant to accept any uncertainty in the query result, there is a strong argument that incurring a small amount of rigorously-controlled inaccuracy in order to provide for efficient computation should not be a concern: data integration will *always* be an approximate process anyway, given the sort of questions that must be asked (that is, one may never be quite sure that `'John Smith'` at Site A is not the same person as `'Jon Smith'` at Site B). Thus, adding a bit of additional inaccuracy above and beyond the inaccuracy already incurred during the integration may be acceptable.

The technical contributions of this paper are as follows:

- We propose several different randomized algorithms for addressing the computational issues associated with wide-area reconciliation during wide-area aggregate processing.
- Our algorithms are very general, in the sense that they can handle any reconciliation problem specified by instantiating a user-defined similarity function $Sim()$ that tells when records need to be reconciled, and a user-defined reconciliation function $Rec()$ that performs the reconciliation.
- The applicability of our various algorithms depends upon simple properties of $Sim()$ and $Rec()$. As long as the user can accurately answer a single "yes/no" question regarding each of these functions, it is trivial for a system to choose the best combination of algorithms to use for greatest accuracy and efficiency.

Paper Organization. The paper is organized as follows. Section 2 formally describes our problem as well as the basic solution. Section 3 considers the important properties of the $Sim()$ and $Rec()$ functions that dictate specific, computational aspects of the solution. Section 4 discusses how the required randomness can be provided via distributed sampling, and Section 5 considers how to boost the accuracy of the computation. Section 6 presents experiments, Section 7 presents related work, and Section 8 concludes the paper.

2. THE RECONCILE-SUM PROBLEM

2.1 Example Application

Imagine that a large company has many employees working in different cities. At each location, a database table stores the information regarding employees' salaries. Since an employee may have recently moved from one of the company's locations to another, there may be two records associated with an employee at two different data sites.

R_1: New York	
Name	Salary
Michael	$10000
Daniel	$7864
David	$8433

R_2: Chicago	
Name	Salary
Christina	$7633
Steven	$8003
Sean	$9607

R_3: Los Angeles	
Name	Salary
Christina	$7412
Emily	$10822
Michael	$9899
James	$7322

Now, imagine that the company wishes to generate a report detailing its current salary expenditure. To allow for this, we assume the existence of a user-specified, boolean similarity function $Sim()$ that returns *true* if two records are similar enough that they may refer to the same entity. $Sim()$ may make use of the results of a mapping from schema to schema [27], and it may encode other complex computations. As we discuss subsequently, many different similarity functions are possible, though the choice of an appropriate similarity function is not the focus of our work. In our example, imagine that $Sim(r_1, r_2)$ returns *true* if and only if $r_1.\texttt{Name} = r_2.\texttt{Name}$. Given the function $Sim()$, the first step in computing the total sum over all employees is to partition $R = R_1 \cup R_2 \cup R_3$ into a set of *equivalence classes*[1]. These equivalence classes are defined so that if there exists a list L where $L = \langle r_1, r_2, ..., r_n \rangle$ and $Sim(r_i, r_{i+1})$ is always *true* for i from 1 to $n-1$, then all of the records in L are in the same equivalence class. Intuitively, if two records are "reachable" from one another in the sense that there is a chain of similar records from one to another, then the two records are said to be in the same equivalence class. In our example, the following are the eight equivalence classes:

$$\begin{aligned}\mathbf{S} = \{&\{(\texttt{Michael}, \texttt{10000}), (\texttt{Michael}, \texttt{9899})\},\\ &\{(\texttt{Christina}, \texttt{7633}), (\texttt{Christina}, \texttt{7412})\},\\ &\{(\texttt{Daniel}, \texttt{7864})\},\\ &\{(\texttt{David}, \texttt{8433})\},\\ &\{(\texttt{Steven}, \texttt{8003})\},\\ &\{(\texttt{Sean}, \texttt{9607})\},\\ &\{(\texttt{Emily}, \texttt{10822})\},\\ &\{(\texttt{James}, \texttt{7322})\}\}\end{aligned}$$

Once R has been partitioned into a set of equivalence classes, then the next step is to apply a reconciliation function $Rec()$ to each equivalence class, and to sum up the results. One way to define $Rec()$ is to take the average of the value to be aggregated over all similar records:

$$\begin{aligned}Rec(\{(\texttt{Michael}, \texttt{10000}),&(\texttt{Michael}, \texttt{9899})\}) =\\ &(10000 + 9899)/2 = 9949.5\end{aligned}$$

Many other functions are reasonable. For example, we can define $Rec()$ to accept the maximum value over all similar records:

$$\begin{aligned}Rec(\{(\texttt{Michael}, \texttt{10000}),&(\texttt{Michael}, \texttt{9899})\}) =\\ &max(10000, 9899) = 10000\end{aligned}$$

[1]Here, "$\cup$" denotes the bag union operation. What we refer to as "equivalence classes" are also referred to as "duplicate clusters" in the literature.

In the first case, the final answer to the query is $9949.5 + 7864 + 8433 + 7572.5 + 8003 + 9607 + 10822 + 7322 = 69523$; in the second case, the final answer is 69684. In either case, given that the similarity function $Sim()$ is equivalent to a check for equality on the `Name` attribute, if the relation R could be materialized in its entirety, the resulting query could easily be written in SQL as:

```
SELECT SUM (GroupTotal) FROM (
 SELECT Rec(r) AS GroupTotal
 FROM R r
 GROUP BY r.Name)
```

This query can be made more general, but for simplicity we will ignore the possibility of including a `WHERE` clause or a `HAVING` clause in the inner query, since this is easy to do in our framework. A `HAVING` clause can easily be incorporated into $Rec()$, and a `WHERE` clause can be handled either at each data site (by simply ignoring records not matching a query predicate) or by incorporation into $Rec()$. Additional `GROUP BY`s (such as returning a separate sum for each of the company's divisions) can be handled by issuing many queries, one for each group. Other aggregation functions (such as `AVERAGE`, `STD_DEV`, and so on) can be seen as special cases of the `SUM` aggregate because they can be written as a function of several `SUM` queries. For example, `AVERAGE` is the ratio of two `SUM` aggregates.

Why is this hard?

This example is simple, but performing the computation in a distributed environment is not. We could simply perform the query locally at each site and add the local totals at a coordinator, but in our example, both `Michael` and `Christina` would be counted twice in the final result. The difficulty of performing this reconciliation in a distributed environment is that *it is virtually impossible to know whether a given record r has external matches under $Sim()$ at other sites without shipping r to an external location to check for a match.* If a query touches millions of records, this shipping of data is not acceptable.

2.2 Problem Definition

The problem we solve is defined as follows:

The Reconcile-Sum (RS) Problem. We are given the following:

- A set of records R distributed over various data sites,
- A similarity function $Sim()$, which takes two records and returns *true* if they are similar, *false* otherwise,
- A reconciliation function $Rec()$, which takes a subset of records of R and returns some numerical value.

Let $\mathbf{S}$ be the set of equivalence classes in R that are induced by $Sim()$. Let M_i denote the sum of $Rec()$ over all classes in $\mathbf{S}$ of size i:

$$M_i = \sum_{S \in \mathbf{S} \wedge |S| = i} Rec(S)$$

Then the goal of the RS Problem is to compute M, which is defined as:

$$M = \sum_{i}^{m} M_i$$

where m is the maximum equivalence class size.

The reader may note that we have broken the computation of M into a number of different compactions, each over a set of classes of the same size. Breaking the computation of M into the computation of a set of M_i values is not necessary when defining the RS problem. However, it is convenient. As we will see in the remainder of the paper, estimating M_i for different values of i can vary widely in terms of the computational cost and/or accuracy. Thus, it will make sense to consider the various M_i's separately.

2.3 The Basic Randomized Solution

Given the computational cost associated with computing the answer to this type of aggregate in a distributed environment, this paper considers several sampling based randomized approaches for computing an approximate answer. Our focus is on sampling due to its generality. Unlike other methods (such as sketches [11]) that are typically constructed to answer a single query, the same sample can be re-used many times for many different queries, and the cost of constructing the sample can be amortized. All of our algorithms are variations on Algorithm 1. The $GetAnswer()$ function of

Algorithm 1: Basic RS Framework

```
Function GetAnswer()
 1. S' = SampleClasses()
 2. M̂ = 0
 3. For i = 1 to m:
    4. M̂_i = 0
    5. For each class S ∈ S' where |S| = i:
       6. M̂_i += (1/p_i) Rec(S)
    7. M̂ += M̂_i
 8. Return M̂
```

Algorithm 1 is run at a designated coordinator site. First, the function $SampleClasses()$ is run to obtain a Bernoulli sample[2] of $\mathbf{S}$, the set of all equivalence classes over R. This requires a distributed algorithm that samples data at all of the sites and returns it to the coordinator (a problem that we consider subsequently). Let the probability of sampling a class of size i be p_i. Then $\hat{M}_i$ as computed in the loop of line (5) is an unbiased estimator for the sum of $Rec()$ applied to all classes of size i. By simply summing over all possible equivalence class sizes, the estimator $\hat{M}$ is then an unbiased estimate for M, the final answer to the RS Problem.

Determining the Accuracy

Not only is this solution fairly simple, it is also easy to describe the accuracy of the resulting estimate. For an unbiased estimator, this is typically done by deriving the estimator's variance:

LEMMA 1. *The variance of $\hat{M}_i$ is*

$$var(\hat{M}_i) = (\frac{1}{p_i} - 1) \sum_{S \in \mathbf{S} \wedge |S| = i} Rec^2(S)$$

[2]A Bernoulli sample is a variable-sized sample where the decision of whether or not to sample an item is made based upon the result of a single, independent coin flip.

For brevity, the proof has been omitted. Because each $\hat{M}_i$ is independent, the next lemma follows immediately:

LEMMA 2. *The variance of $\hat{M}$ is*

$$var(\hat{M}) = \sum_i var(\hat{M}_i)$$

In general, it is not possible to use these formulas to compute $var(\hat{M}_i)$ or $var(\hat{M})$ exactly without access to all of the data. However, we can provide an unbiased estimate as follows:

LEMMA 3. *An estimator of $var(\hat{M}_i)$ is*

$$\hat{var}(\hat{M}_i) = \frac{1}{p_i}(\frac{1}{p_i} - 1) \sum_{S \in \mathbf{S}' \wedge |S|=i} Rec^2(S)$$

LEMMA 4. *An estimator of $var(\hat{M})$ is*

$$\hat{var}(\hat{M}) = \sum_i \hat{var}(\hat{M}_i)$$

Given this estimate for the variance of $\hat{M}$, it is then easily possible to attach confidence bounds to $\hat{M}$ with any applicable method; we choose Central Limit Theorem based bounds. For 95% confidence bounds, one can use $M \in \hat{M} \pm 2 \times var(\hat{M})^{\frac{1}{2}}$; for 99% bounds, the 2 is replaced with a 3. More conservative Chebyshev bounds can also be used.

That's Not All, Is It?

While the basic algorithm is simple, certain technical points require far deeper consideration. The remainder of the paper is devoted to two fundamental questions regarding the algorithm:

1. How can one implement *SampleClasses*() efficiently and correctly compute the required Bernoulli samples over the distributed data?
2. Can more complex methods be used to improve the simple estimator $\hat{M}$?

3. CLASSIFYING THE FUNCTIONS

The answer to the first question depends on the similarity function $Sim()$, and the second depends on the reconciliation function $Rec()$. This section considers at a high level how the two functions answer the aforementioned questions.

3.1 Classifying the Similarity Function

The property of $Sim()$ governing how the sampling should be implemented is whether or not $Sim()$ is transitive. $Sim()$ is *transitive* if and only if given three arbitrary records r_1, r_2, and r_3 it must always be the case that:

$$(Sim(r_1, r_2) = Sim(r_2, r_3) = true) \Rightarrow (Sim(r_1, r_3) = true)$$

A simple equality check on one or more attributes is transitive, as is an equality check preceded by the transformation of all upper-case characters to lower-case. In fact, it can easily be shown that a transitive similarity function is exactly one for which a canonical form for each equivalence class induced by $Sim()$ exists. For example, define the set:

William: {Will, Bill, William, Wm., Billy, Willy, Willie}

All members of this set are names that are often used as synonyms for "William". If $Sim()$ first transforms any occurrence of a value in this set to the canonical form "William" and then checks for equality, $Sim()$ will be transitive.

On the other hand, many similarity functions are not transitive. For example, to allow for misspellings, $Sim()$ may be based upon edit distance. Define $Sim()$ so that it returns *true* if and only if the edit distance between two names is within a cutoff value. It is obvious in this case that both $Sim(r_1, r_2) = true$ and $Sim(r_2, r_3) = true$ cannot guarantee $Sim(r_1, r_3) = true$. Let r_1, r_2, and r_3 be "Tom", "Thom", and "Thomas", respectively. The edit distance between the first two is one, the distance between the last two is two, but the distance between the first and last is three. At a cutoff of two, the first and last records are not similar. However, if all three records appear in S, then r_2 bridges the gap from r_1 to r_3 and all will be members of the same equivalence class.

The Importance of Transitivity

For a transitive similarity function, it is relatively easy to obtain a Bernoulli sample from **S**. We first map each record to its canonical form, and then use the result as an argument to a hash function in order to produce a random value between 0 and 1. If the return value is less than p, the record is sampled and returned to the coordinator. Since all records in an equivalence class map to the same canonical form (and hence to the same random value), either none of the records from a class will be sampled (with probability $(1-p)$), or else they all will (with probability p).

For non-transitive similarity functions, this method does not work because two records in the same equivalence class may be connected via a chain of records in such a way that only adjacent records are similar; thus, membership in an equivalence class is data-dependent. In this case, more complicated methods are needed to obtain a Bernoulli sample, as we discuss subsequently.

3.2 Classifying the Reconciliation Function

We call the analogous, important property of the $Rec()$ function the *size-ratio property*. The presence of this property determines whether we can use a more sophisticated estimator than $\hat{M}$ from Algorithm 1. Formally, the size-ratio property holds if for any equivalence class S in **S**:

$$\frac{\sum_{r \in S} Rec(\{r\})}{Rec(S)} = \rho(|S|)$$

In this equation, ρ is any arbitrary function taking a real-valued input and returning a real-valued output. The size-ratio property requires that the ratio of the sum of the values returned by applying $Rec()$ to each record in S and the value returned by $Rec(S)$ be some function of the size of S. For example, if $Rec()$ uses the average of all aggregate values in an equivalence class, then the size-ratio property holds and $\rho(i) = i$. If $Rec()$ instead returns the maximum value, then the size-ratio property no longer holds.

The Importance of the Size-Ratio Property

If the size-ratio property holds, then it is possible to construct many different estimators for M with little additional computational effort. These can then be combined to produce a single, final estimate which is more accurate than any

constituent. This subsection outlines how this is done; the issue is considered formally in Section 5.

Imagine that we are trying to obtain a better estimate for the answer to the example of Section 2.1. Assume that $Rec()$ simply returns the average aggregate value over an equivalence class, so $\rho(|S|) = |S|$. Let the sampling probability $p_i = .5$ for all i, and imagine that we happen to sample $S_1 = \{(\texttt{Michael}, \texttt{10000}), (\texttt{Michael}, \texttt{9899})\}$, $S_2 = \{(\texttt{Sean}, \texttt{9607})\}$, $S_3 = \{(\texttt{Emily}, \texttt{10822})\}$, and $S_4 = \{(\texttt{Steven}, \texttt{8003})\}$ from $\mathbf{S}$.

Using Algorithm 1, M is estimated as:

$$M \approx \hat{M}_1 + \hat{M}_2 =$$
$$\frac{1}{0.5} \times (avg(9607) + avg(10822) + avg(8003)) +$$
$$\frac{1}{0.5} \times (avg(10000, 9899)) = 56864 + 19899 = 76763$$

Unfortunately, this is not very accurate (recall that the answer for this particular query was 69523).

We can improve this as follows. First, we calculate $W = 10000 + 9899 + 9607 + 10822 + 8003 + 7633 + 7412 + 7864 + 8433 + 7322 = 86995$. W is the overall sum of values returned by applying Rec to each record. W can easily be computed in a distributed fashion by simply scanning the data at each site. Note that if there are data that must be reconciled, then W is not equal to the final query answer M.

We then notice that given ρ, $W = M_1 + 2M_2$ (recall from Section 2.2 that M_i is the total for $Rec()$ applied to all equivalence classes of size i). We also notice that $\hat{M}_2$ is an unbiased guess for M_2. This value is 19899. Thus:$W \approx M_1 + 2 \times 19899$. Or:$M_1 \approx 86995 - 2 \times 19899 = 47197$. Adding this to $\hat{M}_2$ gives us $47197 + 19899 = 67096$ which is another estimate for M. In this particular example, this gives a better guess. In general, we can use this method to build many different estimators. By weighting them properly, we can construct a final ensemble estimate that is far better than any of the constituent individuals.

4. CLASS-WISE BERNOULLI SAMPLING

This section discusses how to implement the $SampleClasses()$ function efficiently and correctly in a distributed environment. As described in Section 3, the implementation of $SampleClasses()$ depends upon the nature of the similarity function $Sim()$, and whether or not it is transitive.

4.1 Sampling With a Transitive Function

The method for sampling from the various equivalence classes given a transitive similarity function has been described previously, and is given formally as Algorithm 2.

This algorithm is invoked with a call to $SampleClasses()$ at the coordinator, with two arguments. The first is the expected fraction of the various equivalence classes that are sampled, and the other is the hash function used to perform the sampling. It is referred to as the "Hash Bernoulli Algorithm" because we use a hash function to perform a Bernoulli sample over all equivalence sets. As discussed in Section 3, this algorithm requires a function $Canonical()$ that is able to transform each record into a single, predesignated member of its equivalence class.

4.2 Sampling With a Non-Transitive Function

Algorithm 2: The Hash Bernoulli Algorithm

```
Function HashSample(p, i, h)
 1. mySam = {}
 2. For each record r at site i:
      /*transfer to canonical form*/
      3. r' = Canonical(r)
      4. If (h(r') < p) then mySam ∪= {r}
 5. Return mySam

Fuction SampleClasses(p, h)
 1. allSam = {}
 2. For each site i:
      3. allSam ∪= HashSample(p, i, h)
 4. Return allSam
```

For non-transitive similarity functions, there is no canonical form for each equivalence class and similarity is data-dependent. In this subsection, we present two different methods for acquiring the required Bernoulli sample of the set of equivalence classes in the non-transitive case. Our first method (described in Section 4.2.1) is quite simple, and is a fairly straightforward modification of a classical algorithm for computing a distributed transitive closure [20]. Unfortunately, there is a significant drawback associated with the simple algorithm: the probability of sampling a given equivalence class increases with the class' size, leading to a sample that is biased towards larger classes. In order to have a reasonable probability of sampling one of the smallest classes, we may have a probability that is almost one of sampling each member of the larger classes. In the worst case, this requires that each site ship the majority of its records to the coordinator. To avoid this and reduce the cost of computing the distributed transitive closure, in Section 4.2.2 we describe an alternative algorithm that can avoid most of the computation associated with any very large classes that may exist in the data.

4.2.1 The Uniform-p Algorithm

The first algorithm accepts as a parameter a sampling probability of p, and samples each record with this probability. All sampled records are then shipped to a coordinator, who computes a transitive closure over each sampled record. The pseudo-code is given as Algorithm 3. This algorithm is invoked with a call to $GetSamples()$ at the coordinator, with an argument that is the expected fraction of records that are sampled the first time around at each site. It is referred to as the "Uniform-p Algorithm" because the probability of sampling each record is uniform.

After the sampling phase, the coordinator sends those samples off to every site in turn. Each site determines if it has any records "reachable" from the set of samples, and returns those to the coordinator. This process is repeated with any newly discovered records until it is impossible to find any similar records. In this way, once the while loop on line (6) of $SampleClasses()$ finishes, $allSam$ will have all records "reachable" from any sampled record via the similarity function $Sim()$. A simple transitive closure over this set then returns a sample $\mathbf{S}'$ of the set of all equivalence classes $\mathbf{S}$, where the probability of sampling a class containing i records is $1 - (1-p)^i$ because the only way in which a class cannot be sampled is if *all* of its members are missed.

Algorithm 3: The Uniform-p Algorithm

```
Function Sample(p, i)
 1. mySam = {}
 2. For each record r at site i:
    3. With probability p, mySam ∪= {r}
 4. Return mySam

Function Extend(cur, i)
 1. res = {}
 2. For each record r at site i:
    3. If Sim(r, s) = true for some s ∈ cur
       4. res ∪= {r}
       5. remember r so it is never considered in fut-
          ure calls to Extend
 6. Return res

Function SampleClasses(p)
 /* first, sample the sites */
 1. allSam = {}
 2. For each site i:
    3. allSam ∪= Sample(p, i)
 /* next, get all records needed for closure */
 4. newRec = allSam
 5. nextRec = {}
 6. While (|newRec| > 0):
    7. For each site i:
       8. nextRec ∪= Extend(newRec, i)
    9. allSam ∪= nextRec
    10. newRec = nextRec
 11. Return the transitive closure of allSam under Sim
```

Several obvious optimizations of the algorithm are possible. For example, the local transitive closure of all of the records in $mySam$ can be computed and added to $mySam$ before $mySam$ is returned at step (4) of the $Sample()$ function. This will save iterations of the algorithm. Likewise, the function $Extend()$ can also include all records similar to s in the set res in line (4) of the $Extend()$ function. Finally, the search in line (3) of the $Extend()$ function can be speeded using any of a number of methods [4, 30, 31, 24].

This instantiation of $SampleClasses()$ can be used seamlessly within Algorithm 1.

4.2.2 The Diminishing-p Algorithm

A problem with the Uniform-p Algorithm is that the sampling probability for an equivalence class increases almost linearly with the size of the group. Since the cost associated with sampling a group increases super-linearly with group size, most of the algorithm's computational effort may be directed at the most expensive groups. This subsection presents an algorithm that does not suffer from this problem, called the Diminishing-p Algorithm.

The key difference between the Uniform-p Algorithm and the Diminishing-p Algorithm is that each originally-sampled seed record is labeled as "active" or not. All of the originally-sampled seed records start out active, but become inactive in a probabilistic fashion; at each iteration, each active record has a probability q of staying active. At the i^{th} iteration of the algorithm, the only classes that we try to extend are those that are currently of size i and still contain active seed records. If a class becomes inactive before we are "done" with it (that is, if the class is of size larger than i, but all of its seed records become inactive before the i^{th} iteration of the algorithm), then the class is discarded.

Algorithm 4: The Diminishing-p Algorithm

```
Function DeactivateSome(active, q)
 1. res = {}
 2. For each record r ∈ active:
    3. With probability q, res ∪= {r}
 4. Return res

Function SampleClasses(p, q)
 /* first, sample the sites */
 1. allSam = {}; ans = {};
 2. For each site i:
    3. allSam ∪= Sample(p, i)
 /* next, get all records needed for closure */
 4. active = allSam
 5. newRec = allSam
 6. For i = 1 to ∞:
    7. checkEm = {}
    8. Compute the transitive closure S' of allSam
       under Sim
    /* see if we are working on any class w active recs */
    9. If there is no S ∈ S' such that
       |S| ≥ i and S ∩ active ≠ {}:
          /* we are not working on any such classes*/
       10. Break
    /* find any records which could make an active,
     size i class larger */
    11. For each set S ∈ S' where |S| = i
       12. If S ∩ active ≠ {}:
             13. checkEm ∪= (S ∩ newRec)
             14. newRec −= (S ∩ newRec)
    /* try to make the active size i class larger */
    15. For each site j:
       16. newRec ∪= Extend(checkEm, j)
    /* find each active size i class not made larger */
    17. For each set S ∈ S' where |S| = i
       18. If S ∩ active ≠ {} and there is no
          s ∈ S, r ∈ newRec where Sim(s, r):
             /* these are added to the answer set */
          19. ans ∪= S
    20. allSam ∪= newRec
    21. active = DeactivateSome(active, q)
 22. Return ans
```

In this way, we can arbitrarily reduce the natural bias towards sampling larger classes. While we still have a greater chance of sampling seed records from a larger equivalence class, the bias induced by this tendency is counteracted by the chance that we probabilistically get "tired" of trying to compute all of the members of a larger class before we finish. This means that we can choose a large p value to ensure that we sample many smaller classes, and we can choose a small q value to ensure that we only sample a reasonable number of larger classes, thus avoiding shipping too many records to the coordinator.

Since this algorithm is fairly complicated, it is worthwhile discussing some of its specifics. There are four important sets maintained by $SampleClasses()$: $allSam$, ans, $active$, and $newRec$. In order, these are: all of the records that the coordinator has obtained from any local site, the set of all sampled equivalence classes that we are building, the set of active seed (originally sampled) records, and the set of records at the "frontier" of the transitive closure computation. That is, $newRec$ is the set of all records that are reachable from some seed record. Furthermore, if a record is in $newRec$, we have not yet sent it off to a local site to

see if it can be used to extend an equivalence class.

The main loop of line (6) of *SampleClasses*() controls the size of the equivalence classes that we are currently dealing with. For example, if $i = 5$, then this means that we are done sampling any equivalence class of size 5 or less, and will try to extend any equivalence class that is currently size 5. Line (9) checks if it is still possible to find any equivalence class of size i or larger. If it *is* possible, the loop at line (11) checks which records in *newRec* could be used to grow a class of size i into a larger class. The set *checkEm* holds the records from *newRec* that could be used for this task. Line (15) then sends all of the records from *checkEm* off to the local sites. Line (17) checks to see if we were unable to extend the size of any active classes of size i. If we were, then we are done with these classes and they are added to the final set of samples. Finally, line (21) of the algorithm randomly removes some of the records from the set *active*. Any still-growing equivalence classes with no active seeds will not be sampled.

Producing an Estimate. The parameters p and q govern how many active seeds are deactivated at each step of the algorithm, and thus affect the value of p_i that must be used in conjunction with the Diminishing-p Algorithm to produce an unbiased estimate. To derive a formula for p_i, assume that equivalence class S is of size i, and let *allSam* be the set of records sampled using the original, Bernoulli sampling process. Finally, let $active_i$ denote the set of records active at the beginning of iteration i of the algorithm. Then:

$$p_i = Pr[S \in ans]$$
$$= \sum_{j=1}^{i} Pr[|S \cap allSam| = j] \times Pr[active_i \cap S \neq \{\} | j]$$

Note that:

$$Pr[active_i \cap S \neq \{\} | j] = 1 - Pr[active_i \cap S = \{\} | j]$$
$$= 1 - (1 - Pr[(s \in S) \in active_i])^j = 1 - (1 - q^{i-1})^j \quad (1)$$

Let $binom(j, i, p)$ denote the binomial probability associated with achieving j out of i successes with a success probability of p. Since *allSam* is sampled using a Bernoulli sampling scheme, $Pr[|S \cap allSam| = j] = binom(j, i, p)$. This gives us:

$$p_i = \sum_{j=1}^{i} binom(j, i, p) \times (1 - (1 - q^{i-1})^j) \quad (2)$$

Choosing p and q. If one has access to good statistics over the data, then the choice of p and q is an optimization problem: choose p and q so as to maximize the accuracy (minimize the variance) of the resulting estimator given a constraint on the amount of data that is to be communicated to the coordinator. In the absence of this information, we use the following heuristic. We begin with the assumption that the variance of *Rec*() does not vary much across equivalence classes of differing sizes and we assume that the number of instances of each equivalence class size is uniform. If we view each equivalence class as a stratum, then the so-called Neyman sampling allocation (Page 471 of Sarndal et al. [32]) will prescribe a uniform number of samples to each stratum to reduce the variance of the resulting estimator. Thus, given p, we then choose q so that the selection probability across classes is as uniform as possible. If we have a rough guess for m (the size of the largest equivalence class), we can attempt this by using numerical methods to choose q so as to minimize the expression:

$$\sum_{i=2}^{m} (p - p_i)^2$$

5. OPTIMIZED ESTIMATORS

The alternative estimator described in this section is based upon the observation that if the reconciliation function that is used has the size-ratio property, then we can easily build *many* different estimators for the final query answer. By combining them in an optimal fashion, it is easy to produce a combined estimator that is always better than the simple one from Section 2. Notice that this is totally orthogonal to the issue of how to compute the Bernoulli sample.

5.1 The Basic Idea

The estimator described in this section is based upon the observation that if the size-ratio property holds, it is easily possible to remove any single $\hat{M}_i$ from the estimator $\hat{M} = \sum_{i=1}^{m} \hat{M}_i$ used by Algorithm 1. To do this, we first compute W as defined in Section 3.2. Recall that W is a simple sum of the reconciliation function applied to each tuple in the system without using the similarity function to perform a grouping. This is easily computed via a single pass through the data at each site. If the size-ratio property holds, then W is nothing more than a weighted sum over each M_i, where $\rho(i)$ is the weight for M_i. For example, if the reconciliation function computes the average of each equivalence class, then $W = \sum_{i=1}^{m} i \times M_i$. This means that we can write any M_i in terms of W and the other M_i values; for example:

$$M_1 = W - \sum_{i=2}^{m} i \times M_i$$

This allows us to define an unbiased estimator for any M_j in terms of the various $\hat{M}_i$ ($i \neq j$) estimators; for example:

$$\hat{M}'_1 = W - \sum_{i=2}^{m} i \times \hat{M}_i$$

Then by replacing any $\hat{M}_i$ in $\sum_{i=1}^{m} \hat{M}_i$ with the corresponding estimator $\hat{M}'_i$, we can obtain an entirely new estimate for the final query answer; the new estimator obtained be replacing $\hat{M}_i$ is denoted by E_i (for consistency, we define E_0 to be equivalent to $\hat{M}$). In the remainder of this section, we describe how to use this basic method to create an entire *ensemble* of estimators in this fashion.

5.2 The Optimized-Estimator Algorithm

Algorithm 5 formalizes the process. Just like Algorithm 1, it is invoked with a call to *GetAnswer*(). Algorithm 5 describes exactly how each individual estimate E_i is computed, but leaves out a few key details regarding how each of the E_is are combined to produce the final estimate E. Specifically, the questions that we have yet to answer are:

- How does the *GetOptWeights*() function determine each α_i? Ideally, each α_i would be computed so as to minimize the variance of the estimator $E = \sum_{i=0}^{m} \alpha_i E_i$, subject to the constraint that $\sum_{i=0}^{m} \alpha_i = 1$.

Algorithm 5: The Optimized-Estimator Algorithm

Function $CompEi(i, \mathbf{S}', W)$
1. Calculate each $\hat{M}_j$ as in Algorithm 1
2. If $i = 0$:
   3. $res = \sum_j \hat{M}_j$
4. Else
   5. $\hat{M}'_i = \frac{W - \sum_{k \neq i} \rho(k) \hat{M}_k}{\rho(i)}$
   6. $res = \frac{W}{\rho(i)} + \sum_j (1 - \frac{\rho(j)}{\rho(i)} \hat{M}_j)$
7. Return res

Function $GetW(i)$
1. For each record r at site i:
   2. $W \ += Rec(\{r\})$
3. Return W

Function $GetAnswer()$
1. $\mathbf{S}' = SampleClasses()$
2. $W = 0$
3. For each site i:
   4. $W \ += GetW(i)$
5. For each $i \in \{0...m\}$:
   6. $E_i = CompEi(i, \mathbf{S}', W)$
/* α is a vector of weights */
7. $\alpha = GetOptWeights()$
8. $E = 0$
9. For each $i \in \{0...m\}$:
   10. $E \ += \alpha_i E_i$
11. Return E

- What exactly *is* the variance of the resulting estimator? Ideally, we would be able to obtain a result analogous to Lemmas 2 and 4 that would allow us to estimate the variance of E, which in turn would allow us to attach confidence bounds to the estimate.

5.3 Computing the Optimal Weights

In general, we can represent every E_i as follows:

$$E_i = \beta_{i0} W + \sum_{j=1}^{m} \beta_{ij} \hat{M}_j$$

β_{ij} is the coefficient for $\hat{M}_j$ in the i^{th} estimator, and β_{i0} is the coefficient of W in the i^{th} estimator. These are computed as described by the $CompEi()$ function in Algorithm 5.

As defined in Algorithm 5's $GetAnswer()$ function:

$$E = \sum_{i=0}^{m} \alpha_i E_i = \sum_{i=0}^{m} \alpha_i (\beta_{i0} W + \sum_{j=1}^{m} \beta_{ij} \hat{M}_j)$$
$$= \sum_{i=0}^{m} \alpha_i \beta_{i0} W + \sum_{j=1}^{m} \sum_{i=0}^{m} \alpha_i \beta_{ij} \hat{M}_j$$

To choose the optimal α_i values, the goal is to minimize the variance of E subject to $\sum_{i=0}^{m} \alpha_i = 1$. To do this, we first divide the expression for E into two parts: one part that is related to W, and one part that is related to each $\hat{M}_j$. Since the part that is related only to W takes a constant (non-random) value, it can be ignored during the variance calculation. We also notice that for any $j \neq k$, $\hat{M}_j$ and $\hat{M}_k$ are independent and so the covariance between them is always 0. This gives us:

$$var(E) = var(\sum_{j=1}^{m} (\sum_{i=0}^{m} \alpha_i \beta_{ij}) \hat{M}_j)$$
$$= \sum_{j=1}^{m} var((\sum_{i=0}^{m} \alpha_i \beta_{ij}) \hat{M}_j) = \sum_{j=1}^{m} (\sum_{i=0}^{m} \alpha_i \beta_{ij})^2 var(\hat{M}_j) \quad (3)$$

Note that the variance of each $\hat{M}_j$ can easily be estimated using Lemma 3.

Since we seek to minimize this subject to $\sum_{i=0}^{m} \alpha_i = 1$, we have an optimization problem that is ideally suited to the use of a Lagrangian multiplier λ [22]. Using this method, our objective function becomes:

$$L = var(E) - \lambda(\sum_{i=0}^{m} \alpha_i - 1) = \sum_{j=1}^{m} (\sum_{i=0}^{m} \alpha_i \beta_{ij})^2 var(\hat{M}_j) - \lambda(\sum_{i=0}^{m} \alpha_i - 1)$$

In order to solve the resulting problem, we take the derivative with respect to each α_i and λ, and set the resulting $m + 1$ equations equal to 0. To do this, notice we can actually rewrite the formula for $var(E)$ as follows:

$$var(E) = \sum_{j=1}^{m} (\sum_{i=0}^{m} \alpha_i \beta_{ij})^2 var(\hat{M}_j) = \sum_{j=1}^{m} var(\hat{M}_j) (\sum_{k=0}^{m} \alpha_k \beta_{kj} \sum_{l=0}^{m} \alpha_l \beta_{lj})$$
$$= \sum_{j=1}^{m} var(\hat{M}_j) (\sum_{k=0}^{m} \sum_{l=0 \wedge l \neq k}^{m} \alpha_k \alpha_l \beta_{kj} \beta_{lj} + \sum_{k=0}^{m} \alpha_k^2 \beta_{kj}^2)$$

Using this form of $Var(E)$ and differentiating L with respect to any arbitrary α_i we have:

$$\frac{\partial L}{\partial \alpha_i} = \frac{\partial}{\partial \alpha_i} (\sum_{j=1}^{m} var(\hat{M}_j) (\sum_{k=0}^{m} \sum_{l=0 \wedge l \neq k}^{m} \alpha_k \alpha_l \beta_{kj} \beta_{lj}$$
$$+ \sum_{k=0}^{m} \alpha_k^2 \beta_{kj}^2) - \lambda \sum_i \alpha_i)$$
$$= \sum_{j=1}^{m} var(\hat{M}_j) (2 \sum_{k=0 \wedge k \neq i}^{m} \beta_{ij} \beta_{kj} \alpha_k + 2 \beta_{ij}^2 \alpha_i) - \lambda$$
$$= \sum_{j=1}^{m} var(\hat{M}_j) (2 \sum_{k=0}^{m} \beta_{ij} \beta_{kj} \alpha_k) - \lambda$$
$$= \sum_{k=0}^{m} (2 \sum_{j=1}^{m} \beta_{ij} \beta_{kj} var(\hat{M}_j)) \alpha_k - \lambda = 0$$

Differentiating L with respect to λ we have:

$$\frac{\partial L}{\partial \lambda} = \sum_{i=0}^{m} \alpha_i = 0$$

By simultaneously solving the $m + 1$ equations in the resulting linear system, we can obtain a set of optimized coefficients as required by $GetAnswer()$ in Algorithm 5. Furthermore, by using equation 3, it is easy to calculate the variance of the resulting estimator, which can be used to obtain confidence bounds, as described in Section 2.3.

6. EXPERIMENTS

There are three specific questions that our experiments are designed to answer:

1. First, is the theory behind our algorithms sound? Do our estimators behave as advertised? Are the various variance computations described in the paper actually correct, and can they be used in conjunction with central-limit-theorem-based confidence bounds in order to accurately predict the accuracy of the estimate?

2. Second, how accurate are these estimators as a function of the data characteristics? Does the use of the more complex estimator enabled by the size-ratio property allow for more accurate estimation?

3. Finally, what is the effect of the sampling strategy on the efficacy of the algorithms? Does simple sampling using a transitive similarity function allow for higher accuracy? How do the algorithms for dealing with a non-transitive similarity function compare?

6.1 Data Generation and Basic Setup

Unless otherwise stated, all experiments are performed on data generated using the following algorithm, parameterized on *size*, *shape*, *scale*, μ, and σ^2. The parameter *size* controls the number of records in the distributed system that are relevant for answering a specific query. Until the data set generated is of size *size*, the following steps are repeated:

1. An equivalence class size is generated by taking a random sample from a gamma distribution (Page 99 of Casella [3]) with shape parameter *shape* and scale parameter *scale*; the result is then rounded up. The gamma distribution can model a large number of data characteristics: from the case where every equivalence is of size one, to the case where most are size one but some are huge, to the case where most are large. For example, if *shape* = 1 and *scale* = 4, then 95% of the classes fewer than 12 members, and 22% are of size 1.

2. A equivalence class mean μ' is generated from a normal distribution with mean μ and variance σ^2.

3. For each record in this equivalence class, an aggregate value is generated using a sample from another normal distribution with mean μ' and variance σ^2.

4. Finally, each record in the equivalence class is randomly sent to one of five data sites.

Due to space constraints, all queries tested are `SUM` queries without selection predicates. We remind the reader that (as discussed in Section 2.1) `SUM` queries can be used to encode more complicated aggregates and queries, hence our focus. In all cases, the function *Rec*() computes an average over all records in the class. We use the same *Rec*() for both the optimized and non-optimized estimators in order to make the results comparable. Since we do not measure wall-clock computation time, the exact similarity function *Sim*() is not important; the important property is whether or not it is transitive. This will vary from experiment to experiment.

6.2 Statistical Properties of the Estimators

Experimental Setup. In this subsection, we describe an experiment designed to test the correctness of the estimators used for the statistical properties of our algorithms. In all, six different combinations of algorithms were proposed, depending on whether the simple or optimized estimator was used, and whether or not the function *Sim*() is transitive (if not, then there are two separate sampling options: the Uniform-p Algorithm and the Diminishing-p Algorithm). All six combinations are tested in this subsection.

To check whether our algorithms are correct, we first generate a data set using arguments *size* = 10^7, *shape* = 1, *scale* = 4, $\mu = 1$, and $\sigma^2 = 1$. Then we test each of the six algorithms, in turn. A 1% sampling fraction is used. For each algorithm, we repeat the sampling and estimation process 500 times. For each of the 500 trials, the estimator's variance is estimated, and a 95% confidence bound is computed using an assumption of normality.

In Figure 1, for each algorithm, we plot the fraction of the 500 trials where the confidence bound reported was correct. In Figure 2 we plot a histogram of all 500 variance estimates, as well as the actual observed variance over the 500 trials (this is given as a dotted line) and the mean variance estimate over the 500 trials (as a thick solid line).

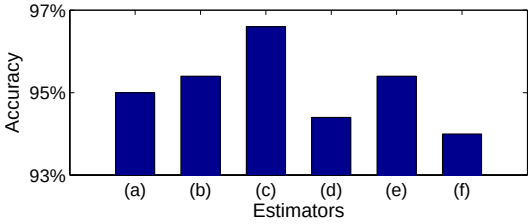

Figure 1: Observed coverage of the 95% confidence intervals for the six combinations of algorithms: (a) non-optimized, transitive; (b) optimized, transitive; (c) non-optimized, Uniform-p; (d) optimized, Uniform-p; (e) non-optimized, Diminishing-p; (f) optimized, Diminishing-p.

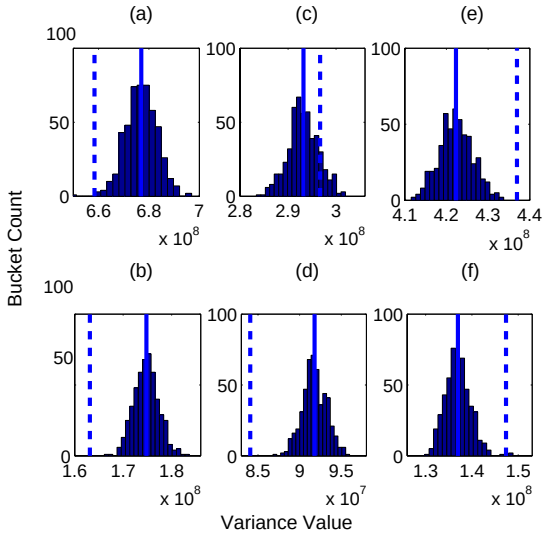

Figure 2: Accuracy of the variance estimator for each of the six algorithm combinations.

Discussion. Figure 1 seems to show conclusively that the

confidence bounds for all six estimators are in fact very reliable. The true coverage rate is always very close to the specified 95%. Note that this experiment is designed only to test the *reliability* of the various estimators. This experiment shows that there is virtually no difference between them. As we will show in the next subsection, the *accuracy* of the estimators (that is, the ability of the various estimators to provide for narrow confidence intervals) will differ significantly with the optimized estimators giving much narrower bounds.

Figure 2 examines the variance estimation in depth. All variance estimates are fairly accurate, and estimation errors of more than 10% are fairly rare. In general, the non-optimized variance estimates are all more accurate than the optimized variance estimates. This is not surprising, given that the optimized estimator *itself* is parameterized on the variance estimates. As a result, the optimized estimator will often magnify any inaccuracy in a variance estimate by weighting equivalence classes with an inaccurate estimate in an especially heavy or light fashion. For the non-optimized variance estimates, the difference between the average of the computed variance and the observed variance is statistically insignificant.

6.3 Estimator Accuracy

In this section, we study the effect of the data generation parameters on estimator accuracy. For simplicity, we test only the Hash Bernoulli sampling algorithm (the other sampling algorithms will be considered subsequently).

Experimental Setup. We run a number of different experiments to test both the accuracy of the simple estimator, as well as the accuracy of the optimized estimator making use of the size-ratio property. We run four different tests:

1. First, we hold the sampling fraction constant at 1%, and use $size = 10^7$, $scale = 1$, $\mu = 1$, and $\sigma^2 = 1$. We then vary the gamma shape parameter from zero to one. With a shape that is close to zero, there are no equivalence classes that have a size larger than one. With a shape of 0.5, a few percent of them do, and with a shape of 1, around 60% of the classes are of size larger than one. The results are given as "Width vs. Shape" in Figure 3. For each shape value, and both of the estimators, we compute the width of a 95% confidence interval as a fraction of the interval's mean value. Thus, a small ratio indicates a tight bound.

2. Next, we hold the shape constant at 1. Using the same parameter settings, we vary the sampling fraction from 0.1% to 10%. The results are given as "Width vs. Sampling Ratio" in Figure 3.

3. Next, we again hold the sampling fraction constant at 1%, but vary the variance parameter σ^2 from zero to ten. The results are given as "Width vs. Variance".

4. Finally, we use $scale = 1$, $shape = 1$, $\mu = 1$, and $\sigma^2 = 1$, and vary the database size. However, we hold the *absolute number* of samples constant at 10000 so that (sampling frac) $= 10000/size$.

We also run one more experiment designed to test the effect of correlation between the size of an equivalence class and its statistical properties. To do this, we generate 10^7 records. The i^{th} record is generated using a normal variable with a mean and variance $11 - (i/10^6)$. Equivalence class sizes are then generated as before, with $shape = 1$ and $scale = 1$. The equivalence classes are then sorted from smallest to largest. The first record generated is assigned to the smallest equivalence class (which is almost surely of size one). All records are then assigned in sequence from smallest equivalence class to largest equivalence class. The statistical properties of the records within an equivalence class are then correlated with class size.

In order to de-correlate class size and the properties of the class, we can randomly swap records before we assign the records to equivalence classes. If 50% of the records are swapped, then half of the records will be in random positions and there will be far less correlation. If all of the records are swapped, there is no correlation. Figure 4 shows the accuracy of the two estimators as a function of the fraction of records swapped before they are assigned to classes.

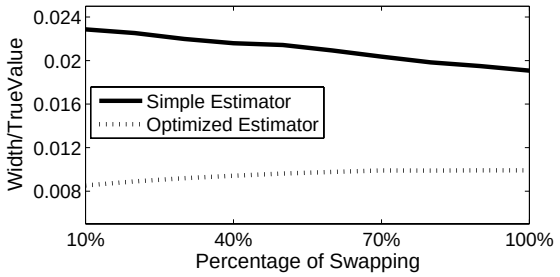

Figure 4: Estimator accuracy as a function of the correlation between equivalence class size and the statistical properties of the class.

Discussion. Several interesting results can be observed. We point out a few significant ones here.

First, we see that the optimized estimator has a zero-width confidence bound when there are no duplicates in the data. Its accuracy then degrades gracefully as the number of duplicates in the database increases. This is a very desirable property, because it is expected that in many applications, few or no records requiring reconciliation will be found. In this case, the optimized estimator is equivalent to simply computing an aggregate locally and shipping just those aggregates back to the coordinator.

Second, we point out that the accuracy of both estimators is related only to the sample size and not the sampling fraction, assuming that all other parameters are held constant. This means that the algorithms do not require more data to be shipped to site to maintain a desired accuracy as the database size grows.

Finally, we note that the accuracy of the optimized estimator actually *decreases* with decreasing correlation between equivalence class size and statistical properties of the class. This can be explained because when equivalence classes are particularly poorly-behaved, the optimized estimator can effectively choose to "ignore" them during the estimation process. The simple estimator performs worse in the highly correlated case because the different equivalence classes are no longer "interchangeable". An error in the small classes with large mean and large variance is more probable and damaging than if all classes behave similarly.

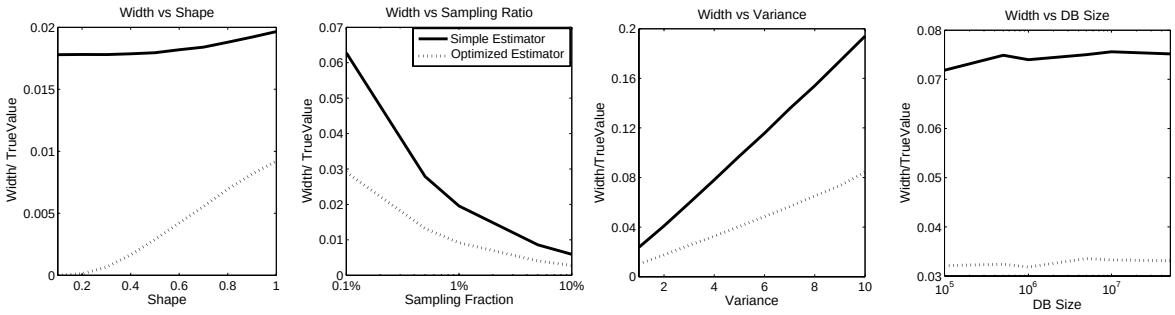

Figure 3: Estimator accuracy as a function of data and sample size characteristics.

6.4 Cost Comparison

Our final set of comparisons is designed to test the merits of the various sampling algorithms: the simple, Hash Bernoulli algorithm for transitive $Sim()$, and the more complicated algorithms for sampling equivalence classes in the non-transitive case. In general, the quality of these algorithms is best measured in terms of the number of records that must be communicated from site-to-site in order to produce an estimate of a certain accuracy.

Experimental Setup. We prepare two datasets for this experiment. The first uses a sampling fraction of 1%, and $size = 10^7$, $scale = 4$, $shape = 1$, $\mu = 1$, and $\sigma^2 = 1$. The second one is identical, except that it uses $scale = 2$. In the first case, 95% of the equivalence classes are sized fewer than 12, and 22% of the classes are size one. In the second case, 95% of the classes have fewer than six elements and 44% of the classes are size one.

We then run each of the six algorithms on both of the data sets (three sampling algorithms × two estimators). For each, we test a number of different sampling fractions. For each combination, we compute the number of records that must be transferred to or from the coordinator, as well as the accuracy of the resulting estimator. This results in a number of (records transferred, accuracy) combinations for each of the six algorithms that are plotted in Figure 5.

Discussion. In all cases, the Hash Bernoulli sampling algorithm is the lowest cost sampling algorithm for a fixed-accuracy estimate. This is not surprising; for a specific equivalence class, it is computationally trivial to obtain all of the records in an equivalence class if the transitive property holds, and no back-and-forth communication is required.

Perhaps most interesting is the relationship between the gamma scale parameter and the relative utility of the Uniform-p algorithm compared to the Diminishing-p algorithm. At $scale = 2$, the two algorithms show almost identical performance. This is not surprising given that there are relatively few duplicates in this data set. However, when $scale = 4$, there are a number of larger equivalence classes; 5% of them exceed size 12. In this case, the Uniform-p algorithm begins to show significant bias towards computational effort for these classes.

7. RELATED WORK

Data integration [17] has been an important research topic; a few well-known projects in this area are Garlic [21], Clio [25], TSIMIIS [6], Piazza [16], and Hyperion [28], just to name a very few. Many of the sub-areas of this field of study such as schema mapping and record linkage are related to the topic of this paper, but are orthogonal in the sense that our algorithms assume that appropriate methods are available and have been selected. Our work has assumed a federated or mediated integration approach. Thus, we assume that a mapping between the various data sources is available. Many methods have been proposed for computing such a mapping, and we cite a few [10, 18, 2]. We also assume that a method is available for dealing with the so-called "record-linkage problem" that arises when checking whether two records refer to the same entity. This functionality is embedded within the $Rec()$ function. This has been an active research area; several approaches have been proposed by the data management community [23, 5, 29, 8, 12]. In the case of a non-transitive similarity function, our Uniform-p and Diminishing-p algorithms assume that an efficient algorithm exists for matching similar records at a specific data site according to the function $Sim()$. This problem has also been tackled [4, 30, 31]; see Koudas and Srivastava's tutorial on this subject [24].

Approximation has been studied widely in the data management literature [26, 19]. The closest work to our own is that of Cormode et al. [9], who considered the problem of continuous monitoring for duplicate-resilient aggregates over a distributed data stream. They maintain an approximate count of the number of distinct values in the stream using a Flajolet-Martin Sketch [13]. They also maintain a distinct sample using Gibbons' distinct sampling algorithm [15]. However, Cormode et al. solve quite a different problem, where "duplicates" are assumed to be identical records; their methods are not applicable to arbitrary similarity and reconciliation functions. We also point out that Gibbons' distinct sampling algorithm is quite different from our own sampling algorithms, in that our goal is to sample *all* of the records from an equivalence class induced by $Sim()$ in a distributed fashion.

8. CONCLUSION

This paper has considered the computational issues associated with handling analytic queries over widely distributed data, where there is a concern that individual records may be repeated, possibly with errors, over multiple data sites. Our algorithms are parameterized on a similarity function and a reconciliation function that are user-defined, making them extremely general. There are several avenues for future work. The most obvious is the extension of our algorithms

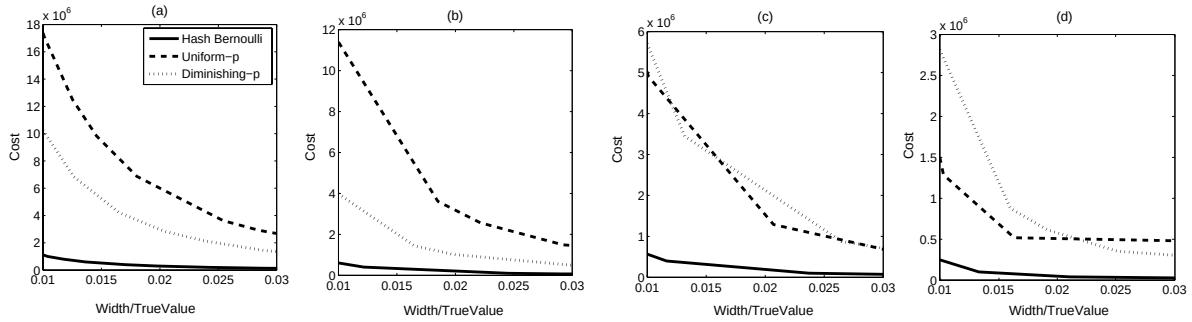

Figure 5: Number of records that must be communicated in order to achieve a desired accuracy. (a) shows results for non-optimized estimators over data with $scale = 4$. (b) shows optimized estimators for $scale = 4$. (c) and (d) show non-optimized and optimized estimators, respectively, for $scale = 2$.

to more complex queries that may include join or relational subtraction operations.

9. REFERENCES

[1] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79, 1999.

[2] P. A. Bernstein, S. Melnik, M. Petropoulos, and C. Quix. Industrial-strength schema matching. *SIGMOD Record*, 33(4):38–43, 2004.

[3] G. Casella and R. L. Berger. *Statistical Inference*. Duxbury Press, second edition, 2001.

[4] A. Chandel, P. C. Nagesh, and S. Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE*, page 28, 2006.

[5] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, pages 313–324, 2003.

[6] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *IPSJ*, pages 7–18, Japan, 1994.

[7] J. Chomicki, J. Marcinkowski, and S. Staworko. Hippo: A system for computing consistent answers to a class of sql queries. In *EDBT*, pages 841–844, 2004.

[8] W. W. Cohen and S. Sarawagi. Exploiting dictionaries in named entity extraction: combining semi-markov extraction processes and data integration methods. In *KDD*, pages 89–98, 2004.

[9] G. Cormode, S. Muthukrishnan, and W. Zhuang. What's different: Distributed, continuous monitoring of duplicate-resilient aggregates on data streams. In *ICDE*, page 57, 2006.

[10] A. Doan, P. Domingos, and A. Y. Halevy. Learning to match the schemas of data sources: A multistrategy approach. *Machine Learning*, 50(3):279–301, 2003.

[11] A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *SIGMOD Conference*, pages 61–72, 2002.

[12] M. Elfeky, A. Elmagarmid, and V. Verykios. Tailor: A record linkage tool box. In *ICDE*, pages 17–28, 2002.

[13] P. Flajolet and G. N. Martin. Probabilistic counting. In *FOCS*, 1983.

[14] A. Fuxman, E. Fazli, and R. J. Miller. Conquer: Efficient management of inconsistent databases. In *SIGMOD Conference*, pages 155–166, 2005.

[15] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB-J*, pages 541–550, 2001.

[16] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The piazza peer data management system. *TKDE*, 16(7):787–798, 2004.

[17] A. Y. Halevy, A. Rajaraman, and J. J. Ordille. Data integration: The teenage years. In *VLDB*, pages 9–16, 2006.

[18] B. He and K. C.-C. Chang. Statistical schema matching across web query interfaces. In *SIGMOD Conference*, pages 217–228, 2003.

[19] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD Conference*, pages 171–182, 1997.

[20] M. Houtsma, P. Apers, and S. Ceri. Distributed transitive closure computations: The disconnection set approach. In *VLDB*, pages 335–346, 1990.

[21] V. Josifovski, P. M. Schwarz, L. M. Haas, and E. T. Lin. Garlic: a new flavor of federated query processing for db2. In *SIGMOD Conference*, pages 524–532, 2002.

[22] D. Klein. Lagrange multipliers without permanent scarring. In *http://www.cs.berkeley.edu/ klein/papers/lagrange-multipliers.pdf*.

[23] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, pages 802–803, 2006.

[24] N. Koudas and D. Srivastava. Approximate joins: Concepts and techniques. In *VLDB*, page 1363, 2005.

[25] R. J. Miller, M. Hernandez, L. M. Haas, L.-L. Yan, C. T. H. Ho, R. Fagin, and L. Popa. The clio project: Managing heterogeneity. *SIGMOD Record*, 30(1):78–83, March 2001.

[26] F. Olken. *Random Sampling from Databases*. PhD thesis, UC Berkeley, 1993.

[27] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal: Very Large Data Bases*, 10(4):334–350, 12 2001.

[28] P. Rodríguez-Gianolli, M. Garzetti, L. Jiang, A. Kementsietsidis, I. Kiringa, M. Masud, R. J. Miller, and J. Mylopoulos. Data sharing in the hyperion peer database system. In *VLDB*, pages 1291–1294, 2005.

[29] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *KDD*, pages 269–278, 2002.

[30] S. Sarawagi and A. Kirpal. Scaling up the alias duplicate elimination system. In *ICDE*, pages 783–785, 2003.

[31] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.

[32] C. Sarndal, B. Swensson, and J. Wretman. *Model Assisted Survey Sampling*. Springer, New York, 1992.