

# Online Estimation For Subset-Based SQL Queries \*

Christopher Jermaine

Alin Dobra

Abhijit Pol

Shantanu Joshi

Department of Computer and Information Science and Engineering  
University of Florida  
Gainesville, FL, 32611 USA  
{cjermain,adobra,apol,ssjoshi}@cise.ufl.edu

## Abstract

The largest databases in use today are so large that answering a query exactly can take minutes, hours, or even days. One way to address this problem is to make use of approximation algorithms. Previous work on online aggregation has considered how to give online estimates with ever-increasing accuracy for aggregate functions over relational join and selection queries. However, no existing work is applicable to online estimation over *subset-based SQL* queries—those queries with a correlated subquery linked to an outer query via a NOT EXISTS, NOT IN, EXISTS, or IN clause (other queries such as EXCEPT and INTERSECT can also be seen as subset-based queries). In this paper we develop algorithms for online estimation over such queries, and consider the difficult problem of providing probabilistic accuracy guarantees at all times during query execution.

## 1 Introduction

Despite the best efforts of software and hardware designers, the largest data warehouses are now so massive that it is impossible to guarantee interactive speeds when answering ad-hoc, analytic style queries. A close examination of the latest TPC-H benchmark results [8] makes it clear that it is possible to spend millions of dollars on hardware and software, only to construct a multi-terabyte warehouse that still requires hours or even days to answer certain queries.

One promising way to address this problem is to redesign analytic query processing systems so that the largest

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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

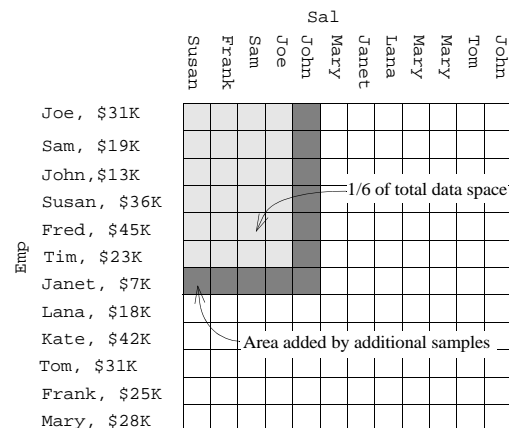


Figure 1: Evaluating a ripple join.

data warehouses are based on *randomization*. Randomization has the overwhelming benefit of facilitating query processing algorithms that do not perform monolithic, days-long calculations, but rather give immediate feedback. Immediately after a query is issued, the system can come back with an initial guess of the form “with 95% probability, the final answer to the query will be within  $\pm 12.4$  of 127.4.” As time goes on and the system is able to process more data, the query error shrinks, and the user can stop the query evaluation as soon as he or she is happy with the accuracy. Pioneering work in the design of such a system has been undertaken by Hellerstein, Haas and their various collaborators [2, 10, 4, 5], resulting in a series of papers on *online aggregation* using random sampling techniques.

Despite the breadth and depth of the seminal research in this area, much work remains if randomization-based data warehousing systems are to become practical and widely-used. One of the most obvious areas where the state-of-the-art is lacking is in an understanding of how to design and implement online, sampling-based algorithms that can be used with common, *subset-based SQL* queries. By *subset-based*, we refer to those queries having a correlated inner query that is related to the outer query through a check for the existence (or lack thereof) of a tuple with a desired

property—that is, we are interested only in a subset of the tuples from the outer query. EXISTS, NOT EXISTS, IN, and NOT IN queries are examples of subset-based queries. We focus specifically on the case where both the outer query and the inner query reference a massive table that is too large to fit in memory, and where evaluating the query exactly is a time-consuming computation. The design of online approximation algorithms for such queries is a daunting task; this task is at the heart of this paper.

### 1.1 Why Are Subset-Based Queries Hard?

Certain operations like join and selection lend themselves to sampling and randomization because it is obvious how to “scale-up” the answer to a query that is obtained over a sample, in order to obtain an unbiased estimate for the final answer to the query<sup>1</sup>. For example, imagine that we wish to estimate the answer to a query of the form:

```
SELECT SUM (Emp.Salary)
FROM Emp, Sal
WHERE pred (Emp.Name, Sal.Name);
```

If Emp and Sal are stored in a randomized order on disk, and the query engine has loaded and joined one-half of the tuples from Emp and one-third of the tuples from Sal and joined all of those tuples to produce a “current” query answer  $\mu$ , then  $(1/2 \times 1/3)^{-1}\mu = 6\mu$  is an unbiased estimate for the eventual result of the query. The intuition behind this is pictured in Figure 1. Since we have explored one-sixth of the join’s data space, our eventual total should be approximately six times as large as the current total. Thus, at all times the current answer to a query over a sample can be “scaled up” to produce a high-quality estimate for the eventual answer to the query. This is the basic idea behind the well-known *ripple join* online aggregation algorithm of Haas and Hellerstein [2].

Unfortunately, most subset-based SQL queries do not have such structure. Imagine that instead of the previous join, we are asked to answer the following, subset-based query:

```
SELECT SUM (Emp.salary)
FROM Emp
WHERE NOT EXISTS (
  SELECT *
  FROM Sal
  WHERE Emp.Name = Sal.Name);
```

If the relation Emp lists all of a company’s employees and the relation Sal lists all of the company’s sales (with Sal.Name listing the employee responsible for the sale), then this query asks: “What is the total salary of all employees who have not generated any sales?”

Suppose that the query engine has loaded and processed one-half of the tuples from Emp and one-third of the tuples

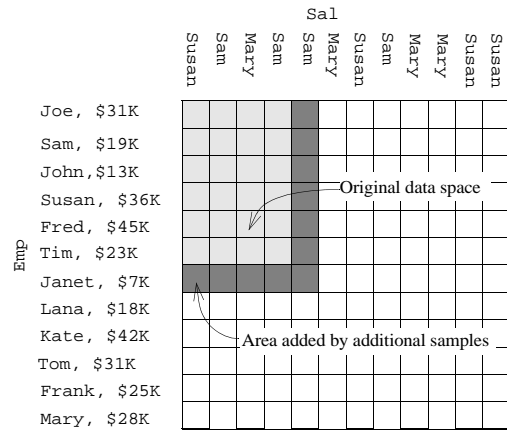


Figure 2: Adding a new tuple that increases the total.

from Sal. In this case, there is no corresponding, straightforward way to scale-up the current answer to the query like there is in the case of a join. As we see more and more tuples from Emp and Sal, the current value for the query answer can *either* shrink or grow, depending on the characteristics of the data, even if the attribute values that are to be aggregated over are strictly positive. This means that by simply scaling up the current answer to compensate for unseen tuples, in general we could be going in either the right or wrong direction. For example:

- Sometimes, processing the whole database will cause the current answer to *shrink*, since with each additional tuple we have a greater and greater chance to find tuples from Sal that will match a given tuple Emp. Reconsider Figure 1. At the current point during the execution of the query, the answer over all tuples read from disk is  $\$13K + \$45K + \$23K = \$81K$ . If we were to read one more tuple from Emp and one more tuple from Sal and incorporate these tuples into the query evaluation, then the current answer shrinks to  $\$75K$ . The answer *decreases* due to the fact that we found a match for John in Sal, which caused John’s salary to be *removed* from the total.
- However, under different conditions, the current result can *grow*. If we have already seen an employee name from Sal, then additional Sal tuples with that name will not remove additional Emp tuples from consideration. Such a situation is depicted in Figure 2. At the current point the answer is  $\$31K + \$13K + \$45K + \$23K = \$112K$ . Incorporating one more tuple from each relation inflates the current answer to  $\$119K$ . The answer actually *increases* because we have added the salary associated with Janet into the total, and seeing another Sam tuple in Sal does not remove any Emp tuples from the total.

This erratic behavior makes it impossible to come up with a one-size-fits-all rule for scaling up or scaling down

<sup>1</sup>“Unbiased” means that if the estimator were repeated an infinite number of times, the average would be equal to the correct answer to the query.

the answer to the query.

## 1.2 Our Contributions

In this paper, we carefully consider the problem of producing accurate, online estimates to *subset-based* queries. By *subset-based*, we mean those queries of the form:

```
SELECT SUM  $f(e)$ 
FROM Emp AS  $e$ 
WHERE NOT EXISTS (
  SELECT *
  FROM Sal as  $s$ 
  WHERE  $pred(e, s)$ );
```

Note that the function  $f$  can encode any mathematical function over tuples from Emp; in particular it can encode a relational selection predicate.  $pred$  is any boolean predicate over tuple pairs from Emp and Sal. This type of query is general enough that it can encode any EXISTS, NOT EXISTS, IN, or NOT IN query over two relations, as well as the set-based INTERSECT and EXCEPT SQL operations. Ratio-based aggregate queries such as AVG can be answered by making use of our algorithms along with Bonferroni’s inequality (i.e., the “union bound”).

The specific contributions of our research are as follows:

1. In this work we develop online, sampling-based algorithms for answering such queries, which rely on reading the relations Emp and Sal in a randomized input order. The algorithms are *online* in the sense that at all times, they produce an estimate that can be updated continuously and presented to the user.
2. The paper includes a statistical analysis of the properties of our estimators, as well as a performance study documenting the feasibility of our approach. We show that the convergence rate of our approach can be faster than that achieved through the obvious, index-based alternative.
3. Our algorithms greatly extend the class of queries that can be efficiently processed using online, sampling-based methods.

## 1.3 Paper Organization

In Section 2, we describe an index-based sampling algorithm for online, approximate answering of subset-based SQL queries. The problem with this algorithm is that it may be extremely slow to converge to a good estimate due to the large number of random disk I/Os required. Section 3 describes an algorithm that is much more efficient in terms of the I/O required, but that may provide an extremely biased estimate for the query result. Sections 4 and 5 discuss how these two flawed estimators may be combined to provide a single, high quality estimator that can adapt to the underlying characteristics of the data. Section 6 discusses how

to extend the algorithms in this paper to additional subset-based queries, and Section 8 describes the related work. The paper is concluded in Section 9.

## 2 A Simple Estimator for Subset Queries

It is actually quite easy to define a very simple and easy-to-implement estimator for the type of subset query considered in this paper. However, as we will discuss in this Section, this estimator is likely to have unacceptable performance under common circumstances.

### 2.1 An Indexed Estimator

It is most natural to begin the quest for any sampling-based estimator by asking: “Can the problem be reduced to estimating the total value of a population by simply sampling without replacement from that population?” While it may not always be possible to develop such an estimator, if it is possible, this represents the easiest solution.

In the case of the subset-based SQL queries considered in this paper, such a simple estimator does actually exist, given the following two reasonable assumptions:

1. First, we assume that the predicate  $pred(e, s)$  contains an equality condition on one or more of the attributes of the two relations. The other predicate conditions, if any, should be in conjunction with the equality condition (for example,  $pred(e, s)$  might be a conjunction of several conditions, including  $e.Name = s.Name$ ).
2. Second, we assume that an index is available on the attribute from the Sal relation on which the equality check is performed. In our example, a B+-Tree or hash table indexing Sal.Name would suffice.

If these assumptions do hold, then Algorithm 1 (hereafter referred to as the “Indexed Algorithm”) gives a natural, online estimate for the query of Section 1.2 (a discussion of the implications if they do not hold is given in Section 6). In the Indexed Algorithm and in the remainder of the paper, we use  $n_S$  to denote the size of a set  $S$ , and we define a function  $one(e, S)$  that evaluates to 1 if and only if there exists  $s \in S$  where  $pred(e, s)$  is *true* (and 0 otherwise). Also, we assume that Emp is stored in a randomized order on disk.

#### Algorithm 1: Indexed Subset Query Estimation

1. Let  $tot = 0$
2. For  $i = 0$  to  $n_{Emp}$  do:
  3. Read the  $i$ th tuple  $e_i$  from Emp
  4. Perform an indexed search through Sal to evaluate  $one(e_i, Sal)$
  5.  $tot = tot + (1 - one(e_i, Sal)) \times f(e_i)$
6. Output  $\hat{\mu} = (n_{Emp}/i) \times tot$  as the current estimate

Not only is the online estimator  $\hat{\mu}$  unbiased, but at all times we can easily compute an unbiased estimate  $\hat{\sigma}^2(\hat{\mu})$  for the variance of  $\hat{\mu}$  [13]:

$$\hat{\sigma}^2(\hat{\mu}) = \left(\frac{n_{\text{Emp}} - i}{n_{\text{Emp}} i}\right) s_{\hat{\mu}}^2$$

where:

$$s_{\hat{\mu}}^2 = \frac{1}{(i-1)} \sum_{j=1}^i (n_{\text{Emp}} f(e_j)(1 - \text{one}(e_j, \text{Sal})) - \hat{\mu})^2$$

In this expression,  $s_{\hat{\mu}}^2$  is the sample variance of the tuples seen thus far, and it can be calculated incrementally as new tuples are encountered. If we make the assumption that  $\hat{\mu}$  is normally distributed (this assumption is reasonable due to the central limit theorem [12]), then it is easy to use this variance estimate to provide confidence bounds on  $\hat{\mu}$  using standard techniques [13].

Note that this simple estimator is very closely related to the estimators developed by Hellerstein, Haas, and Wang for estimates over relational selection predicates [4], the only differences being that (1) we make use of sampling without replacement (which is generally more accurate than sampling with replacement), and (2) we make use of an index on the *Sal* relation in order to evaluate the predicate  $\text{one}(e_i, \text{Sal})$ .

## 2.2 So, What’s the Problem?

Though very simple and attractive at first glance, the Indexed Algorithm can be prohibitively slow to converge due to the large number of random disk I/Os that may be required, specifically because of the extensive reliance on index lookups to perform the estimation.

If the *Sal* relation is very large, every single execution of line (4) of the Indexed Algorithm probably requires at least two random disk I/Os: one random I/O to perform a lookup in the index, and a second random I/O to access the potential matches from *Sal* (assuming that the index on *Sal* is a secondary index). In reality, many more random I/Os would be required in the case where potential matches from *Sal* are scattered all over the disk, and the index lookup for each  $e_i$  returns many tuples from *Sal* that must all be retrieved.

This is problematic because, if every iteration of the loop requires at least two random disk I/Os at around 10 milliseconds each, as is usually the case, we can only examine at most 3000 tuples from *Emp* per minute per disk. As a result, the Indexed Algorithm is only suitable for use with very well-behaved queries and databases.

## 3 A Concurrent, Sampling-Based Estimator

Since the obvious, indexed solution is likely to be unacceptably slow, we now return to the alternative described in the Introduction: what if we assume that both relations are clustered randomly on disk, and just scan them concurrently, evaluating the query as we go, in a manner similar

to the ripple join [2]? Thus, the estimate for the final query answer will be nothing but a “scaled-up” current answer. This section formally describes and mathematically evaluates such an algorithm. Of course, anecdotal evidence was given in the Introduction that such an algorithm is likely to have significant associated problems for subset-based SQL queries. Though it will prove to be unsuitable by itself, the algorithm will still be an important component of the more complete solution we describe in Section 4.

### 3.1 The Concurrent Sampling Algorithm

Algorithm 2 (subsequently referred to as the “Concurrent Algorithm”) formally describes the process illustrated in Figure 1 and Figure 2. In the Concurrent Algorithm, it is assumed that *Emp* and *Sal* are clustered in a statistically random order on disk. Also, it is assumed that each loop iteration (or “sampling step” [2]) loads  $b_{\text{Emp}}$  blocks from *Emp* and  $b_{\text{Sal}}$  blocks from *Sal*, where  $b_{\text{Emp}}$  and  $b_{\text{Sal}}$  are parameters to the algorithm. In the Concurrent Algorithm (and the remainder of the paper),  $\text{Emp}'$  refers to the subset of *Emp* that has been sampled thus far, and  $\text{Sal}'$  is the analogous subset for *Sal*.

#### Algorithm 2: Concurrent Sampling

1. Let  $\text{active} = \text{Emp}' = \text{Sal}' = \{\}$ ;  $\text{tot} = 0$ ;  $\text{cnt} = 0$
2. While  $\text{Emp}' \neq \text{Emp}$  and  $\text{Sal}' \neq \text{Sal}$ :
  3. Add  $b_{\text{Emp}}$  blocks from *Emp* to  $\text{Emp}'$
  4. Add  $b_{\text{Sal}}$  blocks from *Sal* to  $\text{Sal}'$
  5. For each newly added tuple  $e$  from  $\text{Emp}'$  do:
    6.  $\text{cnt} = \text{cnt} + 1$
    7. If  $\neg \exists s \in \text{Sal}'$  where  $\text{pred}(e, s)$  then:
      8.  $\text{tot} = \text{tot} + f(e)$
      9.  $\text{active} = \text{active} \cup \{e\}$
  10. For each newly added tuple  $s$  from  $\text{Sal}'$  do:
    11. For each tuple  $e$  from  $\text{active}$  do:
      12. If  $\text{pred}(e, s)$  then:
        13.  $\text{active} = \text{active} - \{e\}$
        14.  $\text{tot} = \text{tot} - f(e)$
  15. Output  $(n_{\text{Emp}} \times \text{tot})/\text{cnt}$  as the current estimate

In a manner very similar to a ripple join, the Concurrent Algorithm loads a number of sampled records from each relation into memory, and then processes them in such a way that the value of  $\text{tot}$  is always equivalent to the output of the original query, had it been run over the tuples that had been sampled thus far from *Emp* and *Sal*. Assuming that the set  $\text{active}$  of “active” tuples from *Emp* always remains small enough to store in main memory and that the predicate  $\text{pred}(e, s)$  contains an equality check on the attribute *Name*, then it would make sense to index  $\text{Sal}'$  using an in-memory hash table or self-balancing binary tree that organizes the tuples currently in  $\text{Sal}'$  over their *Name* values. This would allow steps (5)-(9) of the Concurrent Algorithm to be efficiently executed. Likewise, assuming that  $\text{Emp}'$  remains small enough to fit in memory, it would

also make sense to index  $\text{Emp}'$  in the same way to facilitate an efficient implementation of steps (10)-(14).

### 3.2 So How Accurate Is It?

This Section considers the problem of formally determining exactly how (in)accurate the estimator implemented by the Concurrent Algorithm is expected to be.

In the remainder of the paper, we let  $\alpha_i$  denote the fraction  $(n_{\text{Emp}'} - i)/(n_{\text{Emp}} - i)$ .  $\alpha_i$  is the probability that an arbitrary tuple  $e \in \text{Emp}$  is also in  $\text{Emp}'$ , given the knowledge that  $i$  tuples that are not  $e$  have already been selected for  $\text{Emp}'$ . Note that if  $i$  is 0, then this is merely the probability that an arbitrary tuple  $e$  is in  $\text{Emp}'$ . We also define a series of Bernoulli (zero/one) random variables, where  $X_i$  governs whether or not the  $i$ th tuple from  $\text{Emp}$  is found in  $\text{Emp}'$  (note that the probability that  $X_i$  is one is simply  $\alpha_0$ ). Given this notation, the following random variable is exactly equivalent to the estimate given in line (15) of the Concurrent Algorithm:

$$N = \frac{1}{\alpha_0} \sum_{e_i \in \text{Emp}} X_i f(e_i) (1 - \text{one}(e_i, \text{Sal}'))$$

Recall from Section 2.1 that  $\text{one}(e, S)$  evaluates to 1 if and only if there exists  $s \in S$  where  $\text{pred}(e, s)$  is *true* (and 0 otherwise).

When trying to answer the question ‘‘How accurate is  $N$ ?’’ the first place to start is to determine if there is any *bias* in the estimate provided by  $N$  (that is, on expectation, does a trial over  $N$  result in the correct answer to the query? See Johnson et al. [6] Chapter 1 for a nice introduction to moments and expectation). Taking the expectation of both sides of the equation for  $N$ , we have:

$$E[N] = E\left[\frac{1}{\alpha_0} \sum_{e_i \in \text{Emp}} X_i f(e_i) (1 - \text{one}(e_i, \text{Sal}'))\right]$$

Since  $\frac{1}{\alpha_0}$  is a constant and the sampling of  $\text{Emp}$  and  $\text{Sal}$  are independent, we have:

$$\begin{aligned} E[N] &= \frac{1}{\alpha_0} \sum_{e_i \in \text{Emp}} E[X_i] f(e_i) E[(1 - \text{one}(e_i, \text{Sal}'))] \\ &= \frac{1}{\alpha_0} \sum_{e \in \text{Emp}} \alpha_0 f(e) E[(1 - \text{one}(e, \text{Sal}'))] \\ &= \sum_{e \in \text{Emp}} f(e) E[(1 - \text{one}(e, \text{Sal}'))] \end{aligned}$$

Note that  $(1 - \text{one}(e, \text{Sal}'))$  is a zero/one random variable that evaluates to 0 if and only if a tuple  $s$  such that  $\text{pred}(e, s)$  is *true* was found in  $\text{Sal}'$ . The expected value of any zero/one random variable is simply the probability that a trial over the variable results in a one. Let  $\varphi(n, m, k)$  denote the probability that we will fail to select any interesting tuples, if we select  $m$  tuples at random from a relation

of total size  $n$  tuples having only  $k$  tuples with a certain property that we are interested in. Then using the hypergeometric distribution, we have:

$$\varphi(n, m, k) = \binom{n-k}{m} / \binom{n}{m}$$

Function  $\varphi$  can be computed essentially in constant time if we express it in terms of the Gamma function, for which series-based algorithms are readily available. Given this, we have:

$$E[N] = \sum_{e \in \text{Emp}} f(e) \varphi(n_{\text{Sal}}, n_{\text{Sal}'}, \text{cnt}(e, \text{Sal}))$$

where  $\text{cnt}(e, \text{Sal})$  counts the number of tuples in  $s \in \text{Sal}$  for which  $e \text{ pred}(e, s)$  evaluates to *true*.

### 3.3 The Really Bad News

Note that the *correct* answer to the query is:

$$\sum_{e \in \text{Emp}} f(e) (1 - \text{one}(e, \text{Sal}))$$

Though at first glance this looks rather similar to the formula for  $E[N]$ , it turns out that  $(1 - \text{one}(e, n_{\text{Sal}}))$  is equivalent to  $\varphi(n_{\text{Sal}}, n_{\text{Sal}'}, \text{cnt}(e, \text{Sal}))$  only if  $e$  will eventually survive the NOT EXISTS clause of the query, or if we have seen enough tuples from  $\text{Sal}$  that we are guaranteed that it is *impossible* to miss any tuple  $s \in \text{Sal}$  for which  $\text{pred}(e, s)$  is *true*. As a result, the estimator of the Concurrent Algorithm is typically not correct on expectation (that is,  $N$  is *biased*). Furthermore, the bias can be arbitrarily large. For example, if each tuple  $e$  from  $\text{Emp}$  has exactly one  $s \in \text{Sal}$  for which  $\text{pred}(e, s)$  is *true*, then the probability that we will not find  $s$  may be almost one. In this case, the bias would be equivalent to the sum of  $f(e)$  over all of the tuples in  $\text{Emp}$ ! As a result, we must search for a better estimator.

## 4 A Combined Estimator

Though the Concurrent Algorithm may produce an estimate with a large bias, the *variance* of this estimate should decrease quickly with time, since the estimate can quickly incorporate a large number of tuples by using a fast, sequential scan of the input relations. As long as the bias can be corrected for, the estimator may be salvageable. Thus, in this Section we consider the problem of correcting Algorithm 2’s bias. The basic strategy is to combine the estimator computed by the Concurrent Algorithm with an estimator that is very similar to the one computed by the Indexed Algorithm, in order to develop a combined estimator that is superior to either individual estimator.

#### 4.1 The Indexed Solution Revisited

Imagine that we developed an estimator  $U$  where:

$$E[U] = \sum_{e \in \text{Emp}} f(e)(1 - \text{one}(e, \text{Sa1}) - \varphi(n_{\text{Sa1}}, n_{\text{Sa1}'}, \text{cnt}(e, \text{Sa1})))$$

Then we know from the linearity of expectation that:

$$E[N + U] = \sum_{e \in \text{Emp}} f(e)(1 - \text{one}(e, \text{Sa1}))$$

This implies that  $N + U$  would be an unbiased estimate for the result of the query. In other words, we could simply add  $U$  to the estimate produced at every execution of line (15) in the Concurrent Algorithm in order to correct for the algorithm's bias.

A natural way to provide an estimator like  $U$  would be to sample a number of tuples from  $\text{Emp}$ . For each tuple  $e$  that is sampled, we compute and sum the exact value of  $f(e)(1 - \text{one}(e, \text{Sa1}) - \varphi(n_{\text{Sa1}}, n_{\text{Sa1}'}, \text{cnt}(e, \text{Sa1})))$ , and then scale up the result accordingly.

The difficulty of providing such an estimator is that it would require that we have good information about how many tuples in  $\text{Sa1}$  correspond with each tuple sampled from  $\text{Emp}$  (that is, we need to be able to compute  $\text{cnt}(e, \text{Sa1})$  for an arbitrary tuple from  $\text{Emp}$ ). One direction to solve this problem would be estimating the quantity  $\text{cnt}(e, \text{Sa1})$  itself. However, simply obtaining an *estimate* for each  $\text{cnt}(e, \text{Sa1})$  is not good enough, because it must be used as an argument for  $\varphi$ . Plugging a value into  $\varphi$  that is an estimate is problematic for two reasons:

1.  $\varphi$  is a complicated nonlinear function. If the input parameters are themselves estimates, then the output of  $\varphi$  becomes an estimate. The complexity of the function would make the quality of the output extremely difficult to reason about statistically.
2.  $\varphi$  takes only discrete values as input. Since most natural estimators are not integer-valued, they cannot easily be used in conjunction with  $\varphi$ . If we apply a natural solution like truncation or rounding to the input parameters, this would make the quality of the output that much more difficult to reason about.

Given that the input to  $\varphi$  should be an exact value, the natural way to compute  $\text{cnt}(e, \text{Sa1})$  would be to rely on an index over  $\text{Sa1}$ , just like the Indexed Algorithm. Of course, the problem discussed in Section 2 was that this tactic will be slow if we need to compute  $\text{cnt}(e, \text{Sa1})$  for every  $e$  contained in a large sample from  $\text{Emp}$ .

However, making use of such index-based sampling is much less of a problem in this context because this index-based sampling will only be used as a *supplement* to the information gathered by the Concurrent Algorithm. As a result, we may not need many index-based samples to

perform this task. Most of the work will be done by the fast, sequential sampling performed by the Concurrent Algorithm, with just enough information added using index-based samples that we can accurately unbiased the Concurrent Algorithm's estimate. The resulting algorithm is Algorithm 3 (hereafter referred to as the Combined Algorithm).

#### Algorithm 3: A Combined Algorithm

1. Let  $E = \{\}$

*Procedure PreSample* (int numSam)

2. For  $i = 1$  to numSam do:

3. Randomly sample  $e$  from  $\text{Emp}$

4. Perform indexed lookup of matches for  $e$  in  $\text{Sa1}$

5.  $E = E \cup \{(e, \text{cnt}(e, \text{Sa1}))\}$

*Function UnBias* (int samSize)

6.  $\text{tot} = 0$

7. For each  $(e, \text{cnt}) \in E$  do:

8. If  $(\text{cnt} > 0)$  then:

9.  $\text{tot} = \text{tot} - f(e)\varphi(n_{\text{Sa1}}, \text{samSize}, \text{cnt})$

10. Return  $(n_{\text{Emp}}/|E|) \times \text{tot}$

*Procedure CombinedAlg* (int preSamSize)

11. Call *PreSample* (preSamSize)

12. Invoke a modified version of the Concurrent Algorithm, with line (15) changed to output  $(n_{\text{Emp}} \times \text{tot})/\text{cnt} + \text{UnBias}(n_{\text{Sa1}'})$  as the current estimate for the query answer

In the remainder of the paper, we assume that the sampling performed in lines (3) and (12) of the Combined Algorithm are independent. To enforce this, our implementation of the Combined Algorithm performs the sampling required by line (3) by seeking to a random location in  $\text{Emp}$  to sample each record; since each record obtained by *PreSample* likely requires two or more additional random I/Os already to perform the index lookup of line (4), this extra random I/O is not too costly. A fast sequential scan of the  $\text{Emp}$  relation to perform the sampling required by line (12) will then produce a sample that is independent of the sample drawn by line (3).

#### 4.2 Analysis and Statistical Bounds

Simply knowing that the Combined Algorithm provides an unbiased estimate is not enough: it is crucial to associate *confidence bounds* with the estimates produced by the algorithm, so that a user can be kept informed of the accuracy of the algorithm's estimate. A *confidence bound* is an assertion of the form "With probability  $p$ , the exact answer to the query is within the range *low* to *high*". The probability  $p$  is typically supplied by the user, and then *low* and *high* are computed by the system.

The first step in developing a confidence bound for the estimate  $(N + U)$  is to derive the *variance* of this estimator

(denoted  $\sigma^2(N + U)$ ). Since, as described above, we force  $N$  and  $U$  to be independent, we know that  $\sigma^2(N + U) = \sigma^2(N) + \sigma^2(U)$ . Since  $U$  is an estimation of a population sum from a random subset of the population, a consistent estimator for  $\sigma^2(U)$  can be obtained using standard formulas, similar to those given in Section 2.1. Specifically:

$$\hat{\sigma}^2(U) = \left(\frac{n_{\text{Emp}} - n_E}{n_{\text{Emp}} n_E}\right) s_{\bar{U}}^2$$

where  $s_{\bar{U}}^2$  is the sample variance of the values used to compute  $U$ . If  $\bar{U}$  is the current estimate of the  $UnBias$  procedure of the Combined Algorithm, then  $s_{\bar{U}}^2$  is as follows:

$$s_{\bar{U}}^2 = \frac{1}{n_E - 1} \sum_{(e,i) \in E} (n_{\text{Emp}} f(e)(1 - \text{one}(e, \text{Sal})) - \varphi(n_{\text{Sal}}, n_{\text{Sal}'}, i) - \bar{U})^2$$

However, a derivation of  $\sigma^2(N)$  is not so straightforward. We know from the definition of variance that:

$$\sigma^2(N) = E[N^2] - E^2[N]$$

Recall that  $E[N]$  was derived in Section 3.2, and so it is an easy matter to square this value and plug the result into the above formula. However, deriving a formula for  $E[N^2]$  is another matter. The formula for  $E[N^2]$  (i.e., the *second moment of  $N$* ) is the paper's central theoretical result.

**Theorem 4.1 Second moment of  $N$ .** *Let  $(e_i \cup e_j)$  denote any tuple matching either  $e_i$  or  $e_j$ . That is,  $(e_i \cup e_j)$  denotes any tuple  $t$  for which either  $\text{pred}(t, e_i)$  or  $\text{pred}(t, e_j)$  evaluates to true. Then:*

$$E[N^2] = \frac{2\alpha_1}{\alpha_0} \left\{ \sum_{e \in \text{Emp}} \frac{1}{2\alpha_1} f^2(e) \varphi(n_{\text{Sal}}, n_{\text{Sal}'}, \text{cnt}(e, \text{Sal})) + \sum_{\{e_i, e_j\} \subset \text{Emp}} f(e_i) f(e_j) \varphi(n_{\text{Sal}}, n_{\text{Sal}'}, \text{cnt}(e_i \cup e_j, \text{Sal})) \right\}$$

**Proof** We know from Section 3.2 that:

$$N = \frac{1}{\alpha_0} \sum_{e_i \in \text{Emp}} X_i f(e_i) (1 - \text{one}(e_i, \text{Sal}'))$$

Thus:

$$N^2 = \frac{1}{\alpha_0^2} \sum_{e_i \in \text{Emp}} \sum_{e_j \in \text{Emp}} X_i X_j f(e_i) f(e_j) \times (1 - \text{one}(e_i, \text{Sal}')) (1 - \text{one}(e_j, \text{Sal}'))$$

And so:

$$E[N^2] = \frac{1}{\alpha_0^2} \left\{ \sum_{e_i \in \text{Emp}} E[X_i f^2(e_i) (1 - \text{one}(e_i, \text{Sal}'))] + \sum_{\{e_i, e_j\} \subset \text{Emp}} 2E[X_i X_j f(e_i) f(e_j) (1 - \text{one}(e_i, \text{Sal}')) \times (1 - \text{one}(e_j, \text{Sal}'))] \right\}$$

Using the independence of the sampling from  $\text{Emp}$  and  $\text{Sal}$  and the fact that  $E[(1 - \text{one}(e_i, \text{Sal}')) (1 - \text{one}(e_j, \text{Sal}'))]$  is  $\varphi(n_{\text{Sal}}, n_{\text{Sal}'}, \text{cnt}(e_i \cup e_j, \text{Sal}))$ , the result follows after algebraic manipulation. ■

Note that this Theorem gives us a way to compute the exact variance of  $N$ , but it requires that we know the values of the various  $\varphi$  terms as well as the value of  $f(e)$  for every tuple in the  $\text{Emp}$  relation. Clearly, this is not practical. Thus, as is standard practice in statistics, we will estimate  $E[N]$  and  $E[N^2]$  from the segment of the population for which we have exact information; specifically, we can compute the required sums only over those tuples for which we obtained exact information during the *PreSample* procedure of the Combined Algorithm, and then scale up the result accordingly [7, 13] (though care must be taken so that the estimate for  $E^2[N]$  obtained from  $E[N]$  is not biased).

Since we now have a high-quality estimator for  $\sigma^2(N + U)$ , it is then an easy matter to derive confidence bounds assuming either a normal distribution for the error [12] (justified by the central limit theorem), or a more conservative distribution-free bound such as the one provided by Chebyshev's inequality [1]. Such techniques are fairly standard in statistics [12, 13].

## 5 Increasing the Accuracy

This Section considers the question of how to fine-tune the Combined Algorithm to maximize its performance.

### 5.1 Are Two Always Better Than One?

It is useful to begin with an intuitive discussion of why the estimate of the Combined Algorithm may be worse than the estimate of the Indexed Algorithm, and why it may be better, even if both algorithms are given the same amount of time for query processing. This will provide the background needed to motivate the development of the remainder of the Section.

It is fairly obvious why the estimate  $(N + U)$  is usually better than  $N$  alone:  $N$  may have severe bias, which is corrected by the addition of  $U$ . However, it is far less obvious why index-based sampling performed by the Indexed Algorithm can be helped through the addition of  $N$ . Why not just rely on the Indexed Algorithm, and forgo the complexity of the Combined Algorithm? Two observations supporting this position are:

1. The indexed sampling performed by the Combined Algorithm is potentially more expensive than the indexed sampling performed by the Indexed Algorithm, since the Combined Algorithm needs access to  $\text{cnt}(e, \text{Sal})$  for every tuple  $e$  obtained during the *PreSample* routine, whereas the Indexed Algorithm only needs access to  $\text{one}(e, \text{Sal})$ . If the query predicate  $\text{pred}(e, s)$  has low filtering power, computing the latter may be far less expensive than the former. This means that the Combined Algorithm must make do with a smaller indexed sample than the Indexed Algorithm.

2. Both algorithms produce unbiased estimators, so the error of both algorithms is related only to the variance of the algorithms' estimators. As discussed above, the Combined Algorithm uses the estimator  $U$ , which typically makes use of a smaller index-based sample than the sample used by the Indexed Algorithm. Thus,  $U$  should have higher variance than the Indexed Algorithm's estimator. Furthermore, variances are additive. The Combined Algorithm must add the estimator  $N$  to  $U$ , which should increase the variance of the Combined Algorithm's estimator even more.

These factors may indeed render the Indexed Algorithm more accurate than the Combined Algorithm in certain situations. However, the Combined Algorithm may still be preferable because it is so tremendously costly to perform every index-based sample, and the Combined Algorithm does not rely exclusively on indexed sampling for its accuracy. In general, the variance of  $N$  will shrink much more quickly than the variance of  $U$ , because  $N$  does not use an index. Thus, very quickly all of the variance (and hence all of the error) of the Combined Algorithm's estimator is related to  $U$ . Critically, the estimator  $U$  differs from the Indexed Algorithm's estimator in that its variance may shrink over time *regardless* of whether or not additional index-based samples are taken. Note that the function  $UnBias$  returns the following value as the estimator  $U$ :

$$\sum_{e \in \text{Emp}'} \frac{f(e)}{\alpha_0} (1 - \text{one}(e, \text{Sal}) - \varphi(n_{\text{Sal}}, n_{\text{Sal}'}, \text{cnt}(e, \text{Sal})))$$

As  $\varphi(n_{\text{Sal}}, n_{\text{Sal}'}, \text{cnt}(e, \text{Sal}))$  approaches  $1 - \text{one}(e, \text{Sal})$ , the variance of  $U$  approaches zero (since all terms in the summation are always zero in this case). As a result, the larger  $n_{\text{Sal}'}$  (that is, the more tuples from  $\text{Sal}$  are used to compute  $N$ ) the lower the variance of  $U$ , and the variance of  $U$  is reduced over time without any costly indexed samples. In many cases, it will be reduced enough that the variance  $N + U$  is actually lower than the variance of the Indexed Algorithm's estimator.

## 5.2 Weighting the Combined Estimator

The previous Subsection gave some justification as to why  $(N + U)$  can be a better estimator than the estimator of the Indexed Algorithm. However, this justification does not hold in all situations. If  $\varphi(n_{\text{Sal}}, n_{\text{Sal}'}, \text{cnt}(e, \text{Sal}))$  is not a good approximation to  $(1 - \text{one}(e, \text{Sal}))$ , then the variance of  $U$  may not be sufficiently reduced to compensate for the addition of  $N$ .

Fortunately, it is possible to modify the computations performed by our algorithms in order to automatically reduce the variance of  $U$  by effectively increasing the variance of  $N$ , while still guaranteeing that  $N + U$  is unbiased. By carefully optimizing this trade-off, the overall estimator can be improved substantially. Given a weight  $w$ , we first modify the  $UnBias$  routine as follows:

```

Function UnBias (int samSize)
6. tot = 0
7. For (e, cnt) ∈ E do:
8. If (cnt > 0) then:
9. tot = tot - f(e)wφ(nSal, samSize, cnt)
10. Else:
11. tot = tot + (1 - w)
12. Return (nEmp/nE) × tot

```

Then, every time line (15) of the Concurrent Algorithm is invoked by the Combined Algorithm, we output:

$$w(n_{\text{Emp}} \times \text{tot}) / \text{cnt} + UnBias(n_{\text{Sal}'})$$

as the current estimate of the Combined Algorithm.

What we have done via this modification is to allow for a relative weighting of the components of the estimator  $(N + U)$ . If the variance of  $N$  is relatively large, then we can use  $w = 0$  to give us an estimate that is totally equivalent to what we would have obtained using the Indexed Algorithm. Using  $w = 0$  eliminates  $N$  from the estimate at the same time that we eliminate the bias correction provided by  $UnBias$ . On the other hand, if the variance of  $N$  is relatively small, then we can use a value for  $w$  that is larger than 1; this will tend to *increase* the variance of  $N$  at the same time that it *decreases* the variance of  $U$ . By carefully choosing a value for  $w$ , we end up with an estimator whose error is always upper bounded by the error of the naive Combined Algorithm, and that will typically be far superior.

## 5.3 Optimizing the Weight Parameter $w$

Fortunately, there is a closed-form formula for the optimal value of  $w$ . To derive this formula, we differentiate  $\sigma^2(N + U)$  with respect to  $w$  and then solve for the zero. The process is tedious but straightforward. We begin the process with  $U$ . Note that with the addition of  $w$  into the Combined Algorithm, the formula for  $s_U^2$  from Section 4.2 becomes:

$$s_U^2 = \frac{1}{n_E - 1} \sum_{(e,i) \in E} (n_{\text{Emp}} f(e) (1 - \text{one}(e, \text{Sal}) - w \varphi(n_{\text{Sal}}, n_{\text{Sal}'}, i)) - \bar{U})^2$$

Then let:

$$\begin{aligned}
a &= \frac{1}{n_E - 1} \left( \frac{n_{\text{Emp}} - n_E}{n_{\text{Emp}} n_E} \right) \\
b &= \sum_{(e,i) \in E} n_{\text{Emp}}^2 f^2(e) (1 - \text{one}(e_x, \text{Sal})) \varphi(n_{\text{Sal}}, n_{\text{Sal}'}, i) \\
c &= \sum_{(e,i) \in E} n_{\text{Emp}}^2 f^2(e) \varphi^2(n_{\text{Sal}}, n_{\text{Sal}'}, i) \\
d &= \sum_{(e,i) \in E} n_{\text{Emp}}^2 f^2(e) (1 - \text{one}(e_x, \text{Sal}))
\end{aligned}$$

$$U_L = \sum_{(e,i) \in E} n_{\text{Emp}} f(e)(1 - \text{one}(e, \text{Sa1}))$$

$$U_R = \sum_{(e,i) \in E} n_{\text{Emp}} f(e) \varphi(n_{\text{Sa1}}, n_{\text{Sa1}'}, i)$$

With this, we have:

$$\bar{U} = \frac{1}{n_E} (U_L - wU_R)$$

$$\sigma^2(U) = a(d - 2wb + w^2c - \frac{[U_L^2 - 2wU_LU_R + w^2U_R^2]}{n_E})$$

(we used the fact that  $\sum_{i=1}^n (x_i - \bar{x})^2 = \sum_{i=1}^n x_i^2 - n\bar{x}^2$ )

Now putting the condition for extremum of the variance of  $wN + U$ ,  $\frac{\partial(\sigma^2(U)^2 + w^2\sigma^2(N))}{\partial w} = 0$  and solving the first degree equation in  $w$ , we obtain as the optimal value for  $w$ :

$$w_{\text{opt}} = \frac{ab - aU_LU_R/n_E}{\sigma^2(N) - aU_R^2/n_E + ac}$$

#### 5.4 Optimizing the Reading

In the previous two Sections we assumed that the sample  $E$  used by the combined estimator was not altered during the estimation process. Clearly, this is an overly restrictive assumption. We also note that the rate at which relations  $\text{Emp}$  and  $\text{Sa1}$  are scanned can be changed depending on the query and the data set. Since we have these extra degrees of freedom in how we execute the query, it is natural to ask the question: what are good choices for these reading rates?

This question can be answered by solving an optimization problem which minimizes the variance of the Combined Algorithm's estimator after a total execution time of  $t$ . Specifically, given a set of times  $t_{\text{Emp}}$ ,  $t_{\text{Sa1}}$ , and  $t_{\text{Index}}$  required to access, process, and add a single tuple to  $\text{Emp}$ ,  $\text{Sa1}$ , and  $E$  respectively, we wish to choose  $n_{\text{Emp}'}$ ,  $n_{\text{Sa1}'}$  and  $n_E$  so as to minimize  $\sigma^2(wN + U)$  subject to the constraint that  $t = t_{\text{Emp}}n_{\text{Emp}'} + t_{\text{Sa1}}n_{\text{Sa1}'} + t_{\text{Index}}n_E$ .

By choosing such optimal reading rates, we can produce a fully optimized version of the Concurrent Algorithm. In our implementation of the optimized version of the Concurrent Algorithm, we begin with an initial invocation of the *PreSample* procedure with  $\text{numSam} = 100$ . We then determine the optimal reading rates using a steepest descent method with random restarts [9]. Once the optimal values of  $n_{\text{Emp}'}$ ,  $n_{\text{Sa1}'}$  and  $n_E$  have been computed,  $b_{\text{Emp}}$  and  $b_{\text{Sa1}}$  from lines (2) and (3) of the Concurrent Algorithm are chosen so that the numbers of tuples read from  $\text{Emp}$  and  $\text{Sa1}$  are proportional to  $n_{\text{Emp}'}$  and  $n_{\text{Sa1}'}$  respectively. After every iteration the current estimate is output in the normal fashion. The final modification is that after each iteration of line (2) of that algorithm has completed, the loop from line (2) of the Combined Algorithm is then executed so that the number of tuples added to  $E$  at each step is proportional to  $n_E$ .

## 6 Discussion

Now that we have defined a complete algorithm for answering a specific class of NOT EXISTS SQL queries, there are a few additional issues that warrant further discussion.

- *How can the algorithms be extended to handle other query types?* As discussed in the Introduction, our algorithms are also suitable for EXISTS, IN, NOT IN, INTERSECT, and EXCEPT queries. Each of these query types can be evaluated using variations on the methods described in this paper. For example, a NOT IN query or an EXCEPT query over two relations can trivially be re-written as a NOT EXISTS query. Evaluation of an EXISTS query is more complicated and requires modification of our algorithms, but the changes required are not radical. Rather than summing over  $(1 - \text{one}(e, \text{Sa1}'))$  for all  $e$  in  $\text{Emp}'$ , the estimator  $N$  will instead sum over  $(\text{one}(e, \text{Sa1}'))$  for all  $e$ . Since this will change the bias of  $N$ , the correction provided by  $U$  must be changed, but the analysis and algorithms are very similar to the results given in the paper for NOT EXISTS queries.

Once EXISTS has been implemented, it becomes possible to evaluate an INTERSECT or IN query via a straightforward transformation into an EXISTS query.

- *What if there is not an equality check within *pred* or we lack an index on *Sa1*?* The indexed solution used in Algorithm 3 is no longer suitable. Our algorithms must be modified slightly to handle this case. In order to implement *PreSample*, we would first sample  $\text{numSam}$  records from  $\text{Emp}$ , and then scan  $\text{Sa1}$  to obtain  $\text{cnt}(e, \text{Sa1})$  for each record in the sample. After this scan is completed, the algorithm could output its first estimate, and begin a block-nested-loops ripple join over both relations, updating the current estimate in an online fashion.

In this case our algorithms require a complete scan of  $\text{Sa1}$  before we can output any results. While this is clearly not desirable, we point out that without an equality check in *pred*, any SQL query engine will have a hard time evaluating the query efficiently. In general, the only way to evaluate such a query *exactly* is with a quadratic cost, nested-loops-style algorithm. Our algorithm in this case would use a similar evaluation technique, but would have the added benefit that after the initial scan of  $\text{Sa1}$ , we would be able to offer online estimates throughout query execution.

- *What if there are joins in the outer query or in the subquery?* As long as only one table is sampled from in the inner query and one table is sampled from in the outer query (and all other tables are buffered in memory or can be read quickly in their entirety), multi-table outer or inner subqueries can be handled with little modification of our algorithms. To handle multiple tables in the outer query, it would first be necessary to join the non-sampled relations with each sampled tuple, and then treat all of the resulting tuples as samples from  $\text{Emp}$  ( $n_{\text{Emp}}$  must then be scaled up accordingly in the estimate and variance calculations). A similar tactic can handle joins in the inner query.

The more difficult case is when, due to the size of the input relations, more than one table must be sampled in ei-

ther the inner or the outer subquery. The statistical analysis presented in this paper is not applicable to such a situation. It seems possible to support joins over multiple sampled relations in the outer query using algorithms very similar to those in this paper, with only some additional analysis required to develop the analog of Theorem 4.1 for the multi-table case. This will be an interesting direction for future work. The much more difficult case is when more than one table in the *inner* query must be sampled from. We doubt that an appropriate statistical analysis of any suitable algorithm is possible in this case. The problem is that it becomes very difficult to evaluate  $\varphi$  exactly for any set of tuples from the outer query if there is a complex join in the inner query (see Section 4.1 for a detailed discussion of this).

## 7 Experiments

The section details the results of a set of experiments designed to test our methods. First, we test the efficiency of our algorithms in a realistic environment by running fully-functional, disk-based implementations of the paper’s algorithms over several multi-gigabyte data sets. Second, we test the statistical properties of the Combined Algorithm by repeating the algorithm hundreds of thousands of times over some small, synthetic data sets.

### 7.1 Large-Scale Experiments

The first set of experiments has several goals. First, we wish to test whether any of these algorithms can produce suitably accurate estimates in realistic cases. Next, we wish to test whether it is actually the case that the Combined Algorithm is able to produce a more accurate estimate than the Indexed Algorithm over a large, disk-resident database. Finally, we wish to test the ability of the optimization described in Section 5.4 to further increase the speed of convergence of the estimator produced by the Combined Algorithm over a large, disk-resident database. Note that we do not experimentally evaluate the Concurrent Algorithm because there is no way to give online confidence bounds with this method; the algorithm has an arbitrarily large bias that cannot easily be computed online without resorting to indexed sampling. This renders any sort of online accuracy guarantees given by the other methods difficult or impossible to provide.

In this Section, we make use of three pairs of synthetically-generated data sets to answer a query of the form:

```
SELECT SUM (e.A) FROM Emp AS e
WHERE e.B AND NOT EXISTS
      (SELECT * FROM Sal AS s
       WHERE s.C AND s.D = e.A)
```

For each data set pair tested, the data set *Emp* is 5GB in size and made of 100B records; the set *Sal* is 25GB in size and also made of 100B records. The sets were generated as follows:

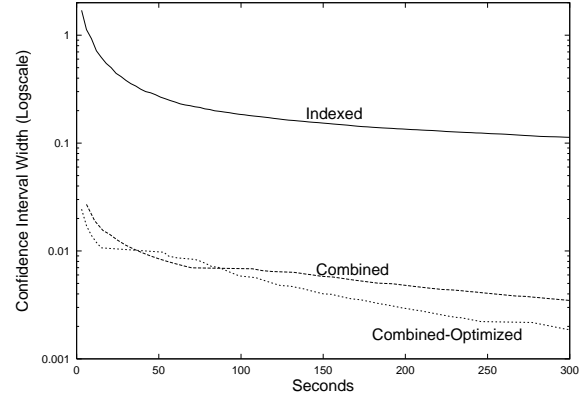


Figure 3: Experimental results for *test1*.

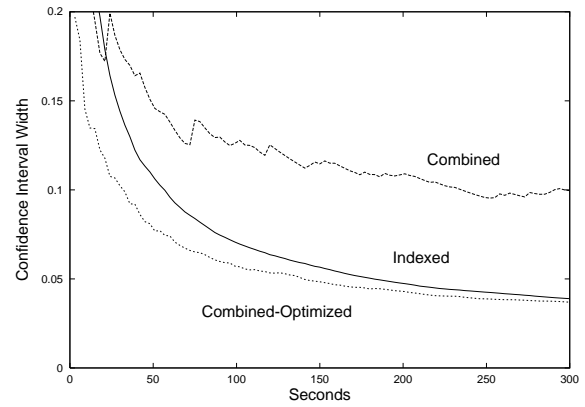


Figure 4: Experimental results for *test2*.

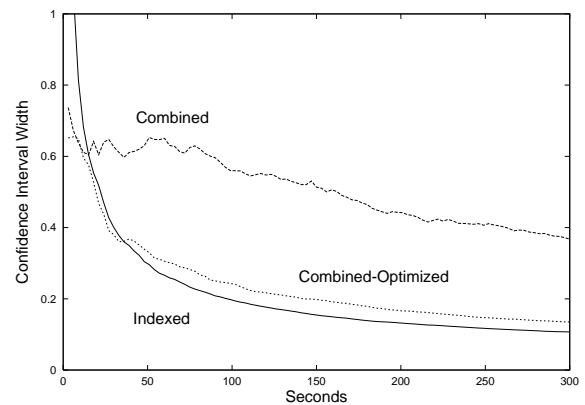


Figure 5: Experimental results for *test3*.

- In *test1*,  $e.A$  is produced using Zipf distribution with parameter 0.8, and  $e.B$  is true 50% of the time. The number of occurrences of each  $e.A$  is also Zipfian with a parameter of 0.8, as is the number of matches for each  $e.A$  in  $Sa1$ .  $s.C$  is true 10% of the time. The average record from  $Emp$  has 113 matching tuples in  $Sa1$  (though only 10% are accepted by the `WHERE` clause of the subquery).
- In *test2*,  $e.A$  is generated using a normal distribution with mean 0 and standard deviation 15; the sign of all negative values for  $e.A$  is inverted, and  $e.B$  is true 50% of the time. The number of occurrences of each  $e.A$  in  $Sa1$  is also generated with a normal distribution having mean 0 and standard deviation 15; a negative value indicates a tuple with no matches in  $Sa1$ . The number of repeats of  $e.A$  in  $Emp$  is similarly distributed. The average tuple from  $Emp$  has 5 matches in  $Sa1$ , and again an average of 10% of those are accepted by the `WHERE` clause of the subquery.
- In *test3*,  $e.A$  is generated similarly to *test2*, but a standard deviation of 1 is used. Again  $e.B$  is true 50% of the time. As in *test1*, the number of occurrences of each  $e.A$  is Zipfian, as is the number of matches for each  $e.A$  in  $Sa1$ , but with a milder skew of 0.4. The average tuple from  $Emp$  has 14 matches in  $Sa1$ , and again an average of 10% of those are accepted by the `WHERE` clause of the subquery.

In each case,  $Emp$  and  $Sa1$  are clustered randomly on disk and a secondary B+-Tree index is created on  $Sa1.D$ . We test three options for estimating the answer to the query: the Indexed Algorithm, the Combined Algorithm with no optimization (equal time is spent sampling for  $E$ ,  $Emp'$ , and  $Sa1'$ ), and the Combined Algorithm with full optimization. The optimization is set so as to minimize the variance after 100 seconds of query execution. All experiments are performed with a cold disk buffer using a 15,000 RPM SCSI disk and a 2.4GHz Pentium machine with 1GB RAM. For each data set, each algorithm is run for 300 seconds. For each experiment, the current confidence interval width at a 95% confidence level is plotted as a function of time in Figures 3, 4 and 5. In order to normalize the results across plots, the width is reported as a fraction of the current estimate. Thus, a confidence interval width of 0.3 means that the span of the 95% confidence interval is 30% as large as the current estimate. The width is computed assuming each estimator is normally distributed.

## Discussion

In general, the Combined Algorithm performs as well as the Indexed Algorithm for all three data sets, and in particular produced confidence bounds that are roughly two orders of magnitude better than the Indexed Algorithm for the most skewed data set (*test1*).

Note that the three data sets are ordered according to the amount of skew in the attribute that is aggregated over,

and that the advantage of the Combined Algorithm generally decreases as the skew in the data set decreases. In particular, since *test1* and *test2* are very similar except for  $e.A$ , it appears that if  $e.A$  has a great degree of skew, the Indexed Algorithm is essentially unusable. On the other hand, for the relatively well-behaved  $e.A$  in *test3*, both the Combined and Indexed Algorithms were roughly equivalent (though the estimate of the Indexed Algorithm was slightly better). It is also interesting to note that the experiments seem to indicate that the optimization to determine the optimal sampling rates is an absolute necessity. Without optimization, the Combined Algorithm can easily perform far worse than the Indexed Algorithm.

## 7.2 Properties of the Combined Algorithm

Given that the Combined Algorithm performed far better for some of the tests of the previous Section, it seems to be a natural choice for the queries considered in this paper. This Subsection describes an additional experimental evaluation of the Combined Algorithm, aimed at evaluating the statistical properties of the algorithm. There are two goals of these tests. First, we wish to obtain some experimental evidence that the mathematical derivations of expected value, variance, and bias correction of the Combined Algorithm described in the paper are in fact correct. Second, we wish to obtain an experimental validation that the normality assumption used to compute the confidence bounds in the previous Subsection is valid.

To test the statistical properties of the Combined Algorithm, we ran a series of tests using the following setup. Smaller versions of the two relations  $Emp$  and  $Sa1$  were created synthetically using methods very similar to those described in the previous Subsection, with each resulting relation having 1000 tuples. Much smaller relations than those tested previously were used in order to facilitate a very large number of algorithm repetitions during testing. We began each test by pre-sampling 100 tuples from  $Emp$ . We then retrieved the exact counts for those tuples from  $Sa1$ , and then ran the Combined Algorithm over  $Emp$  and  $Sa1$  using identical sampling rates for both relations. The process was repeated 100,000 times for each pair of data sets. These 100,000 runs were then used to compute empirically-observed values for  $\sigma^2(wN + U)$  and  $E[wN + U]$ .

## Discussion

Over several different data sets, after 200 samples from  $Emp$  and  $Sa1$ , we observed the following:

- The difference between the theoretical and observed values for  $\sigma^2(wN + U)$  ranged from 0.04% to 0.1%. Given that the variance of such experimentally-observed variances is often relatively large in practice, this is strong evidence for the correctness of our derivations.
- The difference between the theoretical and observed values for  $E[wN + U]$  was always less than 0.01%.

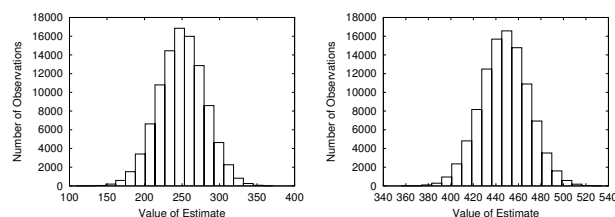


Figure 6: Empirical distributions for Combined Algorithm.

Again, this is good evidence of the correctness of our derivations.

Our experiments also strongly indicate that it is acceptable to assume that the estimate given by the Combined Algorithm is normally distributed when deriving confidence bounds using the variance estimates produced by our algorithms. For example, consider the empirical distributions given in Figure 6. These plots show a strong tendency toward a normal or bell-shaped curve. There was a slight skew observed in each of the empirical distributions, but our experiments seem to give some evidence that this skew is actually related more to  $U$  than to  $N$ , since increasing the size of  $E$  tends to reduce the skew (to a certain extent, this is not surprising since the central limit theorem applies to  $E$ , implying that if  $E$  has enough samples, it will in fact be normally distributed). For a more realistic application (where far more than 100 samples are used), we anticipate that this slight skew may vanish entirely.

## 8 Related Work

The work most closely related to our own is the online aggregation work from Haas, Hellerstein, and their collaborators [2, 10, 4, 5]. In particular, the Concurrent Algorithm is closely related to the ripple join [2], though the statistical analysis is very different. Though our work was clearly inspired by Haas and Hellerstein, ours is the first attempt to extend the online query processing framework beyond selection predicates and joins to subset-based SQL queries.

Approximation over queries similar to the subset-based SQL queries we consider has been attempted before, though not for online applications. Olken [11] considered the problem of sampling from various relational operators, including relational subtraction (which is closely related to SQL's NOT EXISTS). Olken's algorithms are most closely related to the Indexed Algorithm described in this paper. However, Olken's focus was not on developing online algorithms, and the Concurrent and Combined Algorithms we have described are far different from Olken's sampling strategy. Ganguly, Garofalakis, and Rastogi [3] consider the problem of tracking the answer to set-based COUNT queries over continuous update streams with limited main memory (and so approximation is mandatory), but their algorithms actually perform the opposite task that ours are designed to: they assume a known query and then build an updatable model to answer the query; we assume that we have a database given to us beforehand but an unknown

query workload consisting of arbitrary subset-based SQL queries.

## 9 Conclusions

We have considered the problem of performing online estimation over a very general class of subset-based SQL queries (queries with an inner query that is correlated to an outer query via a check for set membership). This greatly extends the class of queries that are amenable to sampling-based, online estimation. We have presented a formal statistical analysis of our estimators, and give a strong experimental argument for their utility with very large, disk-resident databases. The experiments verify that our estimators are able to quickly give accurate answers, especially when compared to the simple, index-based solution.

## References

- [1] G. H. Hardy and J. E. Littlewood and G. Polya. *Inequalities*. Cambridge University Press, 1988.
- [2] P. J. Haas and J. M. Hellerstein. Ripple joins for Online Aggregation. In *SIGMOD*, pages 287 – 298, 1999.
- [3] S Ganguly and M. N. Garofalakis and R. Rastogi. Processing Set Expressions over Continuous Update Streams. In *SIGMOD*, pages 265–276, 2003.
- [4] J. M. Hellerstein and P. J. Haas and H. J. Wang. Online Aggregation. In *SIGMOD*, pages 171–182, 1997.
- [5] J. M. Hellerstein and R. Avnur and A. Chou and C. Hidber and C. Olston and V. Raman and T. Roth and P. J. Haas. Interactive data Analysis: The Control Project. In *IEEE Comp.* 32(8), pages 51 – 59, 1999.
- [6] N. L. Johnson and S. Kotz and A. W. Kemp. *Univariate Discrete Distributions*. Wiley Series in Probability, 1993.
- [7] W. Cochran. *Sampling Techniques*. Wiley and Sons, 1977.
- [8] Transaction Processing Council. *TPC-H Benchmark*. <http://www.tpc.org>, 2004.
- [9] Terrence L. Fine. *Feedforward Neural Network Methodology*. Springer, 1999.
- [10] P. J. Haas. Large-sample and Deterministic Confidence Intervals for Online Aggregation. In *Statistical and Scientific Database Management*, pages 51–63, 1997.
- [11] F. Olken. Random Sampling from Databases. In *Ph.D. Dissertation*, 1993.
- [12] J. Shao. *Mathematical Statistics*. Springer, 2nd Edition, 2003.
- [13] S. K. Thompson. *Sampling*. Wiley and Sons, 2002.