Generating Queries with Cardinality Constraints for DBMS Testing

Nicolas Bruno
Microsoft Research
nicolasb@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Dilys Thomas
Stanford University
dilys@cs.stanford.edu

***Abstract*— Good testing coverage of novel database techniques, such as multidimensional histograms or changes in the execution engine, is a complex problem. In this work, we argue that this task requires generating query instances, not randomly, but based on a given set of constraints. Specifically, obtaining query instances that satisfy cardinality constraints on their sub-expressions is an important challenge. We show that this problem is inherently hard, and develop heuristics that effectively find approximate solutions.**

***Index Terms*— Query Generation, Database Testing, Cardinality Constraints.**

I. Introduction

Evaluating the performance and quality of novel database technology, such as a new multidimensional histogram or changes in the database execution engine, is not an easy task. A common methodology to validate the relative improvements of a new technique is to choose a comprehensive set of databases and queries and compare the behavior of the database system before and after the new component is incorporated. While data generation is a relatively well-studied problem (e.g., [1], [2]), query generation has been given little attention.

Consider for instance a newly designed memory manager, and suppose that we want to evaluate its impact on multi-way hash-join queries (i.e., how the per-operator memory allocation strategy influences the performance of the resulting execution plans). For a given test database, a reasonable testing plan consists of trying different query scenarios and measure their performance when the new memory manager is available. This evaluation would be meaningful only if input queries are carefully chosen to exhibit a wide range of patterns and characteristics. To that end, we could use tools like RAGS [3] or QGen [4], which can stochastically generate a large number of valid SQL statements in a short amount of time. Ideally, our testing strategy should consider join queries with varying memory requirements at each intermediate operator. The memory requirement of a hash join is determined by the size of its inputs (i.e., the sizes of the intermediate results in the query execution plan). Figure 1(a) shows a sample test query (with parameters p_1 to p_4) that joins a large table R_2 with table R_1 to obtain a small intermediate result. When this small intermediate result is joined with R_3, we get a very small final result. While it is not difficult to force a database engine to evaluate a given query using specific operators, or even fixing the join order, there is no easy way to control the sizes of intermediate joins. In this situation, randomly generating queries over the given database as described above would require an extremely large amount of time to cover the desired test scenarios. Alternatively, we could use a painful trial-and-error procedure to generate queries with cardinality constraints.

An alternative approach that we explore in this work consists of automatically generating queries based on specific semantic constraints. In this manner, we separate the problem of obtaining test queries in two stages. First, we declaratively specify semantic properties that the resulting queries should satisfy. Second, we find query instances that satisfy the constraints. While the first step depends on the component being evaluated and therefore requires manual intervention, the second step can be fully automated (though, as we will see, this is not trivial).

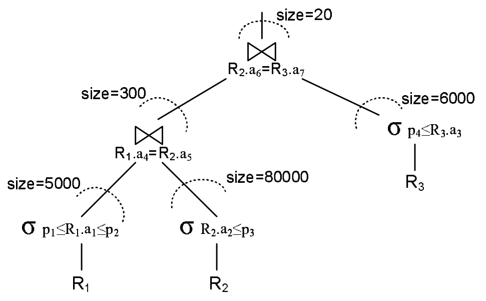
Motivated by the example above, in this work we focus on the problem of automatically generating queries with cardinality constraints on its sub-expressions (Section II formally defines the problem). We show in Sections III and IV that this problem is inherently hard, and then in Section V we develop heuristics that effectively find approximate solutions. Finally we report some preliminary results in Section VI.

II. Query Specification

We next show how to formally state the problem of generating queries with cardinality constraints as in Figure 1(a). We restrict this work to parametric conjunctive queries and parameters to range predicates in the `WHERE` clause. Specifically, we consider two types of parametric predicates: *single-sided* predicates (e.g., $p_1 \leq R.a$ or $R.a \leq p_2$), and *double-sided predicates* (e.g., $p_1 \leq R.a \leq p_2$). Additionally, we focus on constraints that restrict the cardinality of intermediate results of the input query. We now state the query generation problem:

> **Query generation problem:** Given a database D, a conjunctive query Q with parametric range predicates, and cardinality constraints[1] over sub-expressions of Q, find parameter values that make the resulting query satisfy the constraints over D.

[1]Alternatively, we can specify selectivity constraints to avoid rewriting specifications if the sizes of the database tables change. These approaches are equivalent, and we can easily transform one into the other depending on the application.



(a) Graphical representation.

Sub-expression	Card
1.SELECT * FROM R_1 WHERE $p_1 \leq R_1.a_1 \leq p_2$	5000
2.SELECT * FROM R_2 WHERE $R_2.a_2 \leq p_3$	80000
3.SELECT * FROM R_3 WHERE $p_4 \leq R_3.a_3$	6000
4.SELECT * FROM R_1,R_2 WHERE $R_1.a_4$=$R_2.a_5$ AND $p_1 \leq R_1.a_1 \leq p_2$ AND $R_2.a_2 \leq p_3$	300
5.SELECT * FROM R_1,R_2,R_3 WHERE $R_1.a_4$=$R_2.a_5$ AND $R_2.a_6$=$R_3.a_7$ AND $p_1 \leq R_1.a_1 \leq p_2$ AND $R_2.a_2 \leq p_3$ AND $p_4 \leq R_3.a_3$	20

(b) Formal specification.

Fig. 1. A parametric query with cardinality constraints.

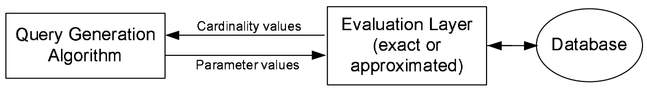

Query Generation Algorithm
Cardinality values
Parameter values
Evaluation Layer (exact or approximated)
Database

Fig. 2. Evaluation model.

As an example, Figure 1(a) can be formally specified as finding values of parameters p_1, p_2, p_3, and p_4 that satisfy the constraints in Figure 1(b). Parameters cannot be shared among *different* predicates. However, a parametric predicate might occur in multiple sub-expressions (e.g., $R_2.a_2 \leq p_3$ above is shared in queries 2, 4 and 5 in Figure 1(b)).

III. Special Complexity Results

For a given assignment of values to the parametric predicates in a constraint, we can use the DBMS to evaluate the instantiated query and verify whether the constraint is satisfied. In this section, we first use a simple evaluation model in which the only mechanism to obtain information from the given database is through an *evaluation layer* (see Figure 2) that returns the cardinality of a constrained sub-expression for a given assignment of parameters (to evaluate multiple cardinality constraints, we need to invoke the evaluation layer repeatedly, once per sub-expression). The evaluation layer can either process queries in the database or use approximations to estimate cardinality values, but we consider it as a black box. We then study lower and upper bounds for the number of invocations of such evaluation module by algorithms that solve the query generation problem. In the remainder of this section, we address the simpler case of single-sided predicates and a single cardinality constraint. Later, in Section IV we generalize the results to multiple cardinality constraints and both single- and double-sided predicates.

To simplify the presentation, we use the following *array notation* to model the evaluation layer. Consider a query constraint with k single-sided predicates $a_1 \leq p_1 \wedge \ldots \wedge a_k \leq p_k$, where a_i are attributes and p_i are parameters. Assume that n_i is the number of distinct values for attribute a_i. We model the evaluation layer as a k-dimensional $n_1 \times \ldots \times n_k$ matrix A. The value of $A[v_1, \ldots, v_k]$ for $1 \leq v_i \leq n_i$ is precisely the cardinality of the query constraint where each p_i is instantiated with the v_i-th smallest distinct value of attribute a_i. Therefore, A satisfies the following monotonicity property: $A[v_1, \ldots, v_k] \leq A[w_1, \ldots, w_k]$ when $v_i \leq w_i$ for all i.

Using the array notation described above, we can see each lookup to matrix A as representing an invocation to the evaluation layer. We now analyze the complexity of query generation algorithms by counting the number of lookups to the corresponding matrices.

A. One Parametric Predicate

Consider a query that contains a single parametric predicate:
SELECT * FROM R WHERE a $\leq$ p (Card = c)
In this situation, the matrix associated with the query constraint is a single-dimensional vector with increasing values. We can then use binary search on this vector and determine whether some value of p satisfies the cardinality constraint. Thus, an upper (and lower) bound for this problem is $\log(n)$ query evaluations, where n is the number of distinct values of attribute a.

B. Two Parametric Predicates

We next show that there is an exponential jump in complexity as we move from one to two parametric predicates ([5], pp. 143). Consider a query that contains two parametric predicates and a cardinality constraint *Card = c*:
SELECT * FROM R WHERE a$_1$ $\leq$ p$_1$ AND a$_2$ $\leq$ p$_2$

Theorem 3.1: **[Lower Bound]** A lower bound on the number of query evaluations for a single constraint with two parametric predicates $a_1 \leq p_1$ and $a_2 \leq p_2$ and cardinality c is $\Omega(n_{min})$, where n_{min} is the minimum number of distinct values in a_1 and a_2.

Proof. Consider the following family of tables with columns a_1 and a_2, where the domain of both a_1 and a_2 is $\{1, \ldots, n\}$. For a given vector $(v_1, \ldots, v_n)$ with $1 \leq v_i \leq n$, we generate a table that contains v_i tuples with value $(n-i+1, i)$ (for $1 \leq i \leq n$), and $(2^{i+j-n-1}n - \alpha_{i,j})$ 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 $(i,j) \neq (i',j')$. This is to ensure that the diagonal is $(v_1, v_2, \ldots, v_n)$ and the non-diagonal elements are constants independent of $(v_1, v_2, \ldots, v_n)$. The evaluation layer for this table is modeled by the matrix in Figure 3(a) (we show an example of such a matrix in Figure 3(b)).

Now, suppose that there is an algorithm that always returns the correct answer using fewer than n evaluations, and set all $v_i \neq c$. In such a case, such an algorithm would return no

$$\begin{bmatrix} 0 & 0 & 0 & \dots & v_n \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & v_3 & \dots & 2^{n-3}n \\ 0 & v_2 & 2n & \dots & 2^{n-2}n \\ v_1 & 2n & 4n & \dots & 2^{n-1}n \end{bmatrix}$$

(a) General matrix

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 2 & 10 \\ 0 & 0 & 3 & 10 & 20 \\ 0 & 5 & 10 & 20 & 40 \\ 1 & 10 & 20 & 40 & 80 \end{bmatrix}$$

(b) Sample matrix for $n = 5$ and $v = (1, 5, 3, 2, 4)$.

Fig. 3. Evaluation layer for two parametric predicates.

matches after examining fewer than n elements, and therefore would miss at least one element in the diagonal (say, the element at position (i^*, i^*)). Using an adversarial argument, we now generate a new instance table where all v_i are the same as before, except $v_{i^*} = c$. This algorithm would not be able to distinguish the difference between the two tables and would report that no match is found, which is incorrect. Thus, as desired, at least n probes are required. □

Theorem 3.2: **[Upper Bound]** An upper bound on the number of query evaluations for a single constraint with two parametric predicates $a_1 \leq p_1$ and $a_2 \leq p_2$ and cardinality c is $O(n_{max})$, where n_{max} is the maximum number of distinct values in a_1 and a_2.

Proof. Consider the $n_1 \times n_2$ matrix A associated with the given query constraint as defined earlier, where n_1 and n_2 are the number of distinct values in attributes a_1 and a_2. Let $S(i_1, i_2)$ denote $\{A[j_1, j_2] : i_1 \leq j_1 \leq n_1, 1 \leq j_2 \leq i_2\}$. We now show that algorithm *Walk* in Figure 4 correctly determines whether any parameter values for p_1 and p_2 satisfy the query constraint. For that purpose, we define the following invariant: $S(i_1, i_2)$ contains the un-probed elements of A that may still contain c. We show that the invariant holds by induction on the number of iterations in the algorithm. The invariant is true initially: when $i_1 = 1$ and $i_2 = n_2$, $S(i_1, i_2)$ includes all elements of A. If $A[i_1, i_2] < c$, due to the monotonicity property of A, $A[i_1, j_2] \leq A[i_1, i_2] < c$ for $1 \leq j_2 \leq i_2$. These $A[i_1, j_2]$ are precisely the elements dropped from $S(i_1, i_2)$ by the update $i_1 = i_1 + 1$ in line 4. Similarly if $A[i_1, i_2] > c$, by monotonicity $A[j_1, i_2] \geq A[i_1, i_2] > c$ for $i_1 \leq j_1 \leq n_1$, and these elements are dropped from $S(i_1, i_2)$ by the update $i_2 = i_2 - 1$ in line 5. If $A[i_1, i_2] = c$ the algorithm returns correctly. Otherwise, each iteration removes elements that cannot be equal to c, maintaining the invariant. At each iteration, either the row index i_1 is increased or the column index i_2 is decreased. Since i_1 and i_2 can only take values from 1 to $n = \max(n_1, n_2)$, the algorithm iterates at most $2 \cdot n$ times, and its complexity is $O(n)$. □

C. Multiple (>2) Parametric Predicates

We now sketch how the theorems in the previous section can be generalized for a single constraint with k parametric predicates. For simplicity, we assume that the number of distinct values for each of the k attributes is equal to n.

Lower Bound: Consider the integer solutions $1 \leq p_i \leq n$ in $p_1 + p_2 + \ldots + p_k = n + k - 1$. The number of solutions is $\binom{n+k-2}{k-1}$ (i.e., the number of ways we can place $k - 1$ delimiters among a sequence of $n + k - 2$ objects). As in Theorem 3.1, we construct a family of tables that take any value in $A[p_1, \ldots, p_k]$ for each $(p_1, \ldots, p_k)$ that is a solution of the above equation. We then use an adversarial argument to get a lower bound of $\binom{n+k-2}{k-1}$ evaluations.

Upper Bound: Consider the k-dimensional matrix A that corresponds to the given query. If we fix all but the last two indices of A, we conceptually obtain n^{k-2} two-dimensional matrices of size $n \times n$. We then use the algorithm of Figure 4 on each of these matrices. Since each execution of the algorithm requires at most $2 \cdot n$ matrix lookups, the overall search algorithm requires at most $n^{k-2} \cdot 2 \cdot n = O(2 \cdot n^{k-1})$ evaluations for $k > 1$.

```
Algorithm Walk (A: n_1 × n_2 matrix, c:integer)
1   $i_1$=1; $i_2$=n_2
2   while $i_1 \leq n_1$ AND $i_2 \geq 1$
3     if ($A[i_1][i_2] = c$) return true
4     else-if ($A[i_1][i_2] < c$) $i_1 = i_1 + 1$
5     else-if ($A[i_1][i_2] > c$) $i_2 = i_2 - 1$
6   return false
```

Fig. 4. Solving one constraint with two parameters.

IV. General Complexity Results

So far we assumed that a database invocation was the only available mechanism to obtain cardinality information from the database. We might believe, then, that other evaluation mechanisms could improve the worst case complexity of the problem. In this section we show that unless *P=NP*, we cannot obtain better results independently of the evaluation model being used.

Theorem 4.1: Given a database D and a single constraint C for a parametric conjunctive query Q, finding parameter values that make Q satisfy C on D is NP-hard.

Proof. We provide a reduction from the *subset-sum* problem [5], which takes as input an integer s and a set of integers S=$\{s_1, s_2, \ldots s_m\}$ (let us assume that all $s_i \leq 2^n$ for some n), and outputs whether there exists a subset $S' \subseteq S$ such that $\Sigma_{s_i \in S'} s_i = s$. Consider table R, shown to the right of Figure 5 (we explain below how to obtain this table). Table R has $m + 1$ columns and $T = \sum s_i$ rows. The rows in R are clustered in m groups, where the i-th group has s_i tuples with $id = i$, and $a_j = 1$ if $i = j$ or $a_j = 0$ otherwise. We ask for the following query:

```
SELECT * FROM $R$                              (Card = $s$)
WHERE $p_{1,1} \leq a_1 \leq p_{1,2}$ AND
      $p_{2,1} \leq a_2 \leq p_{2,2}$ AND ... AND
      $p_{m,1} \leq a_m \leq p_{m,2}$
```

Suppose now that we obtain a solution for this problem (i.e., parameters for the query that make it evaluate to s results). We note that since all predicates are over columns a_i, then the result of any query that contains some tuple with $id = k$ must

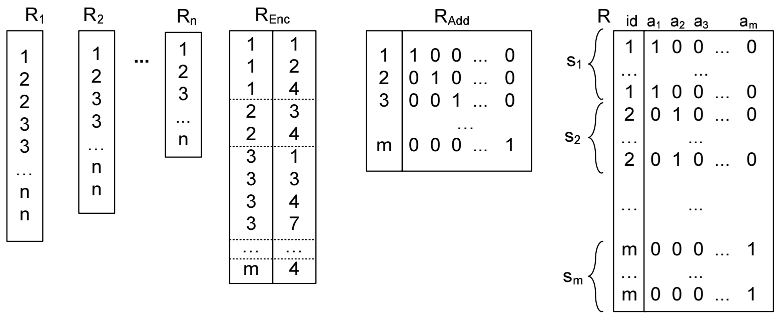


Fig. 5. Reducing *subset-sum* to the query generation problem.

contain *all* tuples with $id = k$ (these are all indistinguishable for the query). Therefore, the answer of the resulting query has s tuples and contains all the tuples in a subset of values of id. The groups returned by this query therefore induce a solution for the original subset-sum problem.

Conversely, we now show that all subset-sums from the original S can be obtained by suitable choices of parameters. Consider S', an arbitrary subset of elements in S. If $s_i \in S'$, we instantiate the $i\text{-}th$ predicate as $0 \leq R.a_i \leq 1$. Otherwise, we use predicate $0 \leq R.a_i \leq 0$. In this way, the answer of such query would contain all tuples except those that have $R.a_i = 1$ while $s_i \notin S'$ (i.e., $\sum_{s_i \in S'} s_i$ of them). Therefore, if the algorithm returns no answers, it means that there is no solution for the corresponding subset-sum problem. Consequently, both problems are equivalent.

Explicitly having table R, however, is not possible because its size is in the worst case exponential in the size of the subset-sum problem. We next show how we can encode R using polynomial space (see Figure 5). For that purpose, we consider tables $R_1, \ldots, R_n$, where R_i contains one instance of all numbers between 1 and i, and two instances of all numbers between $i+1$ and n (recall that all $s_i \leq 2^n$). Clearly, by equi-joining all R_i tables, we obtain a new table that contains 2^{i-1} duplicates of value i, for $1 \leq i \leq n$. Now consider table R_{Enc}, with two columns, defined as follows. Table R_{Enc} contains m groups of rows, each one encoding value s_i as follows. Let $s_i = b_{i_1} b_{i_2} \ldots b_{i_n}$ be the binary representation of s_i. The $i\text{-}th$ group contains as many rows as digits with value one among the b_{i_j}. Each row has value i in the first column, and j in the second column, for each $b_{i_j} = 1$. For instance, the first group in Figure 5 encodes value $s_1 = (11)_{10} = (1011)_2$ and the second group encodes value $s_2 = (12)_{10} = (1100)_2$. Now, if we join the previous m equi-joined tables with R_{Enc} on its second column, we duplicate each group in R_{Enc} exactly s_i times, which corresponds to the first column in the desired table R. A final join with table R_{Add} in the figure results in the original table R that we used in the proof. Tables $R_1, \ldots, R_n, R_{Enc}$ and R_{Add} are therefore a polynomial-sized implicit representation (specifically, $O((m+n) \cdot n)$) of R. We thus replace in the original query the occurrence of R by $R_1 \bowtie \ldots \bowtie R_n \bowtie R_{Enc} \bowtie R_{Add}$ to prove the main result. $\Box$

V. Heuristic Hill-Climbing Approach

As explained in the previous section, the query generation problem is in general NP-hard. In many cases, however, we do not need an exact solution for a given query specification. Suppose there is a constraint with cardinality c=1000 and we obtain parameters that make the corresponding query return c'=995 tuples. Our initial goal is to generate query instances for coverage testing. It is likely that the query characteristics we were interested on initially are preserved for the slightly different query above (e.g., for the example in the introduction, we know that memory requirements for plans with slightly different cardinality values do not change drastically). We relax the original query generation problem so that approximate solutions are acceptable[2]. We then define the *relative error* for a constraint as $error$=$\max\left(\frac{c'}{c}, \frac{c}{c'}\right)$ (if c or c' are zero we arbitrarily replace them with one). We chose relative errors over absolute error as the cardinalities of different constraints may differ in orders of magnitude as in Figure 1. In general, for multiple constraints, we search for parameters that minimize the average relative error.

The search algorithm presented in this section is a hill-climbing variant motivated by the techniques introduced in Section II. This algorithm can be described as a *walk* on the parameter space. At any point in time, our proposed algorithm is in a *state* (i.e., a point in the parameter search space). From the current state the algorithm tests a few *steps* and chooses the one that decreases the error metric the most, stopping when we reach a state that is good enough.

Search Space of Parameter Values: We encode the state for a single-sided parametric predicate (e.g., $R.a < p$) using a number s, with $0 \leq s \leq 1$. In turn, we encode the state for a double-sided parametric predicate (e.g., $p_1 < R.a < p_2$) with a pair of numbers s=(s_1, s_2), such that $0 \leq s_1 \leq s_2 \leq 1$. We obtain the actual parameter values from these encoding by considering s as quantiles in the parameter domains. For instance, $s = 0.5$ for $R.a < p$ encodes p to be the median of the attribute $R.a$ (we use single-column histograms [6] to translate quantiles to actual values in the attribute domains). The state space for multiple parametric predicates is the cross product of states for individual predicates. If $s_1, \ldots s_k$ are states for parametric predicates $P_1, \ldots P_k$, then $(s_1, \ldots, s_k)$ represents the combined state space for the k predicates.

Initialization: We show how to obtain an initial point for our search algorithm that is optimal if the parametric predicates are independent. Let $p_1, \ldots, p_m$ be the parametric

[2]Alternatively, we can relax the original problem by allowing ranges of cardinality constraints, but we do not explore this scenario in the paper.

predicates, and $C_1, \ldots, C_n$ be the cardinality constraints. Assume that C_j contains k_j parametric predicates $p_{j_1}, \ldots, p_{j_{k_j}}$. For each p_i $(1 \leq i \leq m)$ we define a variable l_i, which represents the initial selectivity value for the parametric predicate p_i (l_i are selectivity values, so $0 \leq l_i \leq 1$). Let t_j $(1 \leq j \leq n)$ be the cardinality of constraint C_j where all parametric predicates are removed, and c_j denote the desired cardinality of C_j as specified. Then, the joint selectivity of all parametric predicates for C_j is $sel_j = c_j/t_j$. *Assuming independence* among predicates, the value sel_j is the product of the selectivities of each parametric predicate. Thus, for each constraint C_j ($1 \leq j \leq n$) we write $l_{j_1} \times l_{j_2} \times \ldots l_{j_{k_j}} = sel_j$. Taking logarithms on both sides we obtain a system of n linear equations given by $\log(l_{i_1}) + \log(l_{i_2}) + \ldots \log(l_{i_{k_i}}) = \log(sel_i)$ for $1 \leq i \leq n$. We are interested in values for l_i $(1 \leq i \leq m)$ that minimize the sum of errors in $\log(sel_j)$ $(1 \leq j \leq n)$. Additionally, we have m constraints of the form $0 \leq l_i \leq 1$. Rather than using linear programming [7], we use an approximation by recursive least square estimators [8] that is more efficient. The result of this algorithm is a set of selectivity values that minimize the relative error of the input constraints *assuming independence* among predicates. Once such selectivity values l_i are obtained, we calculate the algorithm's initial state as shown in Figure6(a).

Predicate	Initialization state s
$p \leq R.a$	$s = 1 - l$
$p_1 \leq R.a \leq p_2$	s=$(s_1, s_1 + l)$, where s_1=Random(0..$1 - l$)
$R.a \leq p$	$s = l$

(a) Initializing parameters for selectivity l

Original state	Resulting states
s	$(s + sz)$, $(s - sz)$
(s_1, s_2)	$(s_1 + sz, s_2)$, $(s_1, s_2 + sz)$, $(s_1 - sz, s_2)$, $(s_1, s_2 - sz)$, $(s_1$+sz, s_2+$sz)$, $(s_1 - sz, s_2 - sz)$ $(s_1$-sz, s_2+$sz)$, $(s_1 + sz, s_2 - sz)$

(b) Steps in the state space.

Fig. 6. Details of the hill-climbing-based algorithm.

Steps: Following a hill-climbing approach, we *move*, at each iteration, towards the region in the state space that reduces the error metric the most. Specifically, a *step* is a change in the parameter's state. Let the step size be denoted by sz $(0 \leq sz \leq 1)$. From a state in the parameter space, we consider two steps for single-sided predicates and eight steps for double-sided predicates as shown in Figure 6(b). (If any of the final state values fall outside $[0, 1]$ they are rounded to the boundary point, and if the left parameter value becomes larger than the right parameter value in a double-sided predicate, the step is discarded.) For multiple parameters, we consider steps along a single parameter at a time (i.e., we do not consider *diagonal* steps). This pruning technique reduces the search time from d^m to $d \cdot m$ evaluations for d types of steps and m predicates. The rationale for this pruning is that if the error metric were continuous and the current state were not a local minima, at least one partial differential (a step on a single parameter) would show a decrease in the relative error, and pruning would not compromise quality.

```
1  initialize parameter state (Section V)
2  sz =1; maxHalve=log(n)
3  for i=1 to maxHalve
4    while (step decreases error metric)
5      use step that most decreases error
6    sz = sz/2.
```

Fig. 7. Hill-climbing algorithm with halving search.

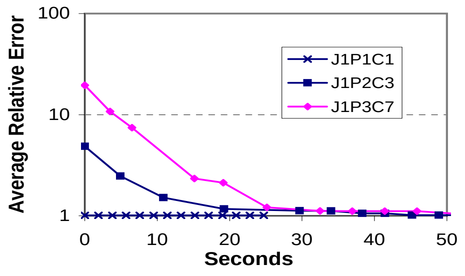
Main Algorithm: The main algorithm is summarized in Figure 7. We start with step size sz=1. When the algorithm cannot find a step that decreases the error in line 4, sz is halved in line 6. This halving is done until the final step size sz can distinguish a single distinct value, guaranteeing convergence to a local minima of the error function. Let a single stage of the algorithm be the execution of lines 4-5 with a constant sz. In practice we observe that at any stage the number of steps in a direction made for a single parameter is at most two. Otherwise, a larger step would have been made in the previous stage when sz was double its current value. Thus, if there are k parametric predicates and d types of steps per predicate, we expect each stage to have $O(d \cdot k)$ steps. If the largest number of distinct values on any attribute in a parametric predicate is n, there will be at most $\log(n)$ stages per attribute, therefore resulting in $O(d \cdot k \cdot \log(n))$ steps before the algorithm converges.

Other Optimizations: We additionally use optimizations for efficiency and robustness, such as starting with multiple initial points to avoid local minima, pruning some steps at each iteration depending on the query, and decomposing a big query into smaller problems when certain properties are satisfied.

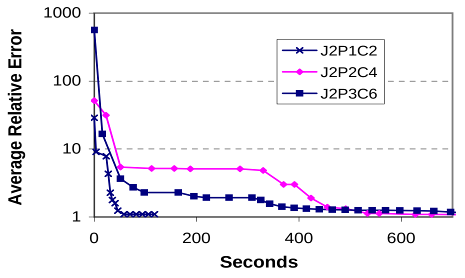
VI. Experimental Evaluation

We next report an experimental evaluation of our technique of Section V to generate queries with cardinality constraints. We generated a synthetic database that consists of three tables, ranging from 10^4 to 10^6 tuples, joined via (quasi) foreign keys. Attribute values are generated with different degrees of skew and correlation, and joins do not satisfy referential integrity. We then generated query specifications using the following encoding. For integers n_1, n_2, and n_3, we denote by $J_{n_1}P_{n_2}C_{n_3}$ a specification of a query with n_1 joins, n_2 parametric predicates, and n_3 cardinality constraints. For instance, $J_2P_1C_3$ represents a specification of a two-way join query with a single parametric predicate and three cardinality constraints.

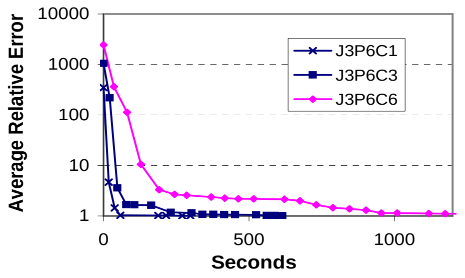
Figure 8 shows the average relative error during the execution of our hill-climbing-based algorithm for different query specifications. In most cases the initial average relative error is very large, sometimes above 10000%. This means that the queries we are interested in significantly deviate from independence among their parametric predicates. Local minima are not uncommon, and sometimes these solutions have relative errors that are significantly larger than the ones reported in the figure. Our approach leads to a more robust strategy by trying multiple initial points. In all cases, the final average relative error is below 1.09 (i.e., 9% of average difference



(a) Single-table queries.



(b) Two-way join queries.



(c) Three-way join queries.

Fig. 8. Average relative error over time for different query specifications.

between the desired and actual constraints). Not surprisingly, the more constraints that are present in the query specification or the more complex the constraints, the longer it takes to converge because there are fewer states in the search space that are relatively accurate. If the right indexes are present in the system, we obtained an efficiency improvement of an order of magnitude (we omit these results due to space constraints). While there is room for improvement (our techniques can take seconds, and sometimes minutes to return query instances), we believe that in our context this can be tolerated. In fact, the objective is not to generate large workloads but queries with specific characteristics to evaluate and pinpoint complex behavior in a system.

VII. Conclusions and Future Work

In this work we considered the problem of generating queries with cardinality constraints. We showed that in general the problem is computationally hard, and developed heuristics that efficiently return approximate results. It is important to investigate whether our techniques can be complemented by specific data generation tools (i.e., generating both database and query instances might be a competitive alternative if we are not constrained to use a specific data source).

References

[1] N. Bruno and S. Chaudhuri, "Flexible database generators," in *Proceedings of the 31th International Conference on Very Large Databases (VLDB)*, 2005.

[2] A. Neufeld, G. Moerkotte, and P. C. Lockemann, "Generating consistent test data for a variable set of general consistency constraints," *VLDB J.*, vol. 2, no. 2, 1993.

[3] D. R. Slutz, "Massive stochastic testing of SQL," in *Proceedings of the 24th International Conference on Very Large Databases (VLDB)*, 1998.

[4] M. Poess and J. M. Stephens, "Generating thousand benchmark queries in seconds," in *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, 2004.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (Second Edition)*. MIT Press, 2001.

[6] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita, "Improved histograms for selectivity estimation of range predicates," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1996.

[7] V. Chvatal, *Linear Programming*. W. H. Freeman, 1983.

[8] S. Haykin, *Adaptive Filtering Theory (Chapter 9)*. Prentice Hall, 2002.