Generating Targeted Queries for Database Testing

Chaitanya Mishra
University of Toronto
cmishra@cs.toronto.edu

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

Calisto Zuzarte
IBM Toronto
calisto@ca.ibm.com

ABSTRACT

Tools for generating test queries for databases do not explicitly take into account the actual data in the database. As a consequence, such tools cannot guarantee suitable coverage of test cases commonly required for database testing. In this paper, we investigate the problem of generating queries that satisfy cardinality constraints on intermediate subexpressions when executed on a given test database. Such queries are required to test the performance of a database system under different operating conditions

We formally analyze this problem, quantify its difficulty and follow up this analysis with a description of a practical algorithm which utilizes sampling and space pruning techniques to quickly generate test queries that have desired properties. We present the results of an experimental evaluation of our approach as implemented in an open source data manager, demonstrating the utility of our proposal.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Relational databases*

General Terms

Performance, Reliability

Keywords

Database Testing, Query Generation

1. INTRODUCTION

A necessary step in the introduction of any new technology in a database system is to test its behaviour across a wide range of operating conditions. This often involves selecting a set of test databases, generating representative query workloads, and executing these workloads on the test databases to evaluate the effect of the new technology. The importance of testing and benchmarking has long been recognized in the database community and there are several standard benchmarks developed for various settings [1]. While these standard benchmarks serve as useful reference points, there is often a need to generate test databases that satisfy certain properties on (for instance) table size, column domains, skew on columns and correlation between columns. To this end there have been several efforts [9, 17, 6] for generating large amounts of synthetic data which satisfy required properties. Correspondingly, there exist tools [16, 14] that can generate a large number of valid SQL queries to execute on a test database. However, these tools take only the database schema as input and generate the queries without looking at the underlying data. Therefore, they *cannot* guarantee generation of queries with certain kinds of properties. In particular, we are interested in generating queries that satisfy cardinality constraints on intermediate subexpressions.

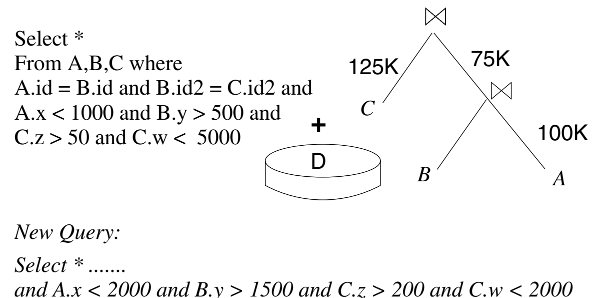


Figure 1: Sample Test Case

Consider, for instance, an improvement to a join algorithm being introduced in a database system. A natural step in evaluating the algorithm would be to test its performance across varying sizes of inner and outer inputs. Given a fixed underlying test database, the input sizes can be controlled by varying the selection predicates on the base relations. However, in the absence of additional information, database testers currently have no means other than a cumbersome trial and error procedure to find a choice of selection predicates that result in the query satisfying the input size targets. To make the problem even more challenging, one might want to test how the join algorithm operates as a component in a pipelined query processing architecture. In this case, one would like to generate queries with varying intermediate join result sizes. This can be particularly difficult to do in the presence of skew and and inter-attribute correlations.

Figure 1 describes a sample test case in this setting. We are given a query Q, a set of target cardinality constraints on intermediate subexpressions in the query evaluation plan, and a test database D on which the query is to be executed. Our goal is to modify Q to generate a new query Q^r that satisfies the target cardinality constraints when executed on database D. The class of modifications we consider in this paper are modifications to the range selection

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, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06 ...$5.00.

predicates of Q. We refer to this problem as the *Targeted Query Generation (TQG)* problem.

In this paper, we study the TQG problem, and provide a formal analysis as well as practical solutions. We extend previous hardness results for the problem established for the special case of a single cardinality constraint [7], and prove lowerbounds on approximating the problem in this case. Our analysis demonstrates the difficulty of the problem, and illustrates the error guarantees one can expect. Further, we analyze the general case of multiple target cardinality constraints and show how it is affected by statistical properties of the underlying database. In particular, we demonstrate that several cases of interest can be reduced to the problem of finding the best fit solution to a system of linear equations.

We follow up the analysis with a description of a practical algorithm that can be used to quickly generate queries that approximately satisfy the target cardinality constraints on subexpressions. Our algorithm progressively refines the range selection predicates using a novel search procedure that utilizes sampling based techniques for fast and accurate cardinality estimation. We have prototyped our techniques inside the Postgresql database system, and demonstrate the utility of our solution through an extensive experimental evaluation.

The rest of the paper is organized as follows. In Section 2 we describe related work. We then present a formal analysis of the targeted query generation problem in Section 3. We follow this with a description of our new algorithm in Section 4. We present a detailed experimental evaluation of our technique in Section 5 and conclude in Section 6.

2. RELATED WORK

Bruno et al. [7] first investigated the TQG problem. Their work primarily considered the special case of TQG with only one cardinality constraint. They formally proved that this restricted version of the TQG problem is NP hard. They also introduced a heuristic hill climbing approach that assumes independence between the selectivities of the different predicates. In our present work, we establish formal guarantees on the hardness of approximating the problem, and describe a practical algorithm that avoids any assumptions about the statistical properties of the data.

The QAGen system [4] introduces a complementary approach towards the targeted testing problem. Instead of generating a test query given the test database, the approach generates a test database given the test query. To do so, QAGen introduces symbolic query processing which necessitates the use of constraint solvers to generate the underlying database. The primary drawback of the approach is that it generates a different database instance *for each test case*. As a result, the storage overheads of applying this approach for large scale testing of a new feature may be unacceptable. In addition, QAGen suffers from the overheads of using an expensive constraint solver which make it unacceptably slow for large databases. For instance, the QAGen paper reports a database generation of approximately 20 hours for a test case based on TPC-H query 3 and a 1 GB database size. More than 80% of this time is spent in the constraint solver. Our experimental evaluation shows that our techniques are several orders of magnitude faster. One advantage of QAGen is that it attempts to satisfy the test case exactly, while we attempt to approximately satisfy the test case. However, their exact approach comes with significant overheads, and we contend that in many cases, generating a query that approximately satisfies the test case should suffice.

The primary focus of research in database testing has been on the test database generation problem. Gray et al. [9] introduced techniques for generating large amounts of data having specific data distributions. More recently, the MUDD data generator [14] for TPC-DS [15] and the DGL language[6] introduced techniques for separating data distribution specification from the actual data generation. Our work enables targeted query generation *without* modifying the underlying database, and thus enables reuse of the standard test databases used at an organization.

The problem of query generation has received comparatively less attention, and typically, the query generation procedure is decoupled from the database generation process. Tools like RAGS [16] and QGen [14] enable large scale generation of valid SQL queries. Since these tools utilize only the schema and are independent of the underlying data, they cannot be used to generate test queries which guarantee specific properties during query execution. Our solution fundamentally differs from these techniques since it is data-aware. In particular, our solution accepts as input a fully specified SQL query, and therefore can utilize the output of such a query generation tool. More recently, Bati et al. [3] introduce a framework for generating test queries by modifying current test queries using execution feedback. Although similar in its basic principle, the focus of the paper is primarily on ensuring coverage of executor and optimizer code in the database system.

In addition to previous research on database testing, our work draws upon techniques for sampling based cardinality estimation of joins in relational databases [11, 10]. We extend these schemes using novel bounding arguments to avoid the cost of sampling multiple times from the disk.

3. ANALYSIS

3.1 Problem Statement

We are given a database D and a conjunctive query Q defined on relations $R_1, \ldots, R_t$. The query has d range predicates $P_1, \ldots, P_d$. Each range predicate takes the form of $R_j.x_i < C_i$ or $R_j.x_i > C_i$. We treat two-sided range predicates as two separate single sided predicates. In what follows, we assume without loss of generality that predicate P_i is of the form $x_i < C_i$. Predicates of the form $x_i > C_i$ can be transformed into $-x_i < -C_i$. We use C_i^l and C_i^u to denote the lower and upper bounds of the domain on which P_i is defined. Thus the constant C_i in predicate P_i is bounded as $C_i^l \leq C_i \leq C_i^u$, and the size of the domain of P_i is $C_i^u - C_i^l + 1$. We refer to the domain of predicate P_i as *dimension* i. The query Q thus has d dimensions.

Range predicates can be modified only by changing the constant in the expression. Thus, for example *age* < 50 can be modified to *age* < 70 or *age* < 30, but not to *age* > 20. Queries can be modified by altering the range predicates. We refer to this process of modifying queries as *query refinement*.

A *test case* τ is defined in terms of a query Q, a database D, and a set of m cardinality constraints $\mathcal{N}$. Each cardinality constraint is defined in terms of a subexpression Q_i of Q, and a target cardinality N_i for Q_i. We represent cardinality constraints as pairs (Q_i, N_i). Likewise we represent test cases as triples $(Q, D, \mathcal{N})$.

Given, these definitions, we now state the TQG problem

Definition 1. **Targeted Query Generation:** Given a test case $\tau : (Q, D, \mathcal{N})$, generate a test query Q^r by refining Q such that Q^r, when executed on database D satisfies the cardinality constraints specified by $\mathcal{N}$.

We use *m* to denote the number of cardinality constraints in the test case. We let *d* be the total number of predicates that can be refined to modify the cardinality of intermediate results i.e the number of dimensions of Q. We use *n* to represent the size of the largest

domain on which a range predicate is defined. Our cost analysis is in terms of these three parameters.

EXAMPLE 1. *Consider the test case specified by Figure 1. We represent the cardinality constraints as* $(A, 100K)$, $(A \bowtie B, 75K)$, $(C, 125K)$. *Therefore,* $m = 3$. *There are 4 range predicates* ($A.x < 1000$, $B.y > 500$, $C.z > 50$ *and* $C.w < 5000$). *Therefore, the number of dimensions* $d = 4$. *We set* n *to be the size of the largest domain among* $A.x$, $B.y$, $C.z$ *and* $C.w$.

An *evaluation layer* is defined as a module that given a query Q returns a cardinality estimate for Q. It can take the form of actual query execution, optimizer cardinality estimates, or any other cardinality estimation scheme. Our algorithms invoke the evaluation layer through the function `QueryEval` which returns an estimate of the cardinality of the given query. Our techniques are independent of the evaluation layer in use. In the following analysis, we assume that the evaluation layer returns *exact* cardinality estimates i.e the error in the estimate is 0. Following [7], we measure the complexity of our algorithms in terms of the number of calls to the evaluation layer. Subsequently, in Section 4.3, we introduce an efficient sampling based evaluation layer for the practical realization of our algorithms. We now present a formal analysis of the hardness of the TQG problem. We first consider the special case when the number of cardinality constraints m is 1. We then consider the general case of $m > 1$ cardinality constraints.

3.2 Single Cardinality Constraint

Consider the TQG problem given only a single cardinality constraint on the result size of the query Q. Since there is only one constraint, we drop the subscript, and denote it as (Q, N) where N is the target result cardinality. For this special case, Bruno et al. [7] proved the following Lemma:

Lemma 1. A lower bound on the number of calls to the evaluation layer to generate a query Q satisfying a single constraint (Q, N) with two parametric predicates $x_1 \leq C_1$ and $x_2 \leq C_2$ is $\Omega(n_{min})$ where n_{min} is the minimum number of distinct values in x_1 and x_2. For $d > 2$ parametric predicates, there is a lowerbound of $\binom{n+d-2}{d-1}$ calls to the evaluation layer.

Note however, that this lowerbound would typically be too expensive since we would expect that $n >> d$, and therefore, even $O(n)$ calls to the evaluation layer might be considered unacceptable. We consider the hardness of determining an *approximate* solution to the TQG problem given a single constraint (Q, N). We use the *absolute error* as our metric in this setting. If we generate a query Q^r which returns N^r tuples, then the absolute error of Q^r is $|N^r - N|$. We state lowerbounds for any algorithm that given a test case with a single constraint (Q, N) as input, guarantees returning a query Q^r with absolute error less than a constant E. If such a query does not exist, the algorithm returns that no query is found. Our lowerbounds also carry over to any algorithm that guarantees absolute error less than ϵN where ϵ is a constant and $\epsilon < 1$.

Lemma 2. Suppose there is an algorithm *Alg* that given a single constraint TQG problem (Q, N) guarantees a solution Q^r with absolute error less than E or ϵN where E and ϵ are constants and $\epsilon < 1$ (if such a solution exists). Then, the lowerbound on the number of calls to the evaluation layer made by *Alg* matches the lowerbound given by Lemma 1 for the exact solution.

We provide the proof of Lemma 2 in the Appendix. Lemma 2 states that any algorithm that provides absolute error guarantees in

Algorithm 1 Single Constraint Procedure

```
1:  Database D
2:  SingleConstraint(Query Q, Target N)
3:  Q^r = Q
4:  E = Error(QueryEval(Q^r,D),N)          Check Error of Q
5:  for all Dimensions i do
6:      P = Predicates(Q)
7:      P_i = NULL                         Disable i^th predicate
8:      Q_P = Query(P)                     Generate new query
9:      Q' = Find(Q_P,N)
10:
11:     E' = Error(QueryEval(Q',D),N)      Check Error of Q'
12:     if E' < E then
13:         Q^r = Q'
14:         E = E'
15:     end if                   If Error is reduced, replace query
16: end for
17: return Q^r

18: Find(Query Q, Target N)
19: i = NullDimension(Q)           Disabled Dimension
20: min = C_i^l                    Set min, and max
21: max = C_i^u                    to domain boundaries
22: val = (min+max)/2
23: while (min <= max) do
24:     P_i = x_i < val            Set the predicate accordingly
25:     Est = QueryEval(Q,D);      Call Evaluation layer for Est.
26:     if Est < N then
27:         min = val
28:     else if (Est > N) then
29:         max = val
30:     else
31:         return Q;
32:     end if
33: end while                      End of loop for binary search
34: return Q
```

the form of $|N^r - N| < E$ or $|N^r - N| < \epsilon N$ has a cost lowerbound which matches the cost lowerbound for the exact solution to the problem. We now present procedure *SingleConstraint* that provides weaker *data-dependent* error guarantees while requiring just $O(d \log n)$ calls to the evaluation layer.

Algorithm 1 describes procedure *SingleConstraint*. The procedure accepts a query Q, and modifies its range predicates to generate a new query Q^r that approximately satisfies the target cardinality constraint N. This is accomplished by iterating over predicates of Q and refining them. For each predicate $P_i : x_i < C_i$ of Q, *SingleConstraint* invokes a function *Find* that performs a binary search between the lower C_i^l and upper C_i^u bounds of the domain of predicate P_i, returning a query that minimizes the absolute error. For each value of the predicate considered by the binary search, *Find* invokes the evaluation layer through function *QueryEval* to get a cardinality estimate for the given query.

EXAMPLE 2. *Suppose our target cardinality is 3477 tuples and our query is on a single table R with 10000 tuples. The query has just one predicate* $R.x < C$ *which can take values of* C *between 1 and 1001, and the data is uniformly distributed on column* $R.x$. *Given this predicate, procedure* `Find` *will do a binary search between these bounds taking values for* `val` *as (in order): 501, 251, 376, ..., 349. This results in the predicate* $R.x < 349$ *which returns 3480 tuples.*

Let M_i be the maximum frequency on dimension i in the result of Q. Let e be the minimum value of M_i over all dimensions i i.e $e = min_i(M_i)$. We now state bounds on the error and performance of *SingleConstraint*

Lemma 3. *SingleConstraint* guarantees an error of $e/2$ at a cost of $O(d \log n)$ calls to the evaluation layer.

PROOF. *SingleConstraint* calls the procedure *Find* once for each dimension. Each call to *Find* makes $O(\log n)$ calls to the evaluation layer, giving us the overall cost of $O(d \log n)$. For the error guarantee, notice that we can add a straightforward condition to the binary search so that it stops at a value such that increasing or decreasing the value by 1 will increase the error. Therefore, a binary search of dimension i can result in an error of at most $M_i/2$. Since we iterate over all dimensions, the global error is at most $min_i(M_i/2) = e/2$ $\square$

The lowerbound on approximation given by Lemma 2 demonstrates that we can always come up with adversarial instances that make the targeted query generation problem hard to approximate. On the other hand, procedure *SingleConstraint* shows that a simple binary search procedure can easily guarantee an error bound of $e/2$ at a cost of $O(d \log n)$. We argue that on reasonable test databases, this would serve as a tight enough error bound. Note that this is a worst case error bound, and in practice the error is expected to be lower.

3.3 Multiple Constraints

The above discussion establishes that approximation guarantees for the TQG problem with a single constraint. We now analyze the case when we have multiple constraints $\mathcal{N} : (Q_1, N_1), \ldots, (Q_m, N_m)$.

The TQG problem becomes significantly more challenging in the presence of multiple constraints. It is easy to observe that in this case, it is possible for the constraints to be *inconsistent*. As a simple example, consider a query $R \bowtie S$ with some selection predicates on R and no predicates on S and suppose every tuple of R joins with 1 tuple of S. Let the test case be specified as $(R, N_R), (RS, N_{RS})$. If $N_R \neq N_{RS}$, one can easily see that the constraints are inconsistent. In general, while there might be several solutions for each of the cardinality constraints, it isn't necessary that these sets of solutions intersect providing a solution that satisfies all m constraints.

We now show how the hardness of the TQG problem with multiple constraints is affected by statistical assumptions on the data. We first introduce some notation which we require for this analysis. Let f_i be the cumulative distribution function on dimension i corresponding to predicate $P_i : x_i < C_i$. Therefore, $f_i(C_i^l) = 0$ and $f_i(C_i^u) = 1$. Let $V(Q_j)$ denote the set of dimensions included in subexpression Q_j. We denote the joint cumulative distribution function over $V(Q_j)$ as f_{Q_j}. Let N_j^m be the cardinality of subexpression Q_j when for every $i \in V(Q_j)$, we set $P_i : x_i < C_i^u$. Since each predicate is set to the upperbound of its domain, N_j^m is the maximum cardinality of subexpression Q_j.

The TQG problem can then be considered to be the problem of identifying C_1,C_2,...,C_d such that:

$$\forall_{1 \leq j \leq m} f_{Q_j}(C_{i:i \in V(Q_j)}) = \frac{N_j}{N_j^m}$$

Each predicate P_i can be set to $x_i < C_i$ to obtain the resulting test query.

3.3.1 Independence assumptions

To begin with, we assume independence between the columns of the table. We make these assumptions only to highlight a special case of the TQG problem which is amenable to analysis. Our general solution for the problem is described in Section 4.

With inter-column independence, we have, for each subexpression Q_j:

$$f_{Q_j} = \prod_{i:i \in V(Q_j)} f_i$$

The TQG problem then becomes

$$\forall_{1 \leq j \leq m} \prod_{i:i \in V(Q_j)} f_i(C_i) = \frac{N_j}{N_j^m}$$

Let $y_i = -\log f_i(C_i)$. Similarly, let $z_j = \log \frac{N_j^m}{N_j}$. We denote the y_is using a d dimensional vector $\mathbf{y}$, and the z_js with a m dimensional vector $\mathbf{z}$. Similarly, we can represent the m sets $V(Q_j)$ as a $m \times d$ matrix $\mathbf{V}$. Then the TQG problem with inter-column independence becomes:

$$\forall_{1 \leq j \leq m} \sum_{i \in V(Q_j)} y_i = z_j$$

$$\text{or } \mathbf{V}\mathbf{y} = \mathbf{z}$$

We need to find a solution to this system which satisfies the additional constraints that $\forall_i y_i \geq 0$.

EXAMPLE 3. *Consider the test case defined in Figure 1 and assume that the columns are independent. Suppose that the maximum cardinality of A is 200K, of $A \bowtie B$ is 100K and of C is $200K$. We can represent the TQG problem as the following system of equations:*

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} y_{A.x} \\ y_{B.y} \\ y_{C.z} \\ y_{C.w} \end{pmatrix} = \begin{pmatrix} \log 200K/100K \\ \log 100K/75K \\ \log 200K/125K \end{pmatrix}$$

Suppose we solve the system of equations and obtain values of y_i for every dimension i. This in turn provides a value of $f_i(C_i)$, which corresponds to the selectivity of predicate P_i. However, we are actually interested in the value of C_i. Let R be the relation on which predicate P_i is defined. Then, the selectivity $s = f_i(C_i)$ implies a target cardinality of $s|R|$ for the query $\sigma_{P_i} R$. Since there is just one constraint, we can apply the procedure *SingleConstraint* on this resulting query and solve for C_i. The resulting predicate $P_i : x_i < C_i$ has the property that the query $\sigma_{P_i} R$ returns approximately $s|R|$ tuples. The approximation guarantee is defined as per Lemma 3.

Observe that in our test cases, each cardinality constraint is specified as an intermediate subexpression in a query evaluation plan. Therefore, cardinality constraints are defined on subsets of the relations on which the query executes. As a result, we *do not* require independence between the selection columns on the same table. Therefore, instead of having separate cdfs f_i for each column x_i, we simply define a joint distribution function f_R over the set of dimensions of table R. The system of equations that are formed as a result can be appropriately solved, following which we use procedure *SingleConstraint* to obtain appropriate selection predicates for R. We do however require that the join column of R be independent of the selection columns on R, which in turn implies inter-table independence.

3.3.2 Numerical Solutions to equations

Given the system of equations $\mathbf{V}\mathbf{y} = \mathbf{z}$ we seek solutions for $\mathbf{y}$ subject to the constraint $\mathbf{y} \geq 0$. The system of linear equations given by $\mathbf{V}\mathbf{y} = \mathbf{z}; \mathbf{y} \geq 0$ can be underdetermined, have a unique solution, or be overdetermined. Each of these cases is possible in the TQG problem as we illustrate in the following example:

EXAMPLE 4. *Consider a join of R and S with selection predicates on R.x and S.y. With just one cardinality constraint on $R \bowtie S$, the corresponding system of equations has one equation*

and two variables, and is therefore underdetermined. With cardinality constraints on both $R \bowtie S$ and R, we have two equations, and two variables, giving us a unique solution. Finally, if we have cardinality constraints on $R \bowtie S$, R and S, we have three equations on the same two variables, leading to an overdetermined system of equations.

If the system of equations is consistent (i.e has at least one solution), we can solve for it using standard techniques to obtain a solution for y_i. If there are multiple solutions, any of these solutions would work equally well. All that we require are values for y_i that can be used to solve for C_i to obtain the new predicates P_i.

If however, the system of equations is inconsistent, there is no solution $\mathbf{y}$ that satisfies all the equations in the system. In this case, we seek solutions that optimize some norm between $\mathbf{Vy}$ and $\mathbf{z}$. Depending on desired semantics any of the candidate norms, such as L_1, L_2 and L_∞ can be of interest. Consider the case of the L_2 norm. In this case the problem takes the following form:

$$\begin{aligned} &\min ||\mathbf{Vy} - \mathbf{z}||^2 \\ &\text{s.t } \mathbf{y} \geq 0 \end{aligned}$$

This problem is amenable to a closed form numerical solution. Notice that the L_2 norm we are trying to minimize subject to linear constraints is a convex function. We apply primal-dual techniques to derive a solution. The dual of the problem can be derived as follows:

$$\min_{\mathbf{y}} L(\mathbf{y}, \lambda) = (\mathbf{Vy} - \mathbf{z})^T(\mathbf{Vy} - \mathbf{z}) - \lambda^{\mathbf{T}}\mathbf{y}$$

Setting the derivative of $L(\mathbf{y}, \lambda)$ w.r.t $\mathbf{y}$ as zero we obtain the optimal values of $\mathbf{y}$ minimizing $L(\mathbf{y}, \lambda)$ as a function of λ. In particular we obtain:

$$\mathbf{y}^* = \frac{1}{2}(\mathbf{V^TV})^{-1}(2\mathbf{z}^T\mathbf{V} + \lambda^T) \quad (1)$$

Substituting this choice of $\mathbf{y}^*$ and taking the derivative w.r.t λ, we obtain the optimal value of λ. This can be substituted back in the original equation to obtain the $\mathbf{y}$ vector as a solution to the L_2 minimization problem. A similar analysis can be conducted for the case of other norms such as L_1 and L_∞. However in these cases one can show that no closed form numerical solution is available and one has to resort to numerical procedures to obtain the optimal vector $\mathbf{y}$ [2].

We note that minimizing the L_2 norm is a particularly appropriate choice for the problem under consideration. Minimizing $||\mathbf{Vy} - \mathbf{z}||^2$ is equivalent to minimizing the following error function

$$E = \sum_{1 \leq j \leq m} (\log \frac{N_j}{N_j^r})^2 \quad (2)$$

where N_j^r is the cardinality of the resulting query Q^r at the intermediate subexpression Q_j. Essentially, minimizing the L_2 norm is equivalent to minimizing the *sum squared logarithmic relative error* over the set of cardinality constraints in the test case. This metric is does not have any bias towards subexpressions Q_j with larger target cardinalities N_j, and equally penalizes overshooting and undershooting a constraint. We utilize this error metric in the rest of this paper for the multiple constraint case.

3.3.3 General Case

Our analysis in the previous sections shows that the TQG problem with multiple constraints is solvable if our test database guarantees the property that the join column is independent of the selection columns, and consequently that the selection columns across tables are independent. If we drop these assumptions as well, the TQG problem becomes less amenable to analysis since the data distribution of a selection column can be affected by the distribution on the join column.

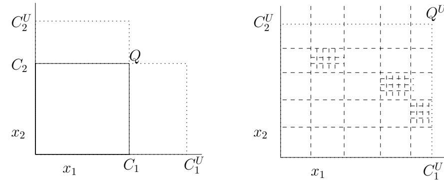


Figure 2: TQGen Algorithm

Advances in data generation techniques have ensured that generating test databases with inter-table statistical dependencies is an easy task. As a consequence we still require a technique to generate targeted test queries which can handle such databases. In the next section, we describe our solution for the general case of this problem. The adversarial argument of Lemma 2 shows that one cannot expect to obtain an efficient algorithm that guarantees returning a query with an arbitrarily small bound on the error. Instead, we present a best-effort algorithm for the TQG problem. In Section 5, we present an experimental evaluation demonstrating the effectiveness of our algorithm.

4. THE TQGEN ALGORITHM

We now present the *Targeted Query Generation (TQGen)* algorithm for generating queries that satisfy cardinality constraints on intermediate subexpressions. TQGen takes as input a test case $\tau = (Q, D, \mathcal{N})$ and generates a new query Q^r that (approximately) satisfies the constraints $\mathcal{N}$ when executed on database D. We note that the TQGen procedure does not make any independence assumptions between attributes. As demonstrated in the previous section however, the targeted query generation problem is computationally difficult. Therefore TQGen is a best-effort search procedure that utilizes heuristics to guide its search for queries satisfying the test case.

The space of all possible refinements of the original query Q can be described as a d dimensional space, with each dimension corresponding to a predicate of Q. Each point in this space corresponds to a setting of the d predicates, and therefore describes a unique query. For instance, Figure 2 describes a 2-dimensional space for a query with predicates on x_1 and x_2. In the figure, the point C_1, C_2 describes the original query in this space. TQGen explores this space to search for test queries that approximately satisfy the test case τ. It does so by *bounding* the search space, and then *exploring* this restricted space.

In the *bounding* phase, which we describe in Section 4.1, TQGen restricts the search space for solutions. This is done by generating a new query Q^U that is a superset of the original query Q. Q^U is generated by refining each predicate of $P_i : x_i < C_i$ of Q to $x_i < C_i^U$ where $C_i \leq C_i^U$. We define the semantics of our space bounding operation further in Section 4.1.

In the *exploration* phase, described in Section 4.2 TQGen explores the space $\forall_i x_i \leq C_i^U$ for valid test queries Q^r. An exhaustive search of this space would however be extremely inefficient. Therefore, we partition the d dimensional space of predicates using a grid as illustrated in Figure 2. The grid is generated by performing an *equi-width* partitioning along each dimension. Each cell in the grid is a candidate for further exploration through repartitioning at a finer granularity. We define a *cell scoring* function to evaluate the utility of exploring a given cell, and select the b-highest scoring

Algorithm 2 Space Bounding

```
1:  Bound(Q)
2:  for all Dimensions i do
3:      C_i^L = C_i
4:      C_i^U = C_i
5:  end for            Initially set bounds to the original values
6:  for all Constraints (Q_j, N_j) do
7:      ComputeBounds(Q_j,N_j,lower)
8:  end for   First compute lowerbounds iterating over constraints
9:  for all Dimensions i do
10:     P_i : x_i < C_i^L.
11: end for            Modify predicates generating Q^L
12: for all Constraints (Q_j, N_j) do
13:     ComputeBounds(Q_j,N_j,upper)
14: end for            Repeat for upperbounds

15: ComputeBounds(Q_j,N_j,which)
16: for all Dimensions i ∈ V(Q_j) do
17:     Temp = P_i
18:     P_i = NULL
19:     Q' = Find(Q_j,N_j)       Calling Find to obtain new value
20:     C'_i = Value(i ∈ V(Q'))
21:     P_i = Temp
22:     if which == lower then
23:         if C'_i < C_i^L then
24:             C_i^L = C'_i
25:         end if           Replace lowerbound if smaller
26:     else
27:         if C'_i > C_i^U then
28:             C_i^U = C'_i
29:         end if           Replace upperbound if larger
30:     end if
31: end for
```

cells for further exploration. The algorithm is a recursive procedure that repeats this process of partitioning and scoring at finer levels of granularity for the selected cells. This recursive partitioning is illustrated in Figure 2 in which the 3 cells have been chosen for further repartitioning.

At all times during query execution, the TQGen algorithm maintains the lowest error query seen till then. We utilize as our error function the sum squared logarithmic relative error function defined in Equation 2. However, our algorithm is error function independent, and can utilize any monotonic error function i.e if the error for any constraint (Q_j, N_j) increases, then the total error cannot decrease.

TQGen uses an *evaluation layer* module for estimating the cardinality of the potential test queries considered. Our algorithm is independent of the evaluation layer used. In Section 4.3, we describe a sampling based evaluation layer designed specifically for use with the TQGen approach. We now present the *bounding* and *exploration* phases of the TQGen algorithm.

4.1 Space Bounding

The first step of the TQGen algorithm consists of identifying reasonable bounds on the search space to be explored for test queries. Each predicate $P_i : x_i < C_i$ can take any value for C_i between C_i^l and C_i^u. Our goal is to find a value C_i^U such that $C_i^l < C_i^U \leq C_i^u$, and C_i^U can serve as an upper bound on dimension i for our exploration procedure.

An upper bound C_i^U for a dimension i is *strict* if no test query Q^r that satisfies the test case can have a predicate $P_i^r : x_i < C_i^r$ with $C_i^r > C_i^U$. It is important to note here that seeking *strict* upper bounds may not lead to any reduction in the space. Consider the following example:

EXAMPLE 5. *Suppose there is a target cardinality constraint of 4000 tuples on a single relation R which contains 10000 tuples.We have two range predicates $x < 50$ and $y < 50$. The domain of both the predicates is $[1, 101]$, the data is uniformly distributed on both columns x,y and they are independent of each other. In this case, the predicates $x < 41 \wedge y < 101$ and $y < 41 \wedge x < 101$ would both serve as solutions to the test case, and therefore the strict upperbound for x and y is 101 which implies no reduction in the space.*

Example 5 illustrates why strict semantics may not lead to any reduction in the search space for the exploration phase. Therefore, we adopt a weaker definition of an upperbound. A query Q^U is a *valid upperbound* if Q^U *overshoots* all the cardinality constraints specified by $\mathcal{N}$ when executed on the database D. A query Q^U overshoots a constraint (Q_i, N_i) if executing Q^U on D returns $N_i^U \geq N_i$ tuples at the subexpression Q_i.

Notice that in the process of refining a query Q to a test query Q^r, we can transform predicates through *contraction* (e.g $age < 50$ refined to $age < 40$) or *relaxation* (e.g $cost < 100$ refined to $cost < 150$). Contracting some predicates may require us to relax other predicates. For instance, in Example 5, contracting the predicate $y < 50$ to $y < 41$ results in the relaxation of the predicate on x to $x < 101$.

We would like our upperbound computation procedure to take into account the effects of predicate contraction. As a result, we adopt a two-step approach to this problem. Given Q, we first compute a new query Q^L with predicates $x_i < C_i^L$ that is guaranteed to *undershoot* all the constraints in $\mathcal{N}$. Each predicate $x_i < C_i^L$ satisfies the following property with respect to Q:

Property 1. If we replace *any* predicate $x_i < C_i$ of Q with $x_i < C_i^L$, and keep all other predicates as before, the resulting query undershoots all cardinality constraints (Q_j, N_j) s.t. $i \in V(Q_j)$

Essentially, C_i^L acts as a lowerbound for dimension i in the sense that if we *only* modify predicate P_i to $x_i < C_i^L$ will lead to the resulting query undershooting all constraints affected by P_i. We then generate Q^U with predicates $x_i < C_i^U$ by refining the query Q^L. The predicates $x_i < C_i^U$ of Q^U satisfy the following property with respect to Q:

Property 2. If we replace any predicate $x_i < C_i$ of Q with $x_i < C_i^U$, and replace all other predicates $x_{i'} < C_{i'}$ with $x_{i'} < C_{i'}^L$, the resulting query either overshoots all cardinality constraints (Q_j, N_j) s.t. $i \in V(Q_j)$, or $C_i^U = C_i^u$.

The resulting query Q^U serves as a reasonable upperbound since it explicitly takes into account the effects of *contracting* some of the predicates.

Procedure *Bound* presented in Algorithm 2 describes how we compute Q^U. *Bound* utilizes the *Find* procedure described in Algorithm 1. Recall that the *Find* procedure accepts a query Q with d predicates, a cardinality constraint (Q, N) and a single predicate P_i specified as NULL. Given this information, *Find* computes the value of predicate P_i that minimizes the absolute error for the specified cardinality constraint (Q_j, N_j).

For each constraint (Q_j, N_j) and each predicate $P_i \in V(Q_j)$, our procedure calls *Find* to compute the best value of predicate P_i which minimizes the error *only for* (Q_j, N_j). By taking the minimum such value of P_i across all constraints (Q_j, N_j), we compute a lowerbound for the predicate P_i. This process is shown through lines 6-8 and 22-24 in Algorithm 2. Having then generated Q^L (lines 9-11) by replacing C_i with C_i^L, we repeat this process for upperbounds (lines 12-14), and take the maximum value of predicate P_i returned by *Find* as an upperbound for dimension i.

Algorithm 3 Exploration of the space

```
1: Explore(Cell T^u, T^l)
2: If isEmpty() return
3: Partition T^u, T^l into upto k^d smaller cells.
4: List = NULL
5: for all T_i^u, T_i^l ∈ Partition do
6:    Compute Error E_i for Q(T_i^u) by calling QueryEval
7:    if E_i < E_best then
8:       Set Q^r = Q(T_i^u)
9:    end if          Check if it has lowest error
10:   Score The Cell defined by T_i^u, T_i^l
11:   if List has space OR Score(T_i^u, T_i^l) > Score(List) then
12:      Add T_i^u, T_i^l to list. Trim if necessary
13:      Update Score(List)
14:   end if          Add to list if it is in the top-b.
15: end for
16: for all (T_i^u, T_i^l) ∈ List do
17:    Explore(T_i^u, T_i^l)
18: end for
```

4.2 Exploration

Given the query Q^U, we utilize procedure *Explore* described as Algorithm 3 to search for potential test queries Q^r that minimize the error function E given by Equation 2. *Explore* takes as input a cell, which is defined by setting upper and lower bounds on each dimension. Each cell is thus specified by two d dimension vectors $(\mathbf{T^u}, \mathbf{T^l})$. The initial cell is thus specified by the vectors $(C_1^U, \ldots, C_d^U)$ and $(C_1^l, \ldots, C_d^l)$ Given a d dimensional vector $\mathbf{T}$, we use $Q(\mathbf{T})$ to denote query Q with each predicate P_i set to the corresponding value in $\mathbf{T}$. For instance, $Q(3, 5, 7)$ would represent the query with predicates set as $x_1 < 3$, $x_2 < 5$, and $x_3 < 7$. Similarly given a constraint (Q_j, N_j), $N_j(\mathbf{T})$ denotes the cardinality of subexpression Q_j on execution of $Q(\mathbf{T})$.

Procedure *Explore* partitions each of the d dimensions of a cell into at most k segments. This partitioning is performed in an *equi-width* manner. This results in the given cell being decomposed into upto k^d smaller cells. Since each dimension i is partitioned into at most k segments, there are at most $k + 1$ boundary values being considered for predicate P_i at each step. The $k+1$ boundaries along the d dimensions together define $(k+1)^d$ intersection points in the d dimensional space. Each such point corresponds to a potential test query. We *evaluate* each such query using our error measure given by Equation 2 (Alg. 3 line 5). This evaluation is done using m calls to the evaluation layer, once for each subexpression (Q_j) with constraint (Q_j, N_j). At all times during execution *Explore* maintains information about the query with minimum error seen until then.

In addition to evaluating the queries defined by the cell boundaries, *Explore* also needs to select b cells to further repartition and explore at a finer granularity (Alg. 3 lines 9-13). To do so, it computes *scores* for each cell in the grid, and selects the b highest scoring cells for further exploration. The scoring function quantifies the utility of further exploring a cell by repartitioning it. In addition, it utilizes *pruning* techniques to rule out exploration of non-promising cells. We next describe both these techniques in more detail.

4.2.1 Scoring Cells

At each step of the *Explore* procedure, we consider upto k^d cells for further exploration. However, to avoid an exhaustive search, we utilize a scoring function to score the potential benefit of exploring the given cell. Our scoring function utilizes the following two parameters.

Number of constraints bounded: For each cell $(\mathbf{T_i^u}, \mathbf{T_i^l})$, we count the number of cardinality constraints (Q_j, N_j) that are bounded by the cell boundaries. We denote this set of bounded constraints by $B(i)$. Therefore, we have $j \in B(i)$ if $N_j(\mathbf{T_i^u}) \geq N_j \geq N_j(\mathbf{T_i^l})$. Thus $|B(i)|$ is an upper bound on the number of constraints that can be satisfied by any query defined by predicates within the cell $\mathbf{T_i^u}$ and $\mathbf{T_i^l}$. Given two cells, $(\mathbf{T_1^u}, \mathbf{T_1^l})$ and $(\mathbf{T_2^u}, \mathbf{T_2^l})$, we assign a higher score to $(\mathbf{T_1^u}, \mathbf{T_1^l})$ if $|B(1)| > |B(2)|$.

Uniformity across constraints: Suppose two cells bound the same number of constraints, and we need to drop one of them. In this case, we can define the distance of the cardinality constraint (Q_j, N_j) from the cell boundaries $(\mathbf{T_i^u}, \mathbf{T_i^l})$ as:

$$D_j(i) = \frac{N_j(\mathbf{T_i^u}) - N_j}{N_j(\mathbf{T_i^u}) - N_j(\mathbf{T_i^l})}$$

Let $\overline{D(i)}$ be the average distance $D_j(i)$ for all $j \in B(i)$. We then compute the standard deviation of these distances across the set of bounded constraints:

$$\sigma^2(i) = \frac{1}{|B(i)|} \sum_{j \in B(i)} (D_j(i) - \overline{D(i)})$$

We assign higher scores to cells with lower values of $\sigma^2(i)$. The reasoning behind this choice is that we desire test queries that approximately satisfy *all* the constraints in the test case. Therefore, we prefer cells that are equidistant from the target cardinality constraints.

Together, these are the two components of our cell scoring function. We use the first component (number of constraints bounded) for the purposes of scoring, and use the second component to break ties.

4.2.2 Pruning

The scoring function described in the previous section enables our procedure to evaluate the utility of further exploring a cell. We now describe two pruning techniques that we use to reduce the costs of our search procedure.

Lowerbound-Upperbound based pruning: If the lowerbound $\mathbf{T_i^l}$ of a cell $(\mathbf{T_i^u}, \mathbf{T_i^l})$ undershoots exactly m of the constraints, then the cell cannot bound more than m constraints i.e $|B(i)| < m$. Consequently, if the last element in the current `list` of cells bounds more than m constraints, we can prune $(\mathbf{T_i^u}, \mathbf{T_i^l})$ *without* evaluating the upperbound $\mathbf{T_i^u}$ of the cell.

Error based pruning: While processing a cell $(\mathbf{T_i^u}, \mathbf{T_i^l})$, suppose that for a constraint (Q_j, N_j), we have $N_j(\mathbf{T_i^l}) > N_j$ i.e the lowerbound of the cell overshoots the constraint. In this case, for constraint (Q_j, N_j), no query in the cell $(\mathbf{T_i^u}, \mathbf{T_i^l})$ can have an error lower than $(\log N_j(\mathbf{T_i^l})/N_j)^2$. Likewise, suppose $N_j(\mathbf{T_i^u}) < N_j$ i.e the upperbound of the cell undershoots the constraint. In this case, no query in the cell can have an error lower than $(\log N_j(\frac{\mathbf{T_i^u}}{N_j}))^2$ for constraint (Q_j, N_j)

As we process the constraints, for each constraint (Q_j, N_j) that is not bounded by the cell i.e $j \notin B(i)$, we add the error lowerbound described above to a running sum S_{Err}. If at any point in this processing, this lowerbound sum S_{Err} exceeds the current best error E_{best}, we can stop processing the cell and prune it from consideration.

4.2.3 Summary

To summarize, the TQGen procedure takes a d dimensional cell, partitions it into k^d smaller cells, and chooses upto b of them for further exploration. We select cells for further exploration using the *scoring* and *pruning* functions which evaluate the utility of a cell. Among all the queries evaluated, TQGen returns the one minimizing the error function given by Equation 2.

We now analyze the cost of our algorithm. Each invocation of the *Explore* procedure involves the evaluation of upto $O(k^d)$ potential test queries. Each query evaluation involves m invocations of the evaluation layer, once for each subexpression Q_j on which a constraint is defined. Therefore the cost per call to *Explore* is $O(mk^d)$. Now suppose the procedure branches into b child cells upto a depth l. Beyond l, it selects only one further child at each level i.e $b = 1$ after a depth l. The total depth of the tree is $\log_k n \geq l$. The total number of invocations performed is therefore $1 + b + \ldots + b^{l-1} + b^l(\log_k n - l) = O(b^l \log_k n)$. Therefore, the total cost of the procedure is $O(b^l mk^d \log_k n)$, which has the essential property of being logarithmic in the size of the domain n.

4.3 Evaluation Layer

In section 3, we defined an evaluation layer as a structure which returns cardinality estimates for queries submitted to it. Our procedures are independent of the actual evaluation layer used. If we desire perfect cardinality estimates, we could actually execute the queries on the database as suggested in [7]. Alternatively, we could make use of optimizer cardinality estimates, histograms [12], sample views [13], statistics on intermediate tables [5], and sampling from base tables [11, 10] to obtain cardinality estimates for our algorithm. In our analysis upto this point, we have defined the cost of our algorithms in terms of the number of calls to the evaluation layer. However, for our algorithms to be practical, we need a fast and accurate evaluation layer.

We now present an evaluation layer that is specifically tailored to interact with the TQGen procedure. We first provide background on the sampling based cardinality estimation scheme utilized by our layer. We then present our techniques for generating an evaluation layer for the *bounding* and *exploration* phases of our algorithms.

4.3.1 Sampling Scheme

We utilize a sampling based evaluation layer to provide cardinality estimates to the TQGen algorithm. Our cardinality estimation procedure is based on the idea of join cardinality estimation taking a random sample from one relation, and joining it with indexes on the other relations. In sampling literature [10] this is referred to as the *t_index* sampling scheme. If a random sample of the outer relation of size n tuples joins with the inner indices to produce N_{join} tuples, we estimate the cardinality of the join as $N_{join} \times N_{outer}/n$ where N_{outer} is the size of the outer relation. In terms of selectivity, Haas et al [10]. show that, under certain reasonable assumptions, if μ is the actual selectivity of the join, and μ_n is the estimated selectivity after n tuples have been read then:

$$P\{|\mu_n - \mu| \leq \epsilon\mu\} \approx 2\phi(\frac{\epsilon\mu\sqrt{n}}{\sigma}) - 1 \quad (3)$$

when n is large and $\epsilon\sqrt{n}$ is small. σ^2 is the variance and ϕ is the c.d.f for a standardized normal random variable.

In our estimation procedures, we perform the join of a random sample of one relation with indices on the other relations. However, unlike the t_index sampling scheme, we do not simply maintain a count of the number of tuples seen. Instead, we maintain the result of this join procedure in an in-memory data structure. We utilize this in-memory result in our cardinality estimation procedures. The join process is halted once the size of the result in memory crosses a specified threshold or if we have exhausted the random sample on disk. We refer to this process of executing a join using a random sample of one relation as a *sampling step*. A naive sampling based cardinality estimation scheme would incur a cost of one sampling step for each call to the evaluation layer. In the following sections, we show how we utilize our space bounding techniques to minimize the number of sampling steps required for cardinality estimation.

4.3.2 Evaluation Layer for Bounding

In the *bounding* phase of the TQGen algorithm, we compute lowerbounds C_i^L and upperbounds C_i^U for each predicate P_i in the original query. Recall that the *Bound* procedure presented as Algorithm 2 computes C_i^L and C_i^U by keeping all predicates other than P_i fixed, and performing a binary search on the domain of P_i. This procedure makes $m \log n$ calls to the evaluation layer to compute an upperbound or lowerbound of a predicate. Hence, the overall cost of the procedure is $2md \log n$ invocations of the evaluation layer.

To avoid $2md \log n$ sampling steps, we utilize the fact that the result of the query, with predicate P_i disabled is a *superset* of the result if we set P_i to any value in its domain. This is illustrated in the following example:

EXAMPLE 6. *Consider a query Q $R \bowtie S$ with predicates $R.x < C_x$ and $S.y < C_y$. Note that the query $Q \subseteq \sigma_{S.y<C_y}(R \bowtie S)$ for any value of C_x. Similarly, $Q \subseteq \sigma_{R.x<C_x}(R \bowtie S)$ for any value C_y.*

Let $Q(\bar{P}_i)$ denote the query Q with predicate P_i disabled. Therefore, $\forall_i Q \subseteq Q(\bar{P}_i)$. To compute the lowerbound C_i^L for predicate P_i, we perform one sampling step for each subexpression Q_j of $Q(\bar{P}_i)$ for which $i \in V(Q_j)$. We store the result of this sampling procedure in memory, and *Find* procedure using this in-memory result as an evaluation layer. Effectively, we reduce the number of sampling steps by a factor of $\log n$ since we sample *only once* for each cardinality constraint (Q_j, N_j) and predicate P_i pair. Our technique differs from traditional sampling based cardinality estimation techniques in that we apply the predicate *after* the join procedure.

We further reduce our sampling costs by a factor of d by combining the separate sampling steps for each predicate into one. This is done by generating a query that returns tuples that satisfy *at least* $d - 1$ of the d predicates. Consider the query given in Example 6. The new query:

$$\sigma_{R.x<C_x \vee S.y<C_y}(R \bowtie S)$$

returns the union of the tuples returned by the two queries $\sigma_{R.x<C_x}(R \bowtie S)$ and $\sigma_{S.y<C_y}(R \bowtie S)$. Therefore, we combine $|V(Q_j)|$ sampling steps for each constraint (Q_j, N_j) into one sampling step of a query that returns tuples that satisfy $|V(Q_j)| - 1$ of the predicates. To enable this additional optimization, we modify the join pipeline to return tuples that satisfy at least $|V(Q_j)| - 1$ of the $|V(Q_j)|$ predicates in the subexpression Q_j. We adopt additional optimization measures to limit the size of the sample result stored in memory. These measures include maintaining frequency counts instead of the actual data, and pruning away sample elements that are far away from our current estimate of the lowerbound.

We repeat the process described above for computing upperbounds C_i^U as well. The result of these two optimizations is that we sample *only twice* for each cardinality constraint (Q_j, N_j), once for lowerbounds, and once for upperbounds. As a result, we reduce the total number of sampling steps from $O(2md \log n)$ to just $O(2m)$ steps.

An additional optimization that is possible is to combine the sample processing for constraints that can be arranged in a pipeline. Consider the query $R \bowtie S \bowtie T$ with constraints on $R \bowtie S$ and $R \bowtie S \bowtie T$. Instead of sampling separately for each, we can combine the sampling steps together into a join pipeline $(R \bowtie S) \bowtie T$, where we read in tuples from a sample of R as the outer and join

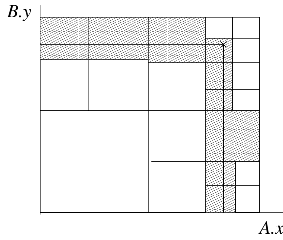


Figure 3: Probing a QuadTree, The shaded regions are where the actual points are inspected.

them with indices on S and T. However, we do not adopt this optimization in our realization of the TQGen algorithm since it would involve significant changes in the architecture of our system.

4.3.3 Evaluation Layer for Exploration

In the *exploration* phase of the TQGen algorithm, we explore a space defined by a query Q^U i.e $\forall_i x_i < C_i^U$. Given a potential test query $Q(\mathbf{T})$, defined by a vector $\mathbf{T}$, we utilize the evaluation layer to obtain cardinality estimates for each subexpression Q_j on which a cardinality constraint (Q_j, N_j) is defined. Observe that any such potential test query $Q(\mathbf{T})$ returns a subset of the tuples returned by Q^U. Therefore, we can perform one sampling step for each subexpression Q_j of Q^U, and store the result in memory. This result can be utilized to estimate the cardinality $N_j(\mathbf{T})$ of any potential test query $Q(\mathbf{T})$ at subexpression Q_j. As a result, we perform only m sampling steps, one for each cardinality constraint (Q_j, N_j).

Given query $Q(\mathbf{T})$, we wish to estimate $N_j(\mathbf{T})$ for each constraint (Q_j, N_j). We do so by counting the number of tuples in the in-memory join sample of Q_j which are *dominated* by $\mathbf{T}$ (i.e are smaller along all dimensions) and then scaling up. To make this step faster, we store the computed result of the join sampling step in a d-dimensional Quadtree data structure. We therefore have m quadtrees associated with each of the m constraints in the query. We use an adaptive QuadTree partitioning scheme in which a node is partitioned if it contains more than a δ fraction of the tuples in the tree. As in the case of the grid partitioning scheme, we can represent each node of the quadtree as a pair of d-dimensional vectors $(\mathbf{T^u}, \mathbf{T^l})$ defining the upper and lower bounds along each dimension. With each node in the tree, we maintain a count of the number of tuples in the descendants of the node

To estimate $N_j(\mathbf{T})$, we recursively explore the nodes of the Quadtree associated with Q_j. Consider a node $(\mathbf{T^u}, \mathbf{T^l})$ of the quadtree. If $\mathbf{T} > \mathbf{T^u}$, then every tuple in the node $(\mathbf{T^u}, \mathbf{T^l})$ contributes to the cardinality $N_j(\mathbf{T})$. In this case, we add the number of tuples in the descendants of the node to the target cardinality estimate with appropriate scaling, and do not recurse further down the tree. On the other hand, if there is a dimension i such that $\mathbf{T^l}[i] > \mathbf{T}[i]$ (i.e $\mathbf{T^l}$ dominates $\mathbf{T}$ on i), then no tuple in $(\mathbf{T^u}, \mathbf{T^l})$ contributes to the join result. In this case as well, we do not recurse further into the children of the node. Using these two checks ensures that our Quadtree probing procedure avoids examining the actual tuples at the leaf nodes as far as possible. Consider for instance Figure 3, which describes a probe to our Quadtree structure. We only inspect the actual data in the shaded leaf nodes. The rest of the nodes are either completely dominated by our query point $\mathbf{T}$, or dominate $\mathbf{T}$ along at least one dimension.

5. EVALUATION

In this section, we describe our system for targeted query generation, and present the results of an experimental evaluation of our techniques.

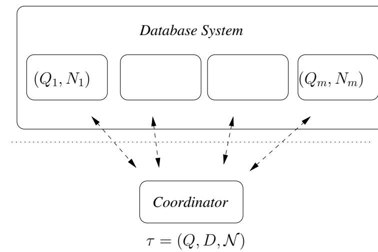


Figure 4: System Architecture

Table Name	Symbol	# Tuples
Lineitem	L	6000003
Orders	O	1500000
Customer	C	150000
Part	P	200000
Supplier	S	10000
PartSupp	PS	800000

Table 1: Table Sizes

5.1 System Architecture

We have implemented a system for targeted query generation that utilizes the TQGen algorithm. The system consists of an external program, which we term the *Coordinator*, and a modified version of the Postgresql 8.0 database system. This architecture is illustrated in Figure 4. The coordinator accepts a test case specified as $(Q, D, \mathcal{N})$. For each cardinality constraint $(Q_j, N_j) \in \mathcal{N}$, it submits a query Q_j to the database system. We have modified the execution engine of the Postgresql database system to provide the functionality of an evaluation layer for both the *Bounding* and *Exploration* phases of the TQGen algorithm as described in Section 4.3. Therefore, the query execution process for each subexpression Q_j acts as an evaluation layer for that subexpression. The coordinator communicates with these m processes through network sockets. Each evaluation layer process is unaware of the other processes, and only communicates with the coordinator. This design ensures that most of the complexity of our algorithm is kept outside the database engine, while leveraging the query processing primitives inside the engine.

5.2 Experiments

We now present the results of an experimental evaluation of our techniques. Our experiments were conducted using two TPC-H databases of size 1 GB each. One of them is the standard TPC-H database generated as per the benchmark specification. Since this database consists of uniformly distributed data, we also generated a TPC-H database with zipfian skew $Z = 1$ using a publicly available tool [8]. The sizes of the tables and the symbols representing the tables used in our experimental evaluation are provided in Table 1. In our graphs, we refer to the database with skew as *TPCH Z=1*, and the uniform database as *TPCH Z=O*.

We first present our experiments evaluating the accuracy of our test query generation techniques. In Section 5.2.1, we evaluate our solution for the TQG problem with a single cardinality constraint, as described in Section 3.2. We follow this with an experimental validation of our analytical model for the TQG problem with multiple constraints on independent columns, as presented in Section 3.3. Finally, we evaluate the TQGen algorithm using workloads with varying numbers of joins and cardinality constraints in

Section 5.2.3. We complement the accuracy evaluation, with experiments presenting the execution costs of our approach in Section 5.2.4.

All our experiments utilize a sampling based evaluation layer as introduced in Section 4.3. For the *bounding* phase of the evaluation layer, described in Section 4.3.2, we stop the sampling step when we have 1000 tuples for each predicate. For the quadtree based evaluation layer for exploration as presented in Section 4.3.3, we halt the sampling step when either the quadtree has 5000 elements or a 5% random sample of the outer relation has been read. The cardinality estimation errors due to our evaluation layer were typically less than 1% and did not affect the quality of the queries generated by our system.

5.2.1 Single Constraint Case

Figures 5(a) and 5(d) present our accuracy experiments for the TQG problem with a single constraint. For the experiment in Figure 5(a), we defined test cases on a selection query on table *PartSupp* (PS) with range predicates on attributes *availqty* and *supplycost*. The test cases were defined with target result cardinality ranging from 80K to 720K tuples i.e target selectivity from 0.1 to 0.9. We plot the relative error of our generated test queries, defined as $\frac{N^r}{N}$ where N is the target cardinality, and N^r is the cardinality of the test query generated by our system. Figure 5(d) presents a similar experiment with cardinality constraints defined on the result size of a three table query $L \bowtie O \bowtie C$ with range predicates on *C.acctbal*, *O.totalprice*, *L.extendedprice* and *L.quantity*. As with the previous experiment, the target selectivity was varied from 0.1 to 0.9. Both the experiments were conducted on *TPCH* databases with $Z = 0$ and $Z = 1$. As the figure illustrates, the relative error of our technique is consistently within 1 ± 0.02. The only point outside this range is in Figure 5(a) for the $Z = 1$ database with selectivity 0.1. This point has a relative error of 1.03, and an absolute error of 2539 tuples from the target cardinality. Recall that the error guarantee of *SingleConstraint* is $e/2$ where e is the minimum of the maximum frequencies along the dimensions of the query. In this particular example, $e/2$ is approximately 8000 tuples, which is much greater than the error at the data point

5.2.2 Multiple Constraints: Analytical Model

We next validate our analytical model for the TQG problem with multiple constraints and independence between predicates as described in Section 3.3. We define test cases on three table query $PS \bowtie P \bowtie S$, with range predicates on *PS.availqty*, *PS.supplycost*, *P.retailprice*, and *S.acctbal*. This query was selected since the TPC-H specification does not define correlations between these three tables. We define the test cases with cardinality constraints on PS, $PS \bowtie P$ and $PS \bowtie P \bowtie S$. The solution to the corresponding system of equations implied a selectivity of s each for PS, P and S, where s varied from 0.1 to 0.9. We computed values for the predicates on each of these relations utilizing procedure *SingleConstraint* (Alg 1).

Figures 5(b) and 5(e) show the results of evaluating our generated test queries for the TPC-H databases with $Z = 0$ and $Z = 1$ plotted against the selectivity s. The labels `Upper`, `Middle` and `Lower` denote the subexpressions $PS \bowtie P \bowtie S$, $PS \bowtie P$ and PS, by virtue of their positions in the join pipeline. As can be seen, the generated queries validate our analytical model, and the relative error decreases as the target cardinality increases. We note that the higher relative errors at selectivity 0.1 for the `Upper` graph in both figures are caused due to the fact that the target cardinality at this node is just 800 tuples (due to a combined selectivity of $0.1 \times 0.1 \times 0.1$) with respect to the outer relation size. Another interesting observation is that the relative error for the `Upper` graph in Figure 5(b) increases when we increase the target selectivity 0.1 to 0.2. This is an effect of the fact that our generated test queries *undershoot* the constraints at the `Middle` and `Lower` nodes for selectivity 0.1, and *overshoot* these constraints for selectivity 0.2. In the first case, the errors partially cancel each other, while in the second case, they add up. Finally, we note that the errors for $Z = 0$ are typically lower than those for $Z = 1$. This is because columns with more skew have higher maximum frequencies, and the error guarantee of our *SingleConstraint* procedure is defined in terms of these maximum frequencies.

5.2.3 Multiple Constraints: TQGen

In the previous two sections, we have evaluated our solution for the special case of TQG with single constraints, and validated our analytical model for multiple constraints with independent predicates. In this section, we evaluate our general purpose TQGen algorithm. Recall that the TQGen exploration phase constructs a grid by partitioning each dimension of a cell into k equi-width partitions. It then selects b cells for further repartitioning. In the following experiments, we set these parameters as $k = 3$, and $b = 2$ upto a depth $l = 2$ in the recursion. Beyond depth l, we set $b = 1$.

We first present our results for the TQGen algorithm given the query $P \bowtie PS$ with two range predicates on each of the relations. We fix the target cardinality of PS to 500K tuples and vary the target selectivity of the join expression $P \bowtie PS$ between 0.1 and 0.9. Figures 5(c) and (f) present our results for these test cases for TPC-H databases with $Z = 0$ and $Z = 1$. The queries generated by our technique have relative error bounded by 1 ± 0.04 for the `Upper` and 1 ± 0.02 for the `Lower` constraint. This experiment illustrates that the TQGen algorithm can generate accurate test queries *without* utilizing the information that the selection columns of P and PS are independent.

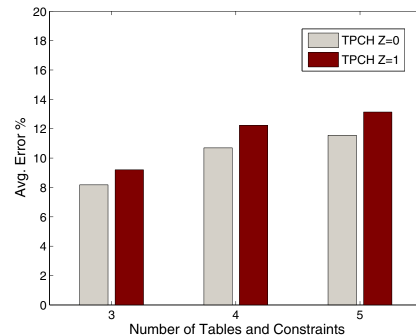
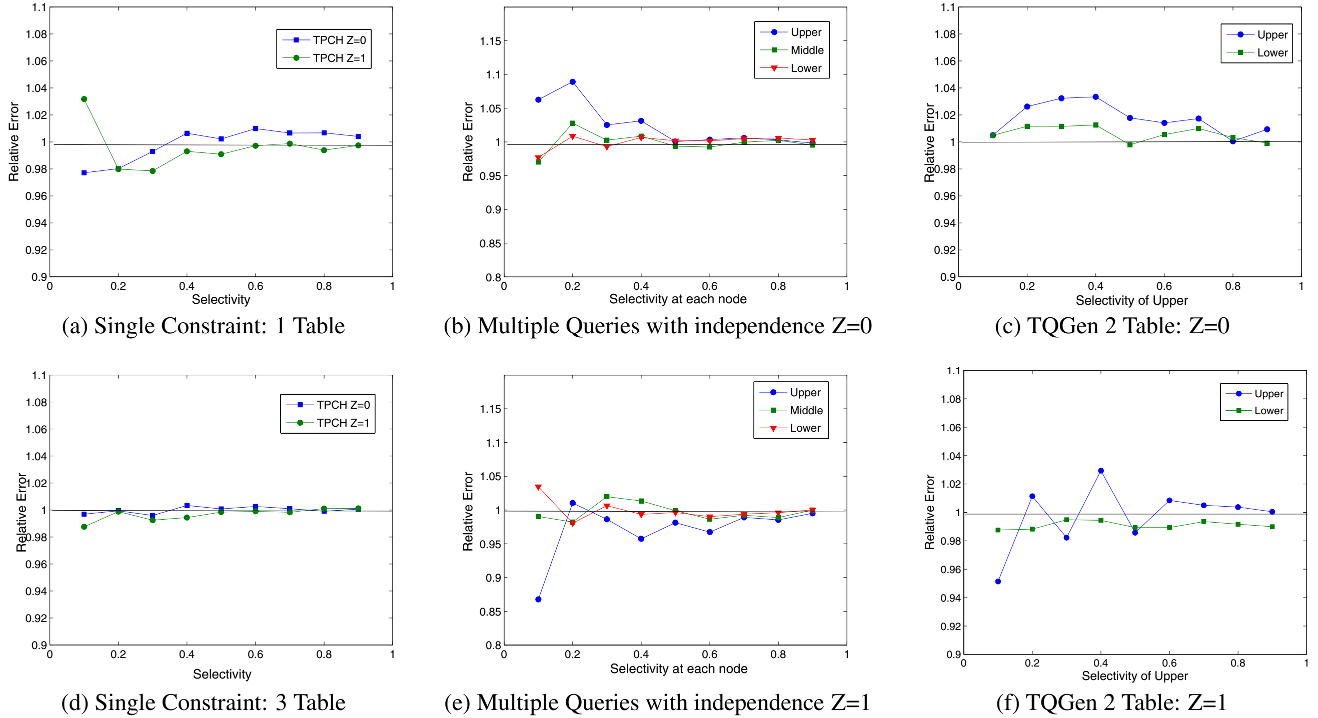


Figure 6: TQGen algorithm: Varying # of tables

We then evaluate the accuracy of the TQGen algorithm for queries with varying numbers of tables and cardinality constraints. Each instance presented in Figure 6, describes a join query over n tables with n cardinality constraints, where n varies from 3 to 5. The constraints are defined at the outer relation, and at the intermediate nodes in a left deep join tree. For each such instance, we generate 10 test cases, keeping the query fixed, and varying the target cardinality constraints. We execute TQGen for each of these test cases, and calculate the relative error incurred by the generated test query at each constraint. We compute the average relative error over the constraints in a test case, and report the average of these errors over all 10 test cases in Figure 6. The results show that increasing the number of tables and constraints in the test case leads to an increase



(a) Single Constraint: 1 Table (b) Multiple Queries with independence Z=0 (c) TQGen 2 Table: Z=0

(d) Single Constraint: 3 Table (e) Multiple Queries with independence Z=1 (f) TQGen 2 Table: Z=1

Figure 5: Accuracy Experiments

in the average error. This is due to the multi-objective nature of the TQG problem. However, we note that in all cases, the errors remain low and yield meaningful solutions to the TQG problem.

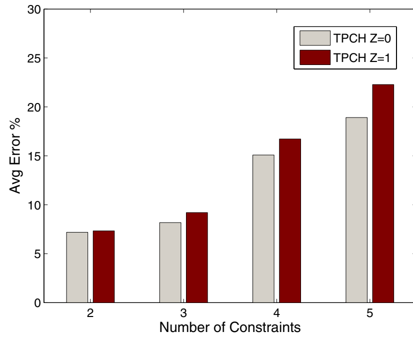
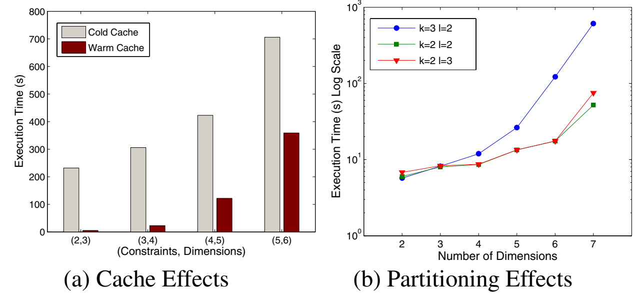


Figure 7: TQGen algorithm: Varying # of constraints

We also evaluate our techniques by fixing the number of tables in our test case, and varying the number of constraints. Figure 6 shows our results for a query on $P \bowtie PS \bowtie S$. We vary the number of subexpressions with cardinality constraints from 2 ($PS, P \bowtie PS \bowtie S$) to 5. As expected, the average error increases with the number of constraints. More interestingly, we note that the test cases with 4 or 5 constraints are *overdetermined*. For instance, the test cases with 5 constraints, have target cardinalities on P, S, PS, $PS \bowtie S$ and $PS \bowtie S \bowtie P$. This effectively specifies a constraint *on every intermediate subexpression* of a pipelined query evaluation plan. Even in this overdetermined case, our techniques generate queries that approximately satisfy the test case, albeit with higher error.

5.2.4 Efficiency



(a) Cache Effects (b) Partitioning Effects

Figure 8: Execution Times

We now describe the results of a performance evaluation of our algorithm. We conducted our experiments on a machine running Suse Linux with 4 GB memory and 3.6GHz clock speed.

We generated a set of test cases consisting of joins of n relations with n cardinality constraints, and $n+1$ range predicates, with n varying from 2 to 5. These are marked as *(Constraints,Dimensions)* in Figure 8(a). We set the *lineitem* (L) as our outer relation. As can be seen from Figure 8(a), the execution time of our framework increases as we add more constraints and range predicates. Although, this experiment was run with parameters $k = 3$, $b = 2$ and $l = 2$, similar behaviour is observed with other choices of these parameters as well. We observed that with a cold cache, a significant chunk of the execution time is spent constructing the evaluation layer. This step involves executing an index nested-loops join pipeline, with a precomputed random sample at the outer relation, and the indices of the inner relations. When we repeated the experiment with a warm cache, we can observe that the execution times

are significantly reduced. Typically, we expect multiple test cases to be provided to our system, and therefore, the process of generating the first test query should pave the cache for the other queries. Since our system relies only on samples and indices, it does not process much disk data, and as a result, it can take advantage of the large memories of modern machines.

The next experiment was performed on a warm cache to explicitly quantify the other factors that affect the execution time of the TQGen algorithm. In the experiment shown in Figure 8(b), we fixed the number of constraints to 2, and varied the number of predicates between 2 and 7 for a query involving a join of 5 tables. We varied the parameters k and l, while keeping b fixed as 2. Figure 8(b) demonstrates that the expected execution time increases with the number of dimensions d and the parameter k. Since our exploration procedure partitions a d-dimensional space using k buckets along each dimension, this time increase is expected. Alternate forms of exploration in high dimensional spaces that reduce the dimensionality of the problem by fixing the predicates on some dimensions are possible. However, we do not discuss this issue further due to space limitations. Finally, we note that in Figure 8(b), the execution times for the TQGen procedure with parameters $k = 2, l = 2$ and $k = 2, l = 3$ are identical upto 6 dimensions. This is a side effect of our network socket based communication mechanism, which results in the system being network-bound. With a more tightly coupled implementation, one would expect to see increasing execution time with increasing values of l. Similarly, although we do not show it here due to space constraints, we can also observe increasing execution times with increasing values of the branch factor b.

6. CONCLUSION

In this paper, we have investigated several aspects of the Targeted Query Generation problem. We have formally established the hardness of obtaining algorithms with approximation guarantees, and identified cases where the problem is amenable to a solution. We have introduced new best-effort algorithms that utilize novel sampling and space bounding techniques to identify test queries. Our experimental evaluation shows the utility of our techniques, and demonstrates that despite the difficulty of the problem, it is possible to generate targeted queries for reasonable test databases.

7. REFERENCES

[1] Transaction processing performance council. http://www.tpc.org.

[2] N. N. Abdelmalek. L_1 Solution of Overdetermined Systems of Linear Equations. *ACM Trans. Math. Softw.*, 6(2):220–227, 1980.

[3] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna. A genetic approach for random testing of database systems. *VLDB*, 2007.

[4] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. Qagen: generating query-aware test databases. *SIGMOD*, 2007.

[5] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. *SIGMOD*, 2002.

[6] N. Bruno and S. Chaudhuri. Flexible database generators. *VLDB*, 2005.

[7] N. Bruno, S. Chaudhuri, and D. Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE Trans. Knowl. Data Eng.*, 18(12):1721–1725, 2006.

[8] S. Chaudhuri and V. Narasayya. Program for TPC-D Data generation with skew. *ftp://ftp.research.microsoft.com/users/viveknar/tpcdskew*.

[9] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. Weinberger. Guickly generating a billion-record synthetic databases. *SIGMOD*, 1994.

[10] P. Haas, J. Naughton, S. Seshadri, and A. Swami. Fixed Precision Estimation Of Join Selectivity. *PODS*, June 1993.

[11] P. Haas and A. Swami. Sequential Sampling Procedures For Query Size Estimation. *SIGMOD*, 1992.

[12] Y. E. Ioannidis. The history of histograms (abridged). *VLDB*, 2003.

[13] P.-Å. Larson, W. Lehner, J. Zhou, and P. Zabback. Cardinality estimation using sample views with quality assurance. *SIGMOD*, 2007.

[14] M. Poess and J. M. Stephens. Generating thousand benchmark queries in seconds. *VLDB*, 2004.

[15] M. Pöss, R. O. Nambiar, and D. Walrath. Why You Should Run TPC-DS: A Workload Analysis. *VLDB*, 2007.

[16] D. R. Slutz. Massive Stochastic Testing of SQL. *VLDB*, 1998.

[17] J. M. Stephens and M. Poess. MUDD: a multi-dimensional data generator. *WOSP*, 2004.

Appendix

Proof of Lemma 2

We first prove for the case of two cardinality constraints $x_1 \leq C_1$ and $x_2 \leq C_2$. Our proof follows the same lines as the lowerbound proof given by Bruno et al. [7]. Consider a table with columns x_1 and x_2 where the domain of each column is $(1, \ldots, n)$. For a given vector $(v_1, \ldots, v_n)$, we generate a table that contains Ev_i tuples with value $(n - i + 1, i)$ for $(1 \leq i \leq n)$, and $E2^{i+j-n}n - \alpha_{i,j}$ tuples with value (i, j) where $1 \leq i \leq n, 1 \leq j \leq n, i+j > n+1$, and $\alpha_{i,j}$ is the number of tuples (i', j') such that $i' \leq i, j' \leq j$ and $(i', j') \neq (i, j)$. We can draw this table more easily as the following matrix, where entry i, j is the number of points returned with $x_1 \leq i$ and $x_2 \leq j$.

$$\begin{array}{ccccc} 0 & 0 & 0 & \ldots & Ev_n \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & Ev_3 & \ldots & 2^{n-2}En \\ 0 & Ev_2 & 4En & \ldots & 2^{n-1}En \\ Ev_1 & 4En & 8En & \ldots & 2^n En \end{array}$$

Now, select any value of N (target cardinality) between E and En. Set $v_1, \ldots, v_n$ to be any values such that $Ev_i \notin (N-E, N+E)$. The algorithm should return *no match found* with less than n calls to the evaluation layer. Therefore, there must be some point on the diagonal (i^*, i^*) which is not checked by the evaluation layer. Now we generate a new table which is identical to the previous one, except that $Ev_{i^*} \in (N - E, N + E)$. The algorithm will evaluate as before, and return *no match found* even though there is a solution (i^*, i^*). Therefore, the algorithm needs to call the evaluation layer for every point on the diagonal of the matrix.

To generalize to d dimensions, notice that the proof for 2 dimensions hinges on a construction which allows us to fix a value on every element of the diagonal $(n - i + 1, i)$ for$(1 \leq i \leq n)$, while keeping all other elements independent. In d dimensions, the equivalent construction fixed a value every element $(x_1, \ldots, x_d)$ which is a solution to the equation $x_1 + x_2 + \ldots + x_d = n + d - 1$. There are $\binom{n+d-2}{d-1}$ solutions to this equation. Therefore, we can extend the above adversarial construction to show that we must examine every one of these solutions.

A similar argument can be constructed to prove identical lowerbounds for the case when we would like to generate a query that satisfies the cardinality constraints with absolute error $\leq N\epsilon$.