

Relational Confidence Bounds Are Easy With The Bootstrap*

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

Abstract

Statistical estimation and approximate query processing have become increasingly prevalent applications for database systems. However, approximation is usually of little use without some sort of guarantee on estimation accuracy, or “confidence bound.” Analytically deriving probabilistic guarantees for database queries over sampled data is a daunting task, not suitable for the faint of heart, and certainly beyond the expertise of the typical database system end-user. This paper considers the problem of incorporating into a database system a powerful “plug-in” method for computing confidence bounds on the answer to relational database queries over sampled or incomplete data. This statistical tool, called the *bootstrap*, is simple enough that it can be used by a database programmer with a rudimentary mathematical background, but general enough that it can be applied to almost any statistical inference problem. Given the power and ease-of-use of the bootstrap, we argue that the algorithms presented for supporting the bootstrap should be incorporated into any database system which is intended to support analytic processing.

1 Introduction

Statistical estimation and approximate query processing (AQP) are relevant whenever an answer to a query must be computed from a summary or a sample of a data set, rather than the complete data set itself. AQP is particularly common in data warehousing applications, which may store samples from the complete data set to decrease response time, among other reasons. See the Aqua project from Bell Labs [4] for an example of this. Furthermore, in many data warehouses, it is common for the data that are actually recorded and stored in the warehouse to be a sample of a “universal” data set that is infeasible to collect in its entirety. This is particularly common in scientific applications. For example, a telescopic sky survey may cover only part of the sky, or some portions of the sky may be photographed at a higher resolution than other portions.

In general, estimates over incomplete or sampled data are only of limited utility if they are not accompanied with some sort of *confidence bounds* that give the user an idea of the accuracy of the

approximate answer. *Confidence bounds* assure the user that the actual answer to the query is within a certain interval with a user-specified probability p (typically 95% or 99%). For example, an answer and its associated confidence bounds might be “The percentage of fraudulent transactions in 1999 was between 2.4% and 2.8%, with 95% confidence.” Such bounds are usually derived by associating a probability distribution with the actual answer to the query, and then computing the interval around the distribution’s mode which contains a total mass of p .

Since statistical estimation is an important task demanded by many applications of database technology, a natural question is: How can a database system be augmented to provide the typical application programmer or data analyst with the ability to determine how accurate estimates over incomplete data are, by providing the option of computing confidence bounds along with the answer to a query over an incomplete or sampled data set?

An obvious solution would be to incorporate some of the existing techniques for computing analytic, “formula-based” confidence bounds [6][7][8][9][11][12][15] into the system. Many different bounds have been developed by database researchers over nearly two decades. However, there are two tremendous problems with relying on analytic derivation of confidence bounds:

- *Confidence bounds must be derived for every possible class of query, and this can be very hard to do.* While it may be possible to shield database end-users from the complexity associated with computing confidence bounds by building bounds directly into the system, if an application demands that a new type of query be answered, then confidence bounds must be re-derived for that new type of query. Even now, there are many, simple relational database queries for which no confidence bounds exist. For example, the confidence bounds derived by Haas and Hellerstein for samples over a join of two relations [7][8] are not valid in the case of a self-join over a single, sampled database table. This leads us to another problem with analytically-derived confidence bounds: they can be difficult to derive. There is no single formula or one-size-fits-all derivation methodology. Furthermore, for many queries, developing usable analytic confidence bounds may be nearly impossible if the queries to be estimated are more complicated than a mean or a sum. For example, it would be very difficult to derive confidence bounds that could be used when a user wishes to estimate the projected sales for the next few years based on a least squares fit to sales figures from the last few years.

- *Even after they have been derived, analytic confidence bounds can be hard to use.* Even if confidence bounds *can* be derived with little trouble, it is very difficult to shield the end-user from all of the underlying complexity. Confidence bounds may be tight for one data set, but not for another, or they may be correct for one data set, and not for another. For example, many tight confidence bounds are based on the *central limit theorem* (CLT),

*This material is based upon work supported by the National Science Foundation under Grant No. 0347408.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA
Copyright 2005 ACM 1-59593-060-4/05/06 \$5.00

which in some situations can be taken as justification that the estimation error for a query is normally distributed¹. If instead one wants to play it safe and not make any normality assumptions, one can appeal to distribution-free bounds such as the Vysochankii-Putunin inequality [18] or Chebyshev’s or Hoeffding’s inequalities. However, CLT bounds can be far tighter than those produced by these distribution-free inequalities. If the CLT *does* hold, it is obviously preferable to use the bounds it provides; if not, a distribution-free bound is preferable. Choosing which way to go is often far from obvious.

Pulling Oneself Up By One’s Bootstraps

Fortunately, there is a general-purpose technique for providing confidence bounds that can address these issues. The last 25 years have seen the widespread acceptance of *experimental* or *simulation-based* confidence bounds in statistics. Simulation-based confidence bounds eschew analytic derivation and parametric assumptions in favor of raw computational power. Simulation methods are easy to use, even for non-experts. They have the important advantage that they offer a generic technique for computing confidence bounds for most statistical functions, without any analytic derivation. Simulation-based methods are already important components of popular statistical software packages such as SAS [20]. Given their power and generality, we argue that simulation-based estimation techniques are prime candidates for integration into data management software.

In this paper, we concentrate on providing database support for one type of very popular, simulation-based confidence bound, called the *bootstrap* [2]. At the highest level, the bootstrap is extremely simple, and should be usable by most end-users of database systems. To provide statistical confidence bounds, the bootstrap method simply re-runs the statistical estimator hundreds or thousands of times over a large number of simulated data sets. These simulated data sets are produced by *resampling* the underlying database tables. After the estimator has been run over all of the simulated data sets, then confidence bounds (or other important characteristics such as the variance or standard error) are obtained by looking at the estimator’s distribution with respect to the simulated data sets. For basic bootstrap inference problems, no analytic formulas are required.

This Paper’s Technical Contributions

Given the power and versatility of the bootstrap and the fact that it is easily used by anyone with a rudimentary understanding of statistics or mathematics, we assert that it is an important tool for use with incomplete data and a prime candidate for inclusion in modern, large-scale, analytic processing systems.

However, the primary problem with using the bootstrap in a database environment is performance. The obvious way to provide for bootstrapped confidence bounds would be to simply run the query several thousand times over the database, each time on a new set of samples from the underlying database tables. This would result in a query response time thousands of times slower than if one were to simply run the query in isolation, without resampling. Obviously, this is not practical.

The main technical contribution of the paper is the development of two bootstrapping algorithms that actually function as middle-

ware that sit on top of (and possibly below) the query processing engine. The algorithms have several obvious advantages:

- Aside from moving grouping and aggregation into a post-processing phase, our algorithms do not require any modification to the underlying query processing engine, and so could be easily integrated into existing DBMS.
- The additional time required by our algorithms to re-run the query many thousands of times over the bootstrapped sample can actually be *less than* the time to evaluate the query for the first time.
- One of our algorithms is *adaptive*, in the sense that though it can function with only a small amount of main memory, as more main memory is made available, the algorithm will run faster.

Paper Organization

The remainder of the paper is organized as follows. In Section 2, we give a brief introduction to the bootstrap method. Section 3 considers the problem of using the bootstrap with relational databases. An efficient RDB bootstrapping using the *Tuple Augmentation (TA)* algorithm is presented in Section 4, and an alternative, the *On-Demand Materialization (ODM)* algorithm, is given in Section 5. Section 6 details a set of experiments aimed at benchmarking performance of bootstrapping. We describe related work in Section 7, and conclude the paper in Section 8.

2 The Bootstrap

It has long been understood that *statistical inference* is very difficult. Statistical inference refers to the problem of inferring the characteristics of a probability distribution by analyzing samples from that distribution. One difficulty associated with statistical inference is that in practice, outside of certain properties like the mean, it can be extremely difficult to infer characteristics of a distribution. Furthermore, the process of statistical inference is necessarily riddled with assumptions that may or may not be true for a given data set. Motivated by these sorts of problems and facilitated by the advent of inexpensive computational power, the last 25 years have seen the widespread acceptance of *experimental* or *simulation-based* confidence bounds in statistics. Such confidence bounds follow from a general set of high-level principles, and do not require a separate analytic derivation for each new problem.

2.1 Bootstrapping Basics

The most popular of these simulation-based techniques is the *bootstrap*. The basic bootstrap is exceedingly simple, and we describe it now with an example. Imagine that we have ten numbers sampled from a distribution F and we want to estimate the mean of the distribution and provide 99.7% confidence bounds on our estimate². The ten numbers sampled are:

$$\mathbf{x} = \{-2.14, -1.98, -0.83, -0.82, 0.36, 0.42, 1.24, 1.26, 1.30, 1.92\}$$

It is very easy to calculate that the mean of this set is $\mu = 0.074$, and if the samples are truly random, this will be an unbiased estimate for the mean of F (if F arises from a finite population, the fact that we can apply essentially the same function to \mathbf{x} to estimate the mean as we can to F to compute the mean is referred to in boot-

1. One often-overlooked fact is that the central limit theorem is a *limiting* theorem, which only guarantees that the observed error is normally distributed as the size of a sample becomes arbitrarily large. In practice, “large” can be 20 samples, or it can be millions of samples.

2. In our example, we choose the 99.7% confidence level for simplicity; it is a well-known fact that 99.7% of the mass of a normally distributed random variable is found within three standard deviations of the mean.

strap parlance as the *plug-in principle*, and the function

$$f(\mathbf{x}) = 1/n \sum_{i=1}^n x_i$$

is the *plug-in estimator* of the mean).

Now, imagine that we suspect that in reality these numbers were sampled from a normal distribution with a standard deviation of one. Given this information, we could compute confidence bounds on our estimate using the standard set of rules for the normal distribution: the real mean would be $\mu \pm 3/\sqrt{10} = 0.074 \pm 0.949$ with 99.7% confidence. If \mathbf{x} was really sampled from a normal distribution with a standard deviation of one, then these confidence bounds are in fact correct.

The problem is that in practice, it is not reasonable to expect that we know the standard deviation or even the parametric distribution function for f beforehand. Or, we may be forced to deal with a probability distribution that does not provide us with a convenient set of rules for deriving confidence bounds, or we may be dealing with a function f for which bounds are difficult or impossible to derive. In this case, we could make use of the *bootstrap*.

The bootstrap works by *resampling* the original data set many times, and empirically computing the distribution of the estimated query answers. To resample a data set $\mathbf{x} = (x_1, x_2, \dots, x_n)$, we simply create a new data set \mathbf{x}^* where each $x_i^* \in \mathbf{x}^*$ is randomly assigned the value of some $x_j \in \mathbf{x}$. Any x_j can be chosen for the assignment, and all are chosen with identical probability. For example, one possible bootstrap sample from \mathbf{x} is:

$$\mathbf{x}^* = \{-2.14, -2.14, -0.83, -0.82, 0.36, 0.36, 0.42, 1.26, 1.92, 1.92\}$$

Note that since all $x_i^* \in \mathbf{x}^*$ are independent, the bootstrap sample is taken *with replacement*, and samples may be repeated.

The task we are most concerned with in this paper is computing confidence bounds at $\alpha\%$ and $(1 - \alpha)\%$ for the value of $f(\mathbf{x})$. The computation of our confidence bounds using the bootstrap is given as Algorithm 1. The pairs computed by this algorithm are known as *bootstrap percentiles*, and are a first-order accurate approximation to the correct confidence bounds (see Efron and Tibshirani [2], Chapter 13). The value b in this algorithm denotes the number of *bootstrap replications*. Estimates for other properties of $f(\mathbf{x})$ such as the standard error and the variance can also be given using the bootstrap. In our particular example, running Algorithm 1 with a value of $b = 20,000$ gave us 99.7% confidence bounds of -1.24151 and 1.2518, which is a good estimate of the correct confidence bounds, but requires no knowledge of the properties of the underlying data distribution.

Finally, we note that in general, $f(\mathbf{x})$ does not need to be a plug-in estimator for the bootstrap to be applicable (that is, it does not have to be the same function that we would apply to F to compute the actual value of the statistic we are trying to estimate). Often, other estimators have better accuracy; in this case, they could be used instead. For example, extensive work has been done in the database area to develop estimators for the number of distinct values in a relation that are more accurate than the obvious plug-in estimator $f(\mathbf{x}) = (\# \text{ distinct values in } \mathbf{x})$ [5][1].

2.2 Bootstrap Caveats

The obvious advantage of the bootstrap is that it is an exceedingly simple and mechanical algorithm which can give confidence bounds for any arbitrary estimator f , no matter how complicated f is. No assumptions about the underlying data distribution are needed. Furthermore, there is a large body of theoretical work backing up the correctness of the bootstrap (see Hall for the most commonly referred to theoretical treatment of the bootstrap [10]).

Algorithm 1: Bootstrapped confidence intervals

- (1) Vector $B = \langle \rangle$
- (2) For $i = 1$ to b do
- (3) Create a new bootstrap sample \mathbf{x}^*
- (4) Append $f(\mathbf{x}^*)$ to B
- (5) Sort the contents of B
- (6) Return the pair $B[b \times \alpha]$, $B[b \times (1 - \alpha)]$ as the low/high confidence bounds (if $b \times \alpha$ is not integral, then estimate $B[b \times \alpha]$, $B[b \times (1 - \alpha)]$ using standard histogram techniques)

Despite the power of the bootstrap, the method is not entirely without pitfalls. In general, there are a few caveats that must be considered when applying the bootstrap:

- The bootstrap works well for determining the properties of “smooth” statistical functions over a data set but can be useless for certain non-smooth functions. “Smooth” can be defined intuitively as follows. Imagine that we re-sample a data set to create a new data set \mathbf{x}^* , and compute $f(\mathbf{x}^*)$. Then we remove one sample from \mathbf{x}^* and replace it with a new random sample from \mathbf{x} . If $f(\mathbf{x}^*)$ cannot change significantly in response to this slight change in the sample, then f is smooth and amenable to bootstrap analysis. However, certain functions are not smooth and should not be bootstrapped. For example, the obvious plug-in estimator for estimating the minimum value of a data set (taking the minimum value in the sample) should not be bootstrapped.
- Enough bootstrap samples must be taken that the distribution of $f(\mathbf{x}^*)$ that is observed is a good approximation of the distribution of $f(\mathbf{x}^*)$ over all of the possible simulated data sets. For many applications, 200 simulated data sets is enough, but for some applications many more can be required (see Efron and Tibshirani [2] for a discussion of this). This can lead to very expensive computations; reducing this expense in a database environment is at the heart of this paper.
- In general, the objects in the data set that are to be resampled should be independent, identically distributed samples from the underlying data distribution. The bootstrap must be used with care in applications where the missing data may not be distributed in the same way as the data stored in the database. For example, an incomplete data set may result when records are discarded during data integration due to integrity constraint violations. Such records may tend to be older, and hence they may have a different distribution than other records in the database; the bootstrap may not be applicable in such a situation. Still, the i.i.d. requirement is not specific to the bootstrap; almost any parametric technique for statistical inference will have similar requirements.
- Finally, we mention that the very simple bootstrap described in Algorithm 1 can be sensitive to issues with *bias* and *acceleration* in the standard error of the bootstrapped estimator (acceleration refers to the rate of change of the standard error of an estimated parameter with respect to the true value). One strength of the bootstrap is that it can automatically correct for problems with bias and acceleration, though the simple bootstrap described in this paper is susceptible to these problems. For greater accuracy in accessing the correct confidence intervals, slightly more complicated analysis of the bootstrap is required (for example, one bias-corrected and accelerated bootstrap confidence bound is the BC_a method (see Efron and Tibshirani [2] Chapter 22 and the references contained therein). Even so, Algorithm 1 can be

expected to give very good results that will often be superior to standard analytic techniques that make use of parametric assumptions.

Despite these drawbacks, we assert that the bootstrap should be considered an important tool for use in statistical estimation and AQP in database systems.

3 The Bootstrap and the Relational Model

In this Section, we consider the problem of using the bootstrap to gauge the accuracy of an estimate over an incomplete or sampled data set stored in a relational database, thereby creating a generic and simple plug-in method for deriving confidence bounds in the context of a relational database system. The next two Sections then consider the algorithmic issues associated with incorporating the bootstrap into a relational database system.

3.1 Bootstrapping Relational Data

Imagine that we have a standard select-project-join aggregate query issued over a set of database tables R_1, R_2, \dots, R_y . In this paper, we consider the class of queries that can be written in SQL as follows:

```
SELECT f(R1.att1, R1.att2, ..., R2.att1, R2.att2, ...)
FROM R1, R2, ..., Ry
WHERE expression (R1.att1, R1.att2, ..., R2.att1, R2.att2, ...)
```

In this query, f may encode a GROUP BY, and can include any built-in SQL aggregate function or may be a user-defined function of arbitrary complexity.

Now, imagine that database tables R_1, R_2, \dots, R_x are incomplete or sampled for $x \leq y$. The question we seek to answer is: what is the distribution of f , given that a subset of the tables are considered to be sets of samples from one or more underlying distributions? Knowing the distribution of f would allow us to derive confidence bounds for f , along with other vital information. As described in the Introduction, an analytic derivation of these confidence bounds is not easy in many circumstances. However, it is easy to experimentally derive the distribution of f using the bootstrap. Consider the following version of the previous query:

```
SELECT BOOTSTRAP f(R1.att1, R1.att2, ..., R2.att1, R2.att2, ...)
FROM Rx+1, Rx+2, ..., Ry
INCOMPLETE R1, R2, ..., Rx
WHERE expression (R1.att1, R1.att2, ..., R2.att1, R2.att2, ...)
RESAMPLE b TIMES
```

Without giving a formal semantics for this query, it essentially does nothing more than rerun the original SQL query b times. However, rather than running the query each time over the original database tables, the tables listed as INCOMPLETE are resampled for each of the b runs. The result of the query is a new relation where each of the b results is a tuple in the relation. The process is described in Algorithm 2. The end result of this query is simply a relation which lists, in order, the result of re-running the original query b times. If $b = 1000$, in order to obtain the 90% confidence bounds using the percentile bootstrap described in the previous section, one only has to query the *Result* relation:

```
SELECT *
FROM Result
WHERE i = 50 OR i = 951
```

Algorithm 2: Bootstrapping a relational database query

- (1) Vector $temp = \langle \rangle$
- (2) For $i = 1$ to b , do:
- (3) For $j = 1$ to x , do:
- (4) Resample R_j to create R_j^*
- (5) Append the result of the following query to $temp$:
- (6) SELECT $f(R_{1.att1}, R_{1.att2}, \dots, R_{2.att1}, R_{2.att2}, \dots)$
- (7) FROM $R_1^*, R_2^*, \dots, R_x^*, R_{x+1}, R_{x+2}, \dots, R_y$
- (8) WHERE $expression (R_{1.att1}, R_{1.att2}, \dots, R_{2.att1}, \dots)$
- (9) Relation $Result = \{ \}$
- (10) Sort $temp$
- (11) For $i = 1$ to b , do:
- (12) $Result = Result \cup \{ (i, temp[i]) \}$

3.2 Can It Really Be So Easy?

The above procedure is compatible with any function f (not just an estimator for the standard SQL aggregate functions such as SUM, AVERAGE, COUNT, etc.). It is compatible with any valid expression in the WHERE clause of the query, and would work with GROUP BY queries as well. In this Section, we describe why the bootstrap approach over relational data is so simple and yet widely applicable, and contrast this with some of the difficulties faced in deriving analytic confidence bounds over database queries.

3.2.1 Analytic Bounds Over Relational Joins

To highlight the simplicity and power of the bootstrap method, we first contrast it with the difficulties associated with analytically derived confidence bounds over relational joins. Consider the following query:

```
SELECT COUNT (*)
FROM R, S
WHERE expression (R.att1, R.att2, ..., S.att1, S.att2, ...)
```

Imagine that we have two relations R' and S' that are 10% samples of R and S respectively. The following query will serve as an unbiased estimate for the answer to the original query:

```
SELECT 100 * COUNT (*)
FROM R', S'
WHERE expression (R.att1, R.att2, ..., S.att1, S.att2, ...)
```

Now, imagine that we want to characterize the distribution of the answer to this query, in order to derive confidence bounds for the estimate it provides. One problem that we face when analytically deriving such bounds is that it is very difficult to derive confidence bounds for anything other than the mean of a distribution. As a result, the natural way to view this problem is to pick either R' or S' , and view each of the tuples from this relation as a single sample from a distribution whose mean we are trying to estimate. This is depicted in Figure 1. Each tuple from R' is treated as a sample from a distribution whose mean is the answer to our query. The value of the sample is determined by counting the number of tuples from S' that each tuple from R' matches up with, and multiplying by 80 (the number of tuples in R) and again by 10 (since S' is a 1/10th sample of S).

However, the new problem that we have created by re-casting this problem as a mean-estimation problem is that the values provided by the pair of tuples R_i and R_j are no longer independent,

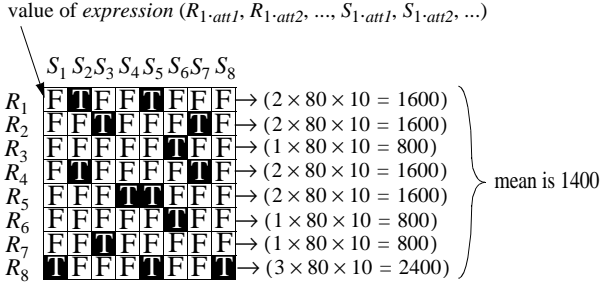


Figure 1: Treating the estimation of a COUNT (*) query as the estimation of the mean of a distribution.

since expression $(R_{i.att1}, \dots, S_{k.att1}, \dots)$ and expression $(R_{j.att1}, \dots, S_{k.att1}, \dots)$ both depend on S_k . Thus, deriving bounds becomes far more difficult. Haas and Hellerstein have managed to overcome this problem in the context of standard aggregation functions over a simple relational join [7][8], but not without substantial efforts that are far beyond the abilities of even mathematically sophisticated database end-users. Furthermore, their analysis is defeated by desirable modifications to the problem definition, such as allowing self-joins over samples.

3.2.2 A Comparison With the Bootstrap

As demonstrated above, the problem with the derivation of analytic confidence bounds over a join is that we have to recast the problem as the problem of estimating a mean, thereby destroying the independence of the values associated with the tuples making up a relation.

In contrast, if we ignore for a moment the extra computational cost, the application of the bootstrap to this sort of relational query is almost trivial. Like most traditional methods, bootstrap analysis does require that the individual data points be independent. However, unlike traditional statistical inference, the bootstrap is applicable to any function over the underlying data, no matter how complicated (such as an aggregate function applied to a join of the constituent tuples). Thus, there is no need to re-cast the inference problem as estimating the mean of a distribution, and so we do not introduce any correlation into the data that was not present prior to the analysis. Furthermore, applying the bootstrap simultaneously to a function estimating a statistic involving more than one data distribution (as is the case in Algorithm 2 if more than one relation is listed as being INCOMPLETE) is not problematic; the bootstrap is quite often applied to more complicated structures such as this (see Efron and Tibshirani, Chapter 8 [2]).

4 Efficient RDB Bootstrap: The TA Algorithm

While the bootstrap has obvious advantages, the drawback is speed. Imagine that we wish to re-run a relational database query 1000 times to derive confidence bounds using Algorithm 2. The algorithm will resample each incomplete relation 1000 times, and then re-run the underlying query for each sampled version of the database. As a result, we will expectedly need more than 1000 times as much time as was required to run the original query. Such a massive performance hit would make the technique largely unusable. The question at the heart of this Section is: what can be done to alleviate this substantial performance penalty, in order to make bootstrap resampling usable in real-life database systems?

4.1 Conservation of Tuples

The obvious place to start when trying to alleviate the cost associated with scores of bootstrap trials is to somehow avoid running

the underlying query more than one time, even though many thousands of trials may be prescribed by the user. For a large class of relational database queries, there is actually a fairly simple way to accomplish this. To describe why it is possible to get away with running the underlying query only one time, we reconsider the following query from Section 3.1:

```
SELECT BOOTSTRAP f(R1.att1, R1.att2, ..., R2.att1, R2.att2, ...)
FROM Rx+1, Rx+2, ..., Ry
INCOMPLETE R1, R2, ..., Rx
WHERE expression (R1.att1, R1.att2, ..., R2.att1, R2.att2, ...)
RESAMPLE b TIMES
```

Then,

- Let S be the multi-set of tuples that f will operate over if the underlying query is run directly over the relations R_1 through R_y . In other words, S is simply the multi-set that would be the result of the query if the aggregate function f were removed, and replaced by a SELECT *.
- Let S^* be the multi-set of tuples that f will operate over if the underlying query is run using resampled versions of R_1, R_2, \dots, R_x rather than running the query over R_1, R_2, \dots, R_x directly.

The key observation that will allow us to run the query only one time is as follows:

Lemma 1: For any resampled versions of R_1, R_2, \dots, R_x , $RemoveDuplicates(S) \supseteq RemoveDuplicates(S^*)$.

Proof: If we resample a relation R to create a new relation R^* , then the set of tuples in R^* must be a set of the tuples in R . Thus, joining R^* with any relation must produce a subset of the tuples that would have been produced by joining R with the same relation.¹ ■

This guarantees that the tuples resulting from the non-resampled version of the underlying query are always a superset of the tuples that result from running the query over any subsampled relations. The net result of this is that if we run the underlying query only once using the original relations R_1, R_2, \dots, R_y , then we will produce a relation S that contains a superset of the set of tuples that would have been produced had we replaced R_1, R_2, \dots, R_x with resampled versions of those same relations. Post-processing of S can then give us any number of results that would have been obtained had we originally run the query over resampled versions of R_1, R_2, \dots, R_x .

For example, consider the incomplete relations SUPERVISES (BOSS, EMP) and AGE (EMP, YEARS), given in Figure 2. We wish to estimate the answer to the question, "What is the average age of the employees supervised by Mr. Smith?" To do this, we make use of the following SQL query:

```
SELECT AVG (YEARS)
FROM SUPERVISES, AGE
WHERE BOSS = "Mr. Smith" AND
SUPERVISES.EMP = AGE.EMP
```

1. This result does not hold for SQL queries in general. Still, Lemma 1 does remain valid for a large class of SQL queries (the exceptions to this include SQL queries with negations over subqueries, such as the NOT IN clause), though more complicated queries are beyond the scope of this paper.

SUPERVISES:	AGE:	Resampled SUPERVISES:	Resampled AGE:
("Mr. Smith", "John")	("John", 33)	("Mr. Smith", "John")	("John", 33)
("Mr. Smith", "Sue")	("Bob", 38)	("Mr. Smith", "Joe")	("Bob", 38)
("Mr. Smith", "Joe")	("Jeff", 28)	("Mr. Smith", "Joe")	("Bob", 38)
("Mr. Smith", "Sam")	("Tom", 57)	("Mr. Smith", "Sam")	("Tom", 57)
("Ms. Jones", "Tom")	("Jim", 39)	("Mr. Smith", "Sam")	("Joe", 42)
("Ms. Jones", "Jim")	("Joe", 42)	("Ms. Jones", "Tom")	("Joe", 42)
("Ms. Jones", "Jane")	("Jen", 45)	("Ms. Jones", "Jane")	("Jen", 45)
("Ms. Jones", "Joe")	("Ann", 23)	("Ms. Jones", "Jane")	("Jen", 45)

Figure 2: Relations used for a bootstrapped join.

Using this query, the set of tuples S used to compute the average is:

```
("Mr. Smith", "John", "John", 33)
("Mr. Smith", "Joe", "Joe", 42)
```

for a final estimate of 37.5.

Now, imagine that we resample the two relations, as shown on the right hand side of Figure 2. We again run the query, this time over the resampled relations. This time, the set of tuples S^* used to compute the average is:

```
("Mr. Smith", "John", "John", 33)
("Mr. Smith", "Joe", "Joe", 42)
("Mr. Smith", "Joe", "Joe", 42)
("Mr. Smith", "Joe", "Joe", 42)
("Mr. Smith", "Joe", "Joe", 42)
```

for an estimate of 40.2.

The key observation made in Lemma 1 is that after removing duplicates, the set of tuples used to compute the bootstrapped estimate contains no tuples that were not present before resampling. The process of resampling the underlying relations cannot create any new tuples; it can only change the number of times that each tuple in S appears in S^* . Specifically, if a tuple t from a relation R appears n times in R^* , then any tuple in S that depends on t will appear an additional factor of n times in S^* . In our example, ("Mr. Smith", "Joe") appears twice in the resampled version of SUPERVISES, and ("Joe", 42) appears twice in the resampled version of AGE. As a result, the tuple ("Mr. Smith", "Joe", "Joe", 42) appears four times in S^* .

4.2 More Efficient Bootstrap Repetitions

Given this observation, it becomes possible to simulate re-running the query many times over each set of resampled input relations by simply pre-computing the resampled relations, and then remembering how many times each tuple appears in each resampled relation. The simple algorithm can be used to resample a relation R , even if it is too large to be stored in main memory (see Algorithm 3).

We can then use the counts computed by Algorithm 3 to compute the number of repetitions of every tuple from S in each bootstrapped repetition of the original query. If we assume that all of those counts require too much storage to be buffered in main memory, then the simplest, I/O-efficient algorithm for managing those counts is to "attach" the required information to each tuple, and then rely on the DBMS to manage them throughout the query evaluation process. Post processing of the query result can then give us the final result of the bootstrapped query. The algorithm follows:

- (1) For $1 \leq i \leq x$, let $R_i[j]$ denote the j th tuple in relation i . We begin by pre-computing the number of times that $R_i[j]$ appears in each resampled version of R_i . This can be done in $O(b \log |R_i|)$ time by generating $b|R_i|$ uniformly distributed random numbers from 1 to $|R_i|$, and sorting them appropriately

Algorithm 3: Resampling a relation R

- (1) Vector $temp = \langle \rangle$
- (2) For $i = 1$ to $|R|$, do:
- (3) Generate a random number from 1 to $|R|$
- (4) Append it to $temp$
- (5) Sort $temp$ (if large, use an external sorting algorithm)
- (6) Scan $temp$ from start to finish; for each i in $temp$:
- (7) Count the number of times i appears in $temp$; this is the cardinality of t_i in R^*

(see Algorithm 4A). For $1 \leq k \leq b$, let $R_i[j].numAppear[k]$ denote the number of times the j th tuple from the i th relation appears in the k th resampled version of that relation.

- (2) Next, run the underlying SQL query to produce S , the set of all tuples that will be used by the aggregate function f to compute the answer to the query. This is done using the normal DBMS query execution engine over the data stored in the database, with the single exception being that each tuple from each incomplete relation has been augmented with the array $numAppear$. This array is "carried" with the tuple throughout the execution of the query, and treated as an additional attribute of the tuple.
- (3) Finally, post-process S to produce each S_i^* , for $1 \leq i \leq b$. Once each S_i^* is computed, we complete the process by running the aggregate function f over each S_i^* to produce the final result of the bootstrap query. These last few steps can be done easily using Algorithm 4B.

Note that in many cases, it will not be necessary to materialize each S_i^* , as is done by Algorithm 4B. If f can be computed in a single pass (as is the case with many common aggregate functions such as AVG and SUM), then the value for f can be maintained on-the-fly as the tuples in S_i^* are computed.

4.3 Compressing the Counts

The most obvious drawback of this technique is that an array of counts must be associated with each tuple. This array must be "carried" with each tuple throughout the query evaluation process. Since b (the number of bootstrap repetitions) may be large, the size of this array can be large. A b value of 1000 or larger is not out of the question, and associating an array of 1000 integers with every tuple would clearly lead to a significant performance hit. It is conceivable that the size of each incomplete relation could be increased by several hundred times because of these arrays. However, most of those counts will be very small numbers.¹ Thus, it seems almost mandatory to use some sort of compression in order to decrease the cost associated with propagating them throughout the query evaluation plan.

Because each count details the number of times that each tuple from the relation appears in the resampled version R^* of R , the counts in the array of a single tuple from R are binomially distributed with $|R|$ trials and probability of success of $1/|R|$. For large $|R|$, the `gzip` implementation of the Lempel-Ziv compression

1. Though the counts across tuples are multinomially distributed for each bootstrap repetition, the counts in a single array are binomially distributed with a mean of one; for a large relation, 99.9999% of the counts will be less than 10.

Algorithm 4A: Pre-processing R to produce tuple counts

- (1) Vector $temp = \langle \rangle$
- (2) Let r be the pair $[randNum, repNum]$
- (3) For $i = 1$ to b , do:
- (4) For $j = 1$ to $|R|$, do:
- (5) $r.randNum =$ Random number between 1 and $|R|$
- (6) $r.repNum = i$
- (7) Append r to $temp$
- (8) Sort $temp$ (if required using external sorting algorithm) first based on $r.randNum$ and then on $r.repNum$
- (9) Let $recNum = 0$
- (10) For $i = 1$ to $b|R|$, do:
- (11) If $temp[i].randNum \neq recNum$
- (12) $recNum = temp[i].randNum$
- (13) $R[recNum].numAppears[temp[i].repNum]++$

Algorithm 4B: Post-processing S to produce $Result$

- (1) Relation $Result = \{ \}$; Vector $temp = \langle \rangle$
- (2) Initialize each S_i^* to $\{ \}$;
- (3) For each tuple t in S , do:
- (4) For $i = 1$ to b , do:
- (5) Let $numTimes = 1$
- (6) For each tuple t_j that was concatenated to form t
- (7) $numTimes \times = t_j.numAppear[i]$
- (8) Add $numTimes$ copies of t to S_i^*
- (9) For $i = 1$ to b , do:
- (10) Append $f(S_i^*)$ to $temp$
- (11) Sort $temp$
- (12) For $i = 1$ to b , do:
- (13) $Result = Result \cup \{ (i, temp[i]) \}$

algorithm [19] is able to compress such a list of binomially distributed numbers to slightly less than 3.5 bits per count. This means that associating an array of 1000 counts with each integer (allowing us to simulate $b = 1000$ bootstrap repetitions) will add about 437 bytes to each tuple.

However, there is room to do even better, as described by Lemma 2:

Lemma 2: As the size of the resampled relation $|R| \rightarrow \infty$, there exists an optimal compression scheme requiring ≈ 1.88 bits per count to compress the count array associated with each tuple.

Proof outline: The proof is based on Shannon's famous result that n samples from a distribution can be coded using

$$\sum_{i=1}^n p_i \log_2 p_i \text{ bits [17]. In this case, the distribution in question}$$

is the binomial distribution described above. ■

Thus, it is possible to decrease the cost of 3.5 bits per count by nearly 50% by using some optimal compression scheme. Unfortunately, Lemma 2 does not tell us what that compression scheme is. Fortunately, a very simple algorithm can be derived that will require exactly 2 bits per count, or within around 6% of the optimal. This is given as Algorithm 5 above.

Lemma 3: To encode the b resampled versions of a relation R , Algorithm 5 requires $2b|R|$ bits total.

Algorithm 5: Compressing a list of counts

- (1) Input: A list L of b binomially distributed counts
- (2) Output: a bit string representing those counts
- (3) For int $i = 1$ to ∞ do:
- (4) For int $j = 1$ to $length(L)$
- (5) If $L[j] > i$ then output 1
- (6) Otherwise, delete $L[j]$ from L and output 0
- (7) If $length(L)$ is 0 then exit

Proof outline: The proof follows directly from Algorithm 5; each count averages one 1 and one 0 in all compressed lists. ■

5 The ODM Algorithm

Even though the Tuple Augmentation (TA) algorithm of Section 4 requires that we run the underlying database query only once in order to make use of the bootstrap, it still has two disadvantages:

- (1) Even using the compression of Algorithm 4, we still blow up the size of the database tuples. At two bits per count, simulating 1000 bootstrap repetitions will require that 250 bytes be carried with each tuple from an incomplete relation, throughout the query evaluation process. This may cause a significant performance hit.
- (2) A less obvious problem is that we may do a tremendous amount of work to associate counts with tuples that will never contribute to the result of the bootstrap. Clearly, some tuples will never be joined with any other tuples so as to satisfy the relational selection predicate present in the underlying query. Ideally, we would never have to do any work over such tuples.

Both of these problems are related to the fact that the TA algorithm *first* resamples the incomplete relations, and *then* computes the answer to the query. The alternative described in this Section is to do the opposite; *first* we compute the answer to the query, and only after we have decided which tuples are actually important do we see if they are included in each bootstrap repetition.

5.1 Where Do We Get the Counts From?

At a glance, this seems like a very simple modification to the TA algorithm. However, we face a significant hurdle when actually trying to implement the modification: *it is not obvious how to figure out how many times each tuple appears in a resampled relation, if we do not propagate this information throughout the query evaluation process.*

Recall from Section 4 that in order to bootstrap a relational database query, after running the underlying SQL query to completion, we need to scan the result set S . For each tuple $t \in S$, we must be able to determine how many times the tuples contributing to t appear in the resampled relations. For example, if the tuple ("Mr. Smith", "Joe", "Joe", 42) appears in S , and this tuple itself is a concatenation of the tuples ("Mr. Smith", "Joe") \in SUPERVISES and ("Joe", 42) \in AGE, we will need to know how many times ("Mr. Smith", "Joe") and ("Joe", 42) appear in the resampled versions of SUPERVISES and AGE, respectively. In the TA algorithm, accessing these counts is easy because this information has been stored within each tuple.

Unfortunately, if we do not propagate this information along with each tuple throughout the query evaluation process, we run into problems. Given a tuple $t_i \in R$, the straightforward way to determine how many times t_i appears in R^* (the resampled version

of R) is to pre-compute R^* all at once using Algorithm 3 from Section 4. If we choose not to propagate the resulting counts through the query evaluation process along with the database tuples, then the obvious alternative is to store the counts separately on disk. The counts can then be retrieved as S is processed to compute the final result of the bootstrap query.

The problem with this tactic is that since a tuple from S may be a concatenation of n arbitrary tuples from n different incomplete relations, we will need n random disk accesses in order to access the n sets of counts associated with the tuple. At 10ms per random disk access, we could process only $360,000/n$ tuples per disk per hour using this tactic. Clearly, this is an unacceptably slow processing rate, and we must search for a better algorithm.

5.2 Solution: On Demand Materialization

To ensure good performance, we must be far more careful how we figure out how many times a tuple appears in a resampled version of its relation. This can be done by eschewing Algorithm 3 in favor of a far more suitable method that does not require that we precompute and store a large amount of data on disk.

The new method we propose is based on a data structure called a *resample tree*. A *resample tree* over a relation R allows us to quickly look up the number of copies of a given tuple $t_i \in R$ that occur in a resampled version of R . A single resample tree simultaneously indexes all b resampled versions of R , and can operate effectively with limited main memory (even if there is not enough memory to hold a separate count for each tuple from the relations).

The tree functions by only materializing the part of the structure that is needed to answer a query over a single tuple, so it can operate effectively with a constrained amount of main memory. However, as the amount of available memory increases, more of the tree can be materialized at any given moment, and the speed with which a resample tree can be used to process queries also increases. The next two Sections describe the resample tree data structure in detail. Then Section 5.5 details exactly how the resample tree is used to implement the On-Demand Materialization (ODM) algorithm for bootstrapping a relational database query.

5.3 The Resample Tree Data Structure

The resample tree data structure will allow us to efficiently compute how many times a tuple from the relation R occurs in each of the b different resampled versions of relation R .

To provide an intuitive description of the resample tree, it is useful to first reconsider Algorithm 3, which provides a straightforward method for resampling a relation. Algorithm 3 can be thought of as simulating the following process: n “balls” are randomly tossed into n “slots”, where each slot corresponds to a tuple from the original relation R , and each ball corresponds to a tuple from the resampled relation R^* . If m balls fall in the i th slot, then the i th tuple from R appears m times in R^* .

The resample tree, however, uses a slightly different, multi-stage process to resample R . Assuming for simplicity that n is a power of two, the resample tree treats the process of resampling a relation as a series of $n(\log_2 n - 1)$ Bernoulli trials, rather than a single monolithic experiment. To resample a relation using a series of Bernoulli trials, we begin by randomly tossing each of the n balls into one of two bags, one labeled “left” and one labeled “right”. Next, we take all of the balls from the “left” bag, and again toss them into one of two new bags, also labeled “left” and “right.” We then do the same thing with all of the balls originally tossed into the bag labelled “right”, and keep on recursively repeating the process until each ball has been tossed into a bag exactly $\log_2 n$ times. Thus, at the end a ball residing in the i th leaf bag signifies the

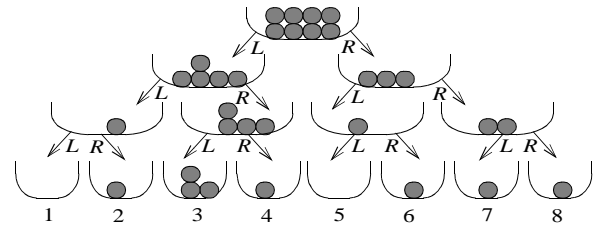


Figure 3: Resampling a relation using Bernoulli trials.

Algorithm 6: Constructing a new resample tree node

- (1) Seed the random number generator with $seed = key$
- (2) Initialize the *left* and *right* arrays of the new node.
- (3) For $i = 1$ to b , do:
- (4) Generate a sequence of $counts[i]$ random bits using the generator *Random*
- (5) For $j = 1$ to $counts[i]$ do
- (6) If the j th random bit is a 0, then $left[i]++$
- (7) If the j th random bit is a 1, then $right[i]++$

selection of the i th tuple in R . The final distribution of balls into bags can then be used to describe the number of times each tuple from R appears in R^* . Figure 3 depicts this process being used to compute how many times each of the 8 tuples from R appears in the resampled relation R^* . For example, the third tuple from R will appear three times in R^* , and the fifth tuple from R will not appear in R^* .

A *resample tree* is nothing more than a simple binary tree which is used to simulate this process. Since each relation must be resampled b times to run a bootstrap query to completion, each node in the tree stores count information for *all* of the b bootstrap repetitions (rather than a single repetition as is depicted in Figure 3). These counts are stored in two integer arrays, called *left* and *right*. Figure 4 shows a resample tree storing the process depicted in Figure 3, as well as five other bootstrap repetitions.

5.4 Resample Tree In Limited Main Memory

The tree depicted in Figure 4 is *complete*, in the sense that it depicts a tree that is small enough to be stored in main memory in its entirety. This is a somewhat uninteresting case, since the resample tree is designed to be used when there is not enough main memory to store the entire structure.

To handle cases where the tree is too large to be stored all at once in main memory, portions of a resample tree are continuously pruned in response to constraints on main memory. A primary consideration in the design of the tree is that the tree must be able to generate b pseudo-random resamplings of R that are “random”, and yet they must also be reproducible so that we can regenerate pruned portions of the tree. To accomplish these goals, the resample tree makes use of a pseudo-random number generator *Random*, such as the linear congruential algorithm (see Knuth [14], Section 3.2.1).

Since the resample tree is a recursive data structure, the creation of a node in the tree depends on the counts stored in the node’s parent. To create a new node in the tree, we need two pieces of information:

- The array *counts* which is composed of b integers passed down from the node’s parent. This tells the new node how many “balls” the new node has been passed from its parent in each of the b bootstrap repetitions.

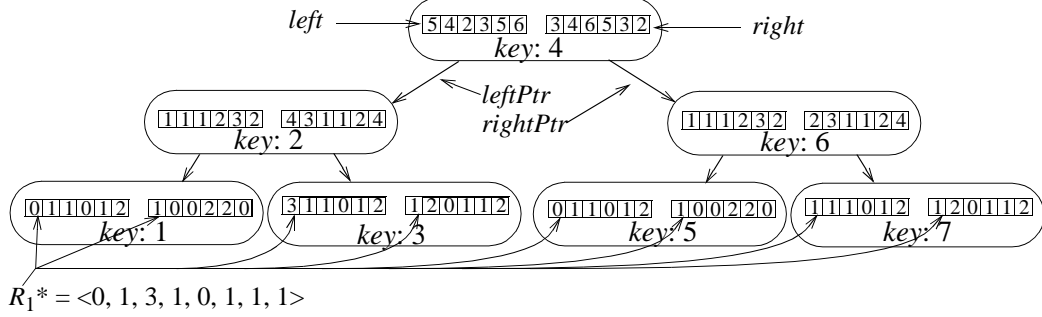


Figure 4: Resample tree indexing six bootstrap repetitions over a relation with 8 tuples. Each tree node encapsulates five data items: *left*, *right*, *leftPtr*, *rightPtr*, and *key*. The counts corresponding to the six resampled versions of R are indicated.

- An integer *key* which uniquely identifies the node in the tree.

Given this, Algorithm 6 is used to construct a new node in the tree. This algorithm simply takes the counts passed from the node’s parent and randomly distributes them to the left and right arrays stored in the node, so that the resample tree mimics the process depicted in Figure 3. The key characteristic of node generation is that this process is random, but it is also repeatable. If the node is subsequently deleted, then it can be regenerated exactly, as long as the same input array and key are passed as arguments.

From the bootstrap’s point of view, the most important operation on the tree is query evaluation. Given an integer i , we wish to access the array which tells us how many times the i th tuple in R appears in each of the b resampled versions of R . The recursive algorithm given as Algorithm 7 can be used to do just that. The algorithm is first invoked on the root node of the tree with key set to $|R|/2$ and the argument *toLeaf* set to $\log_2|R| - 1$. The *toLeaf* parameter tells the algorithm how far it needs to traverse to reach the leaf level of the tree. The query evaluation algorithm then traverses downwards through the tree until the appropriate node at the leaf level in the tree is found. If the algorithm must traverse a path with one or more missing nodes, these nodes are created on the fly and added to the tree.

A primary advantage of the resample tree and its associated algorithms is that the number of random bits that we must generate decreases exponentially with the available amount of main memory. Intuitively, the reason for this is that more information is stored in the upper levels of the tree, since the nodes closest to the root store partition information for the largest number of tuples. As we descend lower in the tree, it becomes easier and easier to repeat any lost computation. Since the number of nodes close to the root of a binary tree is exponentially larger than the number of nodes close to the leaves, we obtain a very large performance gain even if we can store only a small number of nodes.

The following lemma formally describes the efficiency of the resample tree:

Lemma 4: Assume that the m uppermost nodes in a resample tree are materialized. Then the expected number of pseudorandom bits that must be generated to answer a query is bounded

$$\text{by } 2b \left(2^{\left(\lceil \log_2|R| \rceil - \lfloor \log_2(m+1) \rfloor \right)} - 1 \right).$$

Proof Outline: To answer a query, at most $\lceil \log_2|R| \rceil$ nodes must be materialized. We know that at least the first $\lfloor \log_2(m+1) \rfloor$ nodes on the path to the appropriate leaf node will already be materialized. Summing the bits in the $\lceil \log_2|R| \rceil$

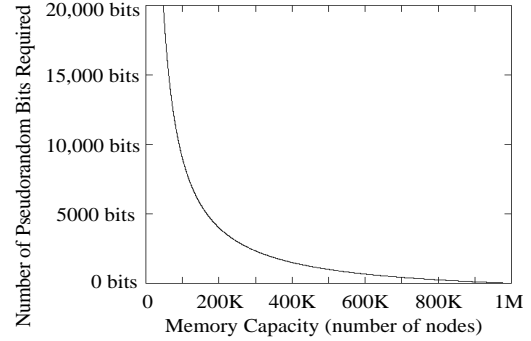


Figure 5: A plot of the number of bits required to answer a query using a resample tree. The size of the underlying relation is one million tuples. The number of bootstrap repetitions b is 1000.

- $\lfloor \log_2(m+1) \rfloor$ nodes that must be materialized to reach the leaf gives the result. ■

What exactly does this say about the speed of using a resample tree? For a concrete example, imagine that we have an incomplete relation with 1 million records, and we wish to generate 1000 resampled versions of the relation. Figure 5 plots approximately the number of random bits that must be generated to answer a query that will tell us how many times an arbitrary tuple from R appears in each of the 1000 resampled versions of R . For example, if we can afford to store 200,000 nodes in main memory, then we only need to generate 4,000 bits to answer each query, or around 125 pseudorandom, 32-bit numbers.

5.5 The Complete ODM Algorithm

Using a resample tree, computing the answer to a bootstrapped relational database query is a three-step process:

- (1) First, the underlying SQL query is run in the normal fashion to produce S , the set of tuples that the aggregate function f will be evaluated over. This is unchanged from the *TA* algorithm of Section 4.
- (2) Next, the tuples in S are sorted according to a pre-defined lexicographic ordering. In our implementation, we number each tuple from each incomplete relation R_i from 1 to $|R_i|$. If a tuple $t \in S$ is a concatenation of the individual tuples t_1, t_2, \dots, t_n , then we associate the multi-dimensional point $(t_1.\text{number}, t_2.\text{number}, \dots, t_n.\text{number})$ with t . S is then sorted using a Hilbert ordering over those multi-dimensional points.

Algorithm 7: Computing the number of times the i th tuple appears in each resampled version of R

- (1) If we are out of main memory, then delete the least-recently-visited node in the tree
- (2) If $toLeaf$ is 0, then:
 - (3) If $i \leq key$ return $left$;
 - (4) Else, return $right$
- (5) If $i \leq key$
- (6) If $leftPtr$ is null, then:
- (7) Use Algorithm 6 to construct a new node using $counts = left$ and $key = key - 2^{toLeaf-1}$
- (8) Set $leftPtr$ to be this new node
- (9) Recursively invoke Algorithm 7 on the node pointed to by $leftPtr$ using $toLeaf = toLeaf - 1$
- (10) Otherwise, if $i > key$
- (11) If $rightPtr$ is null, then:
- (12) Use Algorithm 6 to construct a new node using $counts = right$ and $key = key + 2^{toLeaf-1}$
- (13) Set $rightPtr$ to be this new node
- (14) Recursively invoke Algorithm 7 on the node pointed to by $rightPtr$ using $toLeaf = toLeaf - 1$

(3) In the last step, the tuples in S are loaded into memory, one at a time. For each tuple $t \in S$, the resample trees indexing the resampled versions of the incomplete relations R_1, R_2, \dots, R_x are queried to see how many times each tuple making up t appears in each of the b bootstrap repetitions of the underlying query. As described in Section 4, the number of times that t appears in S_i^* is simply the product of these counts. Finally, these resampled versions of S are used to compute the end result of the bootstrap query, again just as in Algorithm 3.

The reason that a Hilbert ordering is used to sort the tuples from S is that this allows us to traverse the data space with good spatial locality. This is helpful, because as discussed in the previous Section, each resample tree reacts to constraints on the amount of main memory by deleting parts of its structure. If deleted branches must subsequently be accessed, they need to be re-computed, which in turn can increase the amount of time required to execute the bootstrap query. Traversing the x -dimensional space defined by $R_1 \times R_2 \times \dots \times R_x$ so as to preserve spatial locality means that when a resample tree is queried, it is more likely that a relevant portion of the tree is currently materialized. In practice, this heuristic can reduce the number of random bits that must be generated far beyond the numbers prescribed by Lemma 4.

6 Benchmarking

6.1 Overview

In this section, we present a set of experiments aimed at benchmarking the performance of our bootstrapping algorithms for relational database queries. Using the C++ programming language and the PostgreSQL 7.4 DBMS, we have implemented and tested a suite of queries for following three scenarios:

- Query execution using traditional DBMS;
- Bootstrapped query execution with 1,000 repetitions using the TA algorithm; and

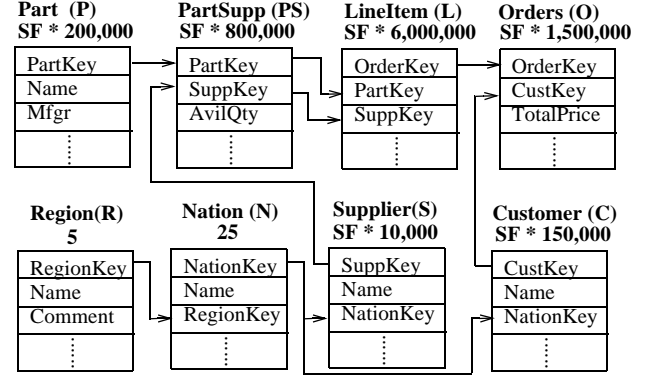


Figure 6: The TPC-H schema.

- Bootstrapped query execution with 1,000 repetitions using the ODM algorithm.

For the latter two cases we executed each query as follows:

- TA algorithm. As described in Algorithm 4A, we first pre-process each input relation to compute the number of times a tuple will appear in the relation for each bootstrap repetition. The counts are then compressed using Algorithm 5 and are appended to the respective tuples of the relation. This was phase one of TA algorithm. In phase two, we execute the query over the modified relations using the PostgreSQL DBMS. Finally, in phase three we post-process the query output to produce the final result. The post-processing was performed using Algorithm 4B. The execution times of all three phases were added to gauge the performance of the TA algorithm.
- ODM algorithm. Our implementation of the ODM algorithm begins by executing the underlying SQL query using PostgreSQL DBMS to produce the set of output tuples, S . S is then sorted in the lexicographic ordering of tuple numbers, as described in Section 5.5. In the last phase, S is post-processed to produce the bootstrapped query result. The counts for each tuple in S were retrieved from resample trees as described in Algorithm 7. As in the case of TA algorithm, performance is benchmarked by adding execution times of different phases.

6.2 Test Data

We tested all three scenarios on the data from the TPC-H benchmark [21]. TPC-H is an ad-hoc, decision support benchmark with eight base tables, as depicted in Figure 6. For more details on data types, table layouts, constraints, and implementation rules refer to the TPC website [21]. The fact tables in this database (tables L and O) are indexed on the composite primary key made up of the foreign keys of the dimension tables, while all other (dimension) tables are indexed on their primary keys. Other columns from various tables are also indexed to improve performance. There are 50 different indices used in TPC-H.

The $DBGEN$ program was used to produce the TPC-H data. $DBGEN$ takes as a parameter the scaling factor (SF) required by the benchmark. We have selected SF of ten to create a dataset of approximately of 11GB with the largest table (L) accounting for 7GB of data with 60 million rows. The record size used by the program varied from 104 to 179 bytes for different relations. The TA algorithm appends compressed tuple counts to the test data, making the records much larger. The ODM algorithm appends tuple numbers to the test data that are used to order the tuples during post-processing.

6.3 Test Queries

TPC-H benchmark consists of a suite of twenty-two business oriented ad-hoc queries. These queries are executed against a standard database under controlled conditions. TPC-H is designed to mimic a decision support environment that examines large volumes of data, executes queries with a high degree of complexity, and gives answers to critical business questions. We have selected the following five queries out of this suite for our benchmarking:

- *Pricing Summary Report Query (Q1)*. This query reports a summary for all lineitems shipped as of a given date. *Q1* lists eight different aggregate values over different attributes using aggregate functions: SUM, AVG, and COUNT. The date in the WHERE clause is chosen so that between 95% and 97% of the rows in the largest table (L) are eligible for the query.
- *Shipping Priority Query (Q3)*. This query retrieves the unshipped orders with highest value. *Q3* joins three largest tables of the database and has GROUP BY clause on a relational key attribute generating around 100,000 of different groups for *SF* of ten. All three attributes of the GROUP BY clause are indexed.
- *Local Supplier Volume Query (Q5)*. This query lists the revenue column done through local suppliers. In addition to a GROUP BY clause and series of WHERE clause predicates, this query has a equi-join over six tables of TPC-H schema, including both fact tables.
- *Forecasting Revenue Change Query (Q6)*. This query can be looked as a “what if” query asked to find ways to increase revenues. The query simply returns an aggregate value selected over largest fact table.
- *Discounted Revenue Query (Q19)*. This query reports the gross discounted revenue attributed to the sale of selected part handled in a particular manner. The query joins the L and P tables from the schema with a complicated WHERE clause.

The SQL statements for the selected queries were generated using *QGEN* program of TPC-H. The query results in each of three testing scenarios were validated against the TPC-H. During the experimental runs, the original queries are executed against the normal TPC-H database for scenario one. For *TA* and *ODM* algorithms, the modified queries (one without evaluation of aggregate value) were executed against the respective databases.

6.4 Results

The experiments were performed on a Linux workstation having an Intel Xeon Processor with 2.4 GHz clock speed and a 2GB of RAM. The machine was equipped with two 80GB 15,000 RPM Segate SCSI disks. Benchmarking of these disks showed a sustained read/write rate of 35-50 MB/Sec, and a worst-case seek time of 10ms. For traditional query execution and *ODM* algorithm, all the tables and their index structures were stored on a single disk, while for the *TA* algorithm the tables and index structures were stored on two separate disks (since they occupied more than 80GB). The following tables present the experimental results.

Table 1: Performance of the TA Algorithm

Q#	Original Query	TA Algorithm			
		Pre-Proc.	Query	Post-Proc.	Total
1	303m23s	332m55s	519m8s	239m6s	1091m9s

Table 1: Performance of the TA Algorithm

Q#	Original Query	TA Algorithm			
		Pre-Proc.	Query	Post-Proc.	Total
3	11m9s	417m31s	80m3s	1m26s	499m0s
5	180m22s	418m11s	347m42s	0m36s	766m29s
6	194m30s	332m55s	614m39s	1m51s	949m25s
19	76m27s	338m11s	236m32s	0m10s	574m53s

Table 2: Performance of the ODM Algorithm

Q#	Original Query	ODM Algorithm		
		Query	Post-Proc.	Total
1	303m23s	157m15s	209m15s	366m30s
3	11m9s	12m25s	43m23s	55m48s
5	180m22s	186m57s	39m59s	226m56s
6	194m30s	206m44s	11m15s	218m9s
19	76m27s	73m12s	14m2s	87m14s

6.5 Discussion

The results show decisively that both the *TA* and the *ODM* algorithms are far superior to the naive approach of simply running the query thousands of times over the resampled database. The additional time taken by our algorithms to re-run the query for 1,000 bootstrap repetition was orders of magnitude smaller than would have been required for the naive approach.

In the case of the *TA* algorithm, the cost of pre-processing relations was dominant, especially for queries involving joins over multiple large tables. As expected, phase two query execution under the *TA* algorithm was slower than what was observed for the non-bootstrapped version of the query, because of the compressed counts stored in the records. However, the post-processing was generally fast because of its simplicity (the exception is *Q1*, which returns a massive query result). In spite of slow phase one and two, the *TA* algorithm typically performed 8-9 times slower than the non-bootstrapped query, despite the fact that the query was effectively computed 1,000 times (the exception to this is *Q3*, which is discussed below). The time requirements would be even lower if the cost of phase one was amortized over execution of multiple queries. In general, it is necessary to re-run phase one only if the user wants to ensure that the resampled version of each relation is different for each execution.

While a query can apparently be bootstrapped 1,000 times with only an order-of-magnitude performance hit using the *TA* algorithm, the *ODM* algorithm showed less than a 25% performance hit for four of the five queries tested. Note that the time required to execute the underlying query by PostgreSQL during the *ODM* algorithm was typically the same or even less than the time required for traditional query execution. This is not surprising since the *ODM* algorithm runs a simplified version of the original query over the same database where evaluation of aggregate functions and GROUP BY clause is moved into the post-processing phase of the *ODM* algorithm. Post-processing (which makes use of the Hilbert ordering) generally finished surprisingly quickly, even for queries with joins over multiple relations.

Interestingly, the one query for which both algorithms showed a significant increase in execution time was Q_3 . The reason for this seems to be that the query involves an efficiently-indexed join over a large database table. Both the *TA* and the *ODM* algorithms must still expend significant computational resources to deal with the large size of this table, even though the query itself is efficiently handled by an index. It may be possible to address this problem in the case of the *ODM* algorithm by using a more efficient algorithm for generating binomially-distributed random numbers in the upper levels of the resample tree (see Related Work below).

7 Related Work

The bootstrap has a 25-year history in statistics. See Efron and Tibshirini for the definitive text [2]. Though to our knowledge the bootstrap has not been applied previously in a database setting, a long list of papers do discuss the inference of analytic confidence bounds for database queries [6][7][8][9][11][12][15].

Our work is mostly concerned with efficiently generating bootstrap samples in an out-of-core environment. The bootstrap requires computation of a large number of binomially distributed random numbers. The methods we rely on for this are simple “coin flips” and uniform random number generation. It may be possible to do better using more advanced methods: see Johnson, Kotz, and Kemp [13] for a description of some other methods for generating binomial random numbers. We point out that though these methods would not solve the problem addressed in this paper (namely, propagating the bootstrap sample through query evaluation), they could still conceivably be used to speed some aspects of our computation (specifically, generating the uppermost nodes in the resample tree).

There are a number of techniques from the statistics literature for speeding up the bootstrap. However, these methods do not concentrate on speeding the actual calculations; rather, they tend to focus on the orthogonal problem of reducing the number of bootstrap repetitions required for specific inference problems (see Efron and Tibshirini [2], Chapter 23).

Finally, we mention that there is a large body of existing work on dealing with uncertainty in database data, from probabilistic relational models [3] to fuzzy spatial databases [16] and everything in between. Again, our work is different in that we concentrate on efficiently incorporating a standard technique for generating confidence bounds into a RDB.

8 Conclusions and Future Work

Statistical estimation and approximate query processing are usually of little use without confidence bounds, and analytically deriving these bounds for database queries over sampled or incomplete data is a daunting task. In this paper we have considered the problem of incorporating into a database system a powerful method called the “bootstrap” that is suitable for computing confidence bounds for a large variety of database queries. The bootstrap is simple to use and widely applicable. The main contribution of this paper is the development of two bootstrapping algorithms that do not require modification of the DBMS query processing engine, and can re-run a query numerous times with a total execution time that is comparable to the evaluation of the query just once.

One obvious direction for the future work is handling sensitivity of our algorithms to bias and acceleration in the standard error of the estimator (Section 2.2). Another issue is that we have considered only simple select-project-join SQL queries with aggregation. This requires that the SQL GROUP BY operator be treated as part of the aggregate function to be bootstrapped, which is sub-optimal because it may require modification of the query compiler/proces-

sor in order for a DBMS to make use of our algorithms. Along the same lines, work remains to be done with respect to adapting the algorithms for nested SQL queries with negation. Another avenue for future work is to consider the problem of applying the bootstrap to sequential sampling schemes in a database environment (like online aggregation [8][11]). Since our algorithms assume a static set of sampled relations, they are not directly applicable to online aggregation where the sampled relations are constantly being updated with new samples.

References

- [1] M. Charikar, S. Chaudhuri, R. Motwani, V.R. Narasayya: Towards Estimation Error Guarantees for Distinct Values. *PODS* 2000: 268-279
- [2] B. Efron and R.J. Tibshirani: *An Introduction to the Bootstrap*. Chapman & Hall/CRC 1998
- [3] N. Friedman, L. Getoor, D. Koller, A. Pfeffer: Learning Probabilistic Relational Models. *IJCAI* 1999: 1300-1309
- [4] S. Acharya, P.B. Gibbons, V. Poosala, S. Ramaswamy: The Aqua Approximate Query Answering System. *SIGMOD* 1999: 574-576
- [5] P.J. Haas, J.F. Naughton, S. Seshadri, L. Stokes: Sampling-Based Estimation of the Number of Distinct Values of an Attribute. *VLDB* 1995: 311-322
- [6] P.J. Haas, J.F. Naughton, S. Seshadri, A.N. Swami: Selectivity and Cost Estimation for Joins Based on Random Sampling. *J. Comput. Syst. Sci.* 52(3): 550-569 (1996)
- [7] P.J. Haas: Large-Sample and Deterministic Confidence Intervals for Online Aggregation. *SSDBM* 1997: 51-63
- [8] P.J. Haas, J.M. Hellerstein: Ripple Joins for Online Aggregation. *SIGMOD* 1999: 287-298
- [9] P.J. Haas, J.F. Naughton, S. Seshadri, A.N. Swami: Fixed-Precision Estimation of Join Selectivity. *PODS* 1993: 190-201
- [10] P. Hall: *The Bootstrap and Edgeworth Expansion*. Springer-Verlag 1995
- [11] J.M. Hellerstein, P.J. Haas, H.J. Wang: Online Aggregation. *SIGMOD* 1997: 171-182
- [12] W.-C. Hou, G. Özsoyoglu: Statistical Estimators for Aggregate Relational Algebra Queries. *ACM Trans. Database Syst.* 16(4): 600-654 (1991)
- [13] N.L. Johnson, S. Kotz, A.W. Kemp: *Univariate Discrete Distributions*. John Wiley and Sons, Inc 1993.
- [14] D. Knuth: *The Art of Computer Programming, Volume 2*. Addison-Wesley Professional 1997.
- [15] R.J. Lipton, J.F. Naughton, D.A. Schneider: Practical Selectivity Estimation through Adaptive Sampling. *SIGMOD* 1990: 1-11
- [16] M. Schneider: Uncertainty Management for Spatial Data in Databases: Fuzzy Spatial Data Types. *SSD* 1999: 330-351
- [17] C. E. Shannon: A Mathematical Theory of Communication. *Bell System Technical Journal*, vol. 27: 379-423 and 623-656, (July and October, 1948)
- [18] D.F. Vysochanskii and Y.I. Petunin: Justification of the 3σ Rule for Unimodal Distributions. *Theory of Prob. and Math. Stat.*, vol 21: 25-36 (1980)
- [19] J. Ziv and A. Lempel: A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Inform. Theory*, 23: 337-343 (1977)
- [20] <http://www.sas.com>
- [21] <http://www.tpc.org/tpch>