Interactive Query Refinement

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

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

ABSTRACT

We investigate the problem of refining SQL queries to satisfy cardinality constraints on the query result. This has applications to the many/few answers problems often faced by database users. We formalize the problem of query refinement and propose a framework to support it in a database system. We introduce an interactive model of refinement that incorporates user feedback to best capture user preferences. Our techniques are designed to handle queries having range and equality predicates on numerical and categorical attributes. We present an experimental evaluation of our framework implemented in an open source data manager and demonstrate the feasibility and practical utility of our approach.

1. INTRODUCTION

Traditional relational database systems support a boolean retrieval model, in which constraints are specified on properties of individual tuples and not on the result table as a whole. As a consequence of this model, an SQL query cannot ensure a *result cardinality* when executed on a given database. However, there are several situations where one would like the associated queries to satisfy a cardinality constraint on the result size. Such cases often occur in Business Intelligence applications, where analysts pose queries on existing corporate databases. For instance, consider a marketing scenario, in which a bank seeks to extend a promotional offer to ten thousand *young*, *high-income professionals*. Given a customer database, one can express these conditions as predicates on the `dateOfBirth`, `salary` and `profession` attributes. However there exists no means other than a cumbersome trial and error procedure for setting the predicates such that the resulting query returns 10K tuples when executed on the given database. In general, today's relational database management systems lack support for tying the query predicates to the output cardinality of a query in an automated fashion. This paper seeks to address this gap.

We refer to the problem of queries returning too many or too few tuples as the *many/few answers* problem. For the many answers case, Carey and Kossmann [4] proposed a `STOP AFTER` operator in SQL to limit the cardinality of a query. Typically, this clause is combined with an `ORDER BY` clause, resulting in a *Top-k* query processing problem [11]. *Top-k* based approaches have been proposed for the few answers problem as well [1]. A fundamental requirement of these techniques is that they require a scoring function. Defining such a scoring function is often a non-trivial task especially when defined on multiple semantically distinct attributes. For instance, to consider the bank example described above, a *Top-k* approach would require a scoring function that combines temporal (`dateOfBirth`), monetary (`salary`) and categorical (`profession`) attributes to return a single score.

An important feature of the many/few answers problem is that users often have preferences on how to transform the original query to increase/decrease the result size. To consider the bank query above, a bank might prefer *younger* customers if it is offering a new account, while it might prefer *high-income* customers if it is extending a new investment opportunity. Defining scoring functions to express such *application-specific* preferences over multiple attributes is challenging. Moreover, such functions provide at best indirect control over the set of tuples returned.

In this paper, we propose the *Stretch 'n' Shrink* (*SnS*) framework for the many/few answers problem which enables *interactive user-aided refinement* of queries to a given result cardinality through transformations of the selection predicates. The transformations take the form of *relaxations* i.e *stretching* query predicates to increase the query cardinality, and *contractions* i.e *shrinking* query predicates to decrease the cardinality. This model of refinement offers the advantage of not requiring the user to define a separate scoring function. Therefore, it generates queries which can utilize traditional query processing primitives without the need of additional infrastructure for processing ranking queries efficiently [19]. Additionally, by casting the many/few answers problem as a predicate transformation problem, this framework is able to explicitly capture preferences on how the query is to be refined, as detailed in the following examples.

EXAMPLE 1. *Consider a query with the predicates* `year > 1990 AND cost < 5000` *returning too few answers. There are many ways in which the query could be refined to satisfy the target result size. For instance, one could change the predicate on* `year` *to* `year > 1980` *while leaving the other predicate unchanged. Alternatively, one could change only* *the predicate on* `cost` *to* `cost < 7000`*. Or, both the predicates could be modified together to* `year > 1987 AND cost < 6500`*. Each of these choices involves relaxing one or more of the predicates i.e increasing the selectivity of the predicates.*

Similarly, for the case of categorical predicates:

EXAMPLE 2. *Consider a query with predicates* `country = 'USA' AND state = 'California'` *, returning too many answers. This query could be refined by adding additional predicates such as* `city = 'Santa Cruz' OR city = 'Milpitas'` *or predicates* `city = 'San Francisco' AND Zip = '94112'`*. These additional predicates further constrain the results returned by the query and reduce its cardinality to the target answer size.*

These examples illustrate that there might be multiple queries that satisfy the result cardinality constraint. Our goal in this paper is to provide a navigational framework that enables users to interactively refine queries as per their preferences. We make the following contributions in this work:

- We formally define the problem of query refinement for queries with range and/or equality predicates which return too many or too few answers.
- We introduce the *SnS* framework for *Interactive Query Refinement* which encompasses all the cases outlined above. *SnS* supports a novel model of query refinement which keeps a user "in the loop" guiding one towards a query that best satisfies the target cardinality.
- We describe the sampling and indexing procedures underlying *SnS*, which are designed to aid accurate and efficient query refinement. We present an implementation and evaluation of the framework in a database system, demonstrating its practical utility.

2. RELATED WORK

There has been some research on modifying query predicates with the intent to relax the query and generate more answers. Chaudhuri [5] introduced a formal model for modifying SQL queries in order to increase their output cardinality. Similarly, Chu and Chen [10] also defined a formal model of query relaxation using a type abstraction hierarchy on attributes, and proposed an extension (CSQL) of SQL to specify such queries. However, both of these are primarily formal models, and the papers do not investigate issues in practically realizing such models, especially with respect to ensuring a target cardinality. More recently, Kadlag et al. [21] introduced algorithms for relaxing multiple predicates using multidimensional histograms. However, their technique is does not consider join queries, nor does it incorporate user preferences.

The typical solution proposed in the literature to handle the many answers problem is to utilize scoring functions and return only the *Top-k* ranked results [11, 7]. The primary problem with this approach is the requirement of a scoring function which may not be readily available. In addition, *Top-k* query processing is typically performed over single tables; optimizing *Top-k* queries over joins is a challenging problem [19]. An alternative approach, used in the many answers case, is to compute the skyline of the query results [2]. However, skyline computation over joins is expensive and predicting the size of the skyline is difficult [13, 6].

There has been much research on detecting and relaxing queries with empty results. Agrawal et al. [1] introduced a technique utilizing ranking algorithms in this context. More recently Luo [23] proposed a method for detecting empty result queries using information collected from previously executed queries. This technique, which uses materialized view matching cannot be generalized to the few answers problem though. Similarly, Koudas et al. [22] introduce a technique for relaxing an empty query by computing the set of results which are closest to the original query as per skyline semantics. However, the method requires expensive skyline computation and cannot provide guarantees on the relaxed result size. In the context of text search, Fontoura et al. [12] recently introduced a model of query relaxation along multiple hierarchical taxonomies. However, their model incorporates a cost based procedure for relaxation, and does not incorporate individual user preferences on query relaxation.

Recently, there has been increasing interest in the problem of generating *targeted test queries* that satisfy cardinality constraints on multiple intermediate subexpressions [3, 26]. This paper extends the applicability of such frameworks by adding support for categorical predicates, and introducing techniques for incorporating user preferences.

3. PRELIMINARIES

3.1 Model

Consider a conjunctive SPJ (Select-Project-Join) query Q with selection predicates. We consider selection predicates to include range ($<, \leq, >, \geq$) and equality ($=$) predicates. Each predicate can be defined on a *numeric* or *categorical* domain. We consider a numeric domain to be any domain on which a range predicate is defined. In the rest of the paper, for ease of presentation, we assume that the numeric domain is the domain of integers, although our techniques work for general numeric domains.

Unlike numeric domains, categorical domains permit only equality predicates. In this paper, we consider the class of hierarchical categorical predicates. Categorical predicates in databases often implicitly express hierarchies e.g (`Country`, `State`, `City`, `Street`, `No.`) or (`Genre`, `Artist`, `Song`). Each attribute is thus at some *level* of a hierarchy, with the root at the smallest level. For instance, `Country` is at level 1, and `Street` is at level 4. We define the *level* of a categorical hierarchical predicate to denote the maximum level at which a predicate is defined on the hierarchy. Thus `Country = 'US' and State = 'FL'` has a level of 2.

We assume knowledge of the hierarchies defined by a database schema. Such hierarchies can be specified by the user or the schema designer. We note that the presence of hierarchies is a *feature* and not a *requirement* of *SnS* framework. In the absence of information about hierarchies, we treat each attribute as a single level hierarchy by itself.

Given query Q, and a target result cardinality k, we generate a new query Q' by *refining* the selection predicates of Q. We term the predicate transformations that increase the cardinality of Q as *relaxations*, and the transformations that decrease the cardinality as *contractions*. We next define the rules for relaxing and contracting numeric and categorical predicates.

3.1.1 Numeric Predicate Refinement

Consider a numeric predicate $P_i : x_i < C_i$. A *relaxation* of P_i is any predicate $P'_i : x_i < C'_i$ s.t $C'_i \geq C_i$. Effectively, we have $P_i \subseteq P'_i$. Likewise, a *contraction* of P_i is any predicate $P'_i : x_i < C'_i$ such that $C'_i \leq C_i$ i.e $P'_i \subseteq P_i$.

We can convert any predicate on a numeric domain to a predicate of the form $x_i < C_i$. For instance, a predicate $x_i > C_i$ can be transformed into $-x_i < -C_i$. We consider range predicates of the form $C^l_i < x_i < C^u_i$ as two separate predicates $-x_i < -C^l_i$ and $x_i < C^u_i$. In the rest of the paper, for ease of exposition, we assume that the numeric predicates have been appropriately transformed into predicates of the form $x_i < C_i$.

3.1.2 Hierarchical Categorical Predicate Refinement

We would like to define relaxation and contraction to generate supersets and subsets (respectively) of the original hierarchical predicate in an analogous fashion to numeric predicates. In the following, we use as a running example the predicate: `Country = 'US' and State = 'FL'`.

Hierarchical Relaxation: We consider two notions of relaxation, namely *Expansion* and *Roll-up*. Expansion denotes the process of *disjunctively* adding additional predicates at the current level of the hierarchical predicate. Thus, we can expand the example predicate to `Country = 'US' and ( State = 'FL' OR State = 'CA' )`. Likewise, *roll-up* is the process of removing all predicates from the current level of the hierarchy. Thus, the example predicate can be rolled up to obtain `Country = 'US'`. We note that this notion of roll-up is analogous to the roll-up operation on data cubes.

Hierarchical Contraction: Similar to the forms of relaxation discussed above, we consider two notions of contraction, namely *shrinking* and *drill-down.* Shrinking is the inverse operation of expansion, in which predicates are removed from a disjunction at the current level of the hierarchy. Drill-down is the inverse operation of roll-up, in which additional predicates are added conjunctively at the next level of the hierarchy. The example predicate can be drilled down to obtain `Country = 'US' and State = 'FL' and City = 'Miami'`.

3.2 Problem Definition

We have defined notions of relaxation and contraction for numeric and categorical domains. We refer to each numeric predicate or categorical hierarchy as a *dimension* of the query.

EXAMPLE 3. *The query:*
*Select * from T*
Where weight < 120 and age < 40
and country = 'US' and state = 'FL'
has 3 dimensions, one for each of the two range predicates and one for the hierarchy (Country, State, ...)

A query is therefore relaxed or contracted along its dimensions. We use d to represent the number of dimensions of a query. We now define the problem of *Query Refinement* as:

DEFINITION 1. ***Query Refinement Problem:*** *For a given SPJ query Q, and a target result cardinality k on database D, generate a query Q' satisfying the following conditions. (i) Q' is generated by using either only relaxations or only contractions along the dimensions of Q. (ii) Q' when executed on D returns k tuples in its result. (iii) There is no other query Q'' satisfying conditions (i) and (ii) such that the user prefers Q'' over Q'.*

We note that the requirement of using only relaxations or only contractions (Condition (i)) is to prevent predicate refinements that cancel each other out. Additionally, it ensures that the space of possible refinements can be bounded, since otherwise one could transform any query to any other query using these transformations. Effectively, when Q is estimated to return fewer tuples than k, Q is relaxed; when Q returns too many tuples, it is contracted.

3.2.1 The Need for Approximation

Condition (ii) in problem definition above requires the generation of a query that returns *exactly* k tuples. This may be difficult since there might be no query that exactly satisfies the target cardinality. Consider a simple selection query with only a single predicate $x < 20$ on column x containing 100 distinct values $(1, \ldots, 100)$, with each value having a frequency of 10. In this case, if the target cardinality is 505 tuples, the best one can do is to refine the predicate to $x < 52$ to obtain 510 tuples. Additionally recent results show that the problem of generating a query that satisfies an output cardinality is hard to solve exactly [3] or to approximate to within a constant absolute or relative error [26]. As a consequence, our framework enables user-aided exploration of the search space to return a query that best captures user preferences, with an acceptable (for the user) error in output cardinality.

3.3 Terminology

Given a query Q with d dimensions, we define boundaries within which the predicate along a given dimension can be relaxed or contracted. This is captured by the notions of maximal relaxations and contractions, as defined next.

DEFINITION 2. ***[Maximal Relaxation] (numeric)*** *Given a numeric predicate $P_i : x_i < C_i$, its maximal relaxation is a predicate $P^m_i : x_i < C^m_i$ such that (i) $C^m_i \geq C_i$. (ii) The refinement Q' of Q produced by refining only the predicate P_i to P^m_i is estimated to return at least k tuples. (iii) There is no predicate $P^{m'}_i : x_i < C^{m'}_i$ such that $C^{m'}_i < C^m_i$ and $P^{m'}_i$ satisfies conditions (i) and (ii). In the absence of a predicate satisfying all three conditions, the maximal relaxation is set to $x_i < \infty$.*

Effectively, for a numeric predicate, the notion of a maximal relaxation defines the boundary upto which one could relax the predicate to satisfy the target cardinality. If the predicate is relaxed further, other predicates must be contracted, violating the requirement of using only relaxations or only contractions.

An analogous notion of *maximal contractions* can similarly be defined for numeric predicates. A maximal contraction bounds how much one can contract a given numeric predicate to obtain the target cardinality. We use the term *maximal transformation* as a generic term for maximal relaxations and contractions (depending on whether the query is to be relaxed or contracted), and denote its value as $P^m_i : x_i < C^m_i$ or (where the context is clear) as just the constant C^m_i.

Definition 3. ***[Maximal Relaxation] (categorical)*** *Given a hierarchical categorical predicate $P_i : x_{i1} = C_{i1} \wedge \ldots \wedge x_{il} = C_{il}$ having a level l, its maximal relaxation is a predicate $P_i^m : x_{i1} = C_{i1} \wedge \ldots \wedge x_{il^m} = C_{il^m}$ such that: (i) The new level $l^m \leq l$. (ii) The refinement Q' of Q produced by refining only the predicate P_i to P_i^m is estimated to return at least k tuples. (iii) P_i^m is a roll-up of P_i. (iv) There is no predicate $P_i^{m'}$ that satisfies conditions (i) - (iii), and has $l^{m'} \geq l^m$. In the absence of such a predicate, the level of the maximal relaxation is set to 0.*

Similar to the case of numeric predicates, maximal relaxations for hierarchical categorical predicates bound the level upto which the predicates along the dimension can be rolled up. However, the analogous notion of *maximal contraction* is not defined for categorical hierarchies due to the multiple possible paths for drilling down a hierarchy.

Given these definitions of maximal relaxations and contractions, we next define two queries at the core of our refinement procedures.

Definition 4. ***[Extended Query]*** *Q^e : Given a query Q with d dimensions, the extended query Q^e is the query which returns tuples satisfying the predicates of Q along at least $d - 1$ of the d dimensions.*

Example 4. *The extended query for the query in Example 3 is*

```
Select * from T where
(age < 40 and country = 'US' and state = 'FL') OR
(weight < 120 and country = 'US' and state = 'FL') OR
(weight < 120 and age < 40)
```

Observe that Q^e is a superset of Q; Q^e is also a superset of any query generated by relaxing Q *along only one dimension.*

Definition 5. ***[Bounding Query]*** *Q^b : Given a query Q with d dimensions, the bounding query Q^b is a refinement of Q with the predicates along each dimension maximally relaxed.*

If Q is to be contracted, Q^b is identical to Q, and is therefore a superset of all queries generated by contracting the predicates of Q. If Q is to be relaxed, Q^b is a superset of any query Q' satisfying the target cardinality constraint, where Q' is generated by relaxing the predicates of Q. Therefore the bounding query Q^b bounds all possible solutions to the query refinement problem.

Example 5. *If the query in Example 3 is to be relaxed, a possible bounding query is:*

```
Select * from T
Where weight < 170 and age < 60
and country = 'US'
```

with the values in bold text indicating the maximal relaxations along each of the three dimensions.

The bounding query Q^b bounds the search space for refinements of Q that satisfy the target cardinality constraint. Generating Q^b requires computation of maximal relaxations along each dimension of Q. The maximal relaxations can be computed by utilizing the extended query Q^e, which is a superset of all queries that relax Q along only one dimension. Before we describe our procedures for performing such computations, we outline the cardinality estimation scheme underlying our framework.

3.4 Cardinality Estimation Scheme

Our query refinement framework requires fast and accurate cardinality estimates for any potential refinement Q' of the original query Q. These estimates could be obtained from the cardinality estimation component of the database system. However, such estimates are often incorrect, especially for queries with multiple joins and selection predicates [20]. The accuracy of refinement directly depends on the cardinality estimation scheme deployed. Therefore, we utilize sampling based estimators for cardinality estimation. In order to avoid sampling repeatedly for each refinement considered, we deploy a *Superset Sampling Estimator (SSE)* which provides fast and accurate cardinality estimates.

Consider an SPJ query Q^s, which we term as a *superset query.* Given Q^s, *SSE* provides cardinality estimates for *any query* $Q' \subseteq Q^s$ to within an error $\epsilon|Q^s|$ with high probability. We next describe *SSE* if Q^s is a single relation or a join query.

3.4.1 Single Relation Queries

Suppose Q^s is a selection query on a single relation A. Since Q^s is defined on a single relation A, a random sample of Q^s can be obtained by sampling from the underlying relation A, and applying the predicates of Q^s. We wish to specify guarantees for using this random sample for estimating the cardinality of any query $Q' \subseteq Q^s$.

A *range space* is a set system, defined by a set P and a set of subsets $\mathcal{R}$ (*ranges*) of P. In our current problem setting, the set P is the superset query Q^s while the ranges are all possible queries $Q' \subseteq Q^s$. An ϵ-approximation E_P of a set P for a range space $\mathcal{R}$ has the property that for any $R \in \mathcal{R}$

$$\left| \frac{|P \cap R|}{|P|} - \frac{|E_P \cap R|}{|E_P|} \right| \leq \epsilon$$

An ϵ-approximation E thus guarantees selectivity estimation to within a $1 \pm \epsilon$ interval. The following lemma of Vapnik and Chervonenkis [27, 16] links the size of a random sample of a set, and the error guarantee obtained using the sample for approximate range counting.

Lemma 3.1. *For any range space with finite VC dimension, a random sample of size $O(\frac{1}{\epsilon^2} \log \frac{1}{\epsilon \delta})$ is an ϵ-approximation with probability $1 - \delta$.*

This Lemma provides guarantees on the size of the random sample of Q^s required to estimate the cardinality of *any* query $Q' \subseteq Q^s$ to within $\epsilon|Q^s|$ with high probability. We note that this random sample needs to be computed only once, and can then be kept in memory.

3.4.2 Join Queries

The *SSE* procedure for a single relation query relies on the fact that one can easily obtain random samples of a base relation. If the superset query Q^s is a join query over multiple relations, then obtaining a uniform random sample of Q^s is known to be difficult [8].

However, utilizing random samples of base relations for join cardinality estimation is a well known technique in database literature [14]. In this work we deploy the *t_index* join cardinality estimation scheme [14] which obtains a random sample from the outer relation, and joins it with indexes on the other relations. We note however, that *SSE* for joins can utilize any alternative sampling based join cardinality estimation scheme as well.

If a random sample of the outer relation of size n tuples joins with the inner indexes to produce n_{join} tuples, the cardinality of the join can be estimated as $n_{join} \times N_{outer}/n$ where N_{outer} is the size of the outer relation. In terms of selectivity, Haas et al. [14]. show that, under certain assumptions, if μ is the actual selectivity of the join, and μ_n is the estimated selectivity after n tuples have been read, then:

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

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.

The *t_index* procedure described here provides a useful means to obtain accurate cardinality estimates for any join query. However, the cost of performing such estimation can be prohibitively high if a join is performed with the inner indexes for each query for which a cardinality estimate is required. Instead, *SSE* executes the *t_index* procedure *only once* for the superset query Q^s. The tuples produced by this procedure are stored in an in-memory data structure. This set of tuples serves as an ϵ-approximation E_{Q^s} for estimating the cardinality of any query $Q' \subseteq Q^s$ to within $\epsilon|Q^s|$ with confidence bounds derived from Equation 1.

Given a superset query Q^s, a target error bound ϵ, and a confidence probability $1 - \delta$, an invocation of *SSE* with Q^s i.e $SSE(Q^s)$ generates an ϵ-approximation E_{Q^s} for the purposes of estimating the cardinality of all queries $Q' \subseteq Q^s$. The primary advantage of utilizing *SSE* is that one can tune the parameters to obtain estimates of desired accuracy, and avoid making any independence assumptions. Our query refinement framework invokes either of the two versions of *SSE* described here, depending on whether the original query is a single relation query or a join query.

3.5 The Stretch 'n' Shrink Framework

We now provide a high level overview of the *Stretch 'n' Shrink* (*SnS*) framework for Interactive Query Refinement. *SnS* refines a query in two phases, with each phase utilizing the *SSE* procedure described in Section 3.4.

3.5.1 Phase 1: Computing Bounds

The goal of the first phase is to:

- Estimate the cardinality of the original query Q and identify whether it is to be relaxed or contracted.
- Compute maximal relaxations and contractions along each dimension of Q and generate the bounding query Q^b.

In order to perform such computation, *SnS* invokes *SSE* with the extended query Q^e as the superset query to generate an ϵ-approximation E_{Q^e}. Since the original query $Q \subseteq Q^e$, one can estimate the cardinality of Q using E_{Q^e}, and identify whether the query is to be relaxed or contracted. Maximal relaxations and contractions can similarly be computed using E_{Q^e}, since they correspond to queries which relax or contract the original query along only one dimension. *SnS* generates the bounding query Q^b by refining Q to its maximal relaxations along each dimension. We provide further details of the first phase in Section 4. We note that this phase does not require any user intervention.

3.5.2 Phase 2: Query Refinement

Phase 2 of *SnS* takes as input the bounding query Q^b computed in Phase 1. Q^b is guaranteed to be a superset of all possible refinements of Q. Therefore, *SnS* can invoke *SSE* with Q^b as the superset query to compute an ϵ-approximation E_{Q^b} which is utilized to estimate the cardinality of any query $Q' \subseteq Q^b$.

As illustrated in Examples 1 and 2, there might be multiple possible refinements of the original query that satisfy the target cardinality constraint. Therefore, *SnS* provides an interactive procedure which takes into account user feedback to refine the query as per one's preferences. There are two components of this interactive procedure:

- **Index structures for cardinality estimation:** *SnS* utilizes the in-memory set E_{Q^b} consisting of tuples produced by $SSE(Q^b)$ to compute cardinality estimates for each possible refinement Q' considered by the procedure. For efficiency purposes, we devise indexing schemes over E_{Q^b} which are tailored to the needs of our refinement framework.
- **Navigation Scheme:** Our goal is to provide a means for users to interactively refine queries. Therefore, we devise user navigation schemes which enable one to explore the search space for refinements Q' that best capture one's preferences.

We provide further details of our index structures and navigation schemes for queries with only numeric, only categorical and both numeric and categorical predicates in Section 5.

4. PHASE 1: COMPUTING BOUNDS

In Phase 1, *SnS* identifies whether the original query Q is to be relaxed or contracted, and computes maximal relaxations or contractions along all dimensions of Q. For this purpose, it generates the extended query Q^e, and invokes $SSE(Q^e)$ generating an in-memory ϵ-approximation E_{Q^e}. E_{Q^e} can be utilized to estimate the cardinality of Q, since $Q \subseteq Q^e$. Additionally, E_{Q^e} enables computation of the maximal relaxations and contractions of Q as described next.

4.1 Maximal Transformations (Numeric)

Consider a numeric predicate $P_i : x_i < C_i$. The *SnS* framework computes its maximal transformation $P_i^m : x_i < C_i^m$ through procedure *MaxTrans* illustrated as Algorithm 1. In order to compute P_i^m, the procedure requires as input the predicates along the remaining $d - 1$ dimensions of the original query, with dimension i set as unknown. *MaxTrans* performs a binary search between the lower (C_i^l) and upper (C_i^u) bounds of the domain of dimension i. For each value *val* considered, the procedure refines predicate P_i to $x_i < val$, and invokes the cardinality estimation component (encapsulated as *CardEst*) to obtain a cardinality estimate for the resulting query.

MaxTrans returns a value C_i^m such that Q with predicate P_i refined to $x_i < C_i^m$ best satisfies the target cardinality constraint. This value is then compared to the original value C_i of the predicate. If $C_i^m \geq C_i$, P_i^m is the maximal relaxation along dimension i; if $C_i^m \leq C_i$, the query is

Algorithm 1 Binary Search for Maximal Transformations

```
Var $D$: Database
Var $Q$: Query
Var $k$: Target

MaxTrans(Point p)
$i$ = UnknownDimension($p$).
min = $C_i^l$                    Set min, and max
max = $C_i^u$                    to domain boundaries
while (min ≤ max) do
   val = (min+max)/2
   $p[i]$ = val                  New value on dim i
   $Q'$ = GenQuery($p$).
   Est = CardEst($Q'$,$D$);     Est. card. of resulting query
   if Est < $k$ then
      min = val
   else if (Est > $k$) then
      max = val
   else
      return $C_i^m$ = val
   end if
end while                        End of loop for binary search
return $C_i^m$ = val
```

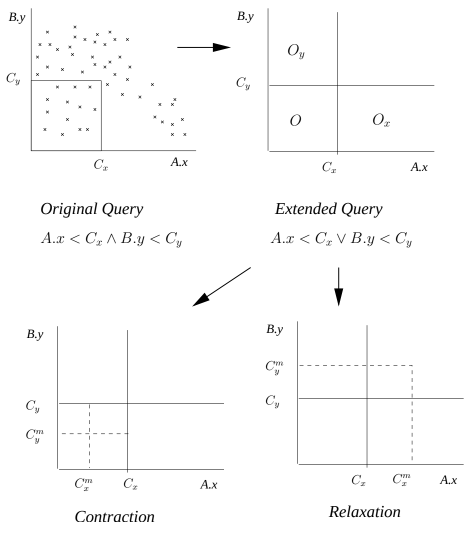

Figure 1: Maximal relaxations and contractions for numeric predicates

to be contracted, and P_i^m is the maximal contraction along dimension i.

The *CardEst* procedure utilizes the ϵ-approximation E_{Q^e} for cardinality estimation. Consider for instance the 2 predicate query shown in Figure 1 with predicates $A.x < C_x \wedge B.y < C_y$. The extended query has predicates $A.x < C_x \vee B.y < C_y$ which divide Q^e into three regions O, O_x and O_y as shown in Figure 1. Observe that one only requires the tuples of E_{Q^e} in the region $O + O_x$ in order to compute the maximal transformation C_x^m along $A.x$; likewise one can compute C_y^m using the region $O + O_y$. More generally, for a d dimensional query, *SnS* computes C_i^m by considering only the tuples in E_{Q^e} which satisfy the predicates along the remaining $d-1$ dimensions. These tuples are used to construct an exact histogram sorted along dimension i in memory. The *MaxTrans* procedure effectively performs a binary search on this histogram as accessed through the *CardEst* wrapper function.

4.2 Maximal Relaxations (Categorical)

Consider a hierarchical categorical predicate $P_i : x_{i1} = C_{i1} \wedge \ldots \wedge x_{il} = C_{il}$. As with numeric predicates, one can compute the maximal relaxation along dimension i by considering the tuples in E_{Q^e} which satisfy the predicates along all $d-1$ remaining dimensions. For example, given the extended query from Example 4, one only needs to consider tuples which satisfy the predicates `weight < 120 and age < 40`.

For each such tuple t , *SnS* computes the maximum level l_t such that t satisfies all predicates of P_i with level $\leq l_t$. Thus, a tuple that satisfies all levels of P_i has a $l_t = l$, while a tuple that fails to satisfy even $x_{i1} = C_{i1}$ has $l_t = 0$. For the hierarchical predicate considered in Example 4, $l_t = 2$ if the tuple satisfies `country = 'US' and state = 'FL'`; $l_t = 1$ if it satisfies only `country = 'US'`, with $l_t = 0$ otherwise.

SnS maintains a counter $N(i)$ for each level $0 \leq i \leq l$ of the hierarchical predicate. For each tuple t generated by $SSE(Q^e)$ which satisfies all remaining $d-1$ dimensions, *SnS* computes l_t and increments $N(i)$ for all $0 \leq i \leq l_t$. At the end of the *SSE* procedure, these counts are scaled up as per the sampling percentage. The level of the maximal relaxation is the level l_m such that $N(l_m) \geq k$ and either $l_m = l$ or $N(l_m+1) < k$. Accordingly $P_i^m : x_{i1} = C_{i1} \wedge \ldots \wedge x_{il^m} = C_{il^m}$ is the maximal relaxation along dimension i. If no such predicate is identified, *SnS* sets $l_m = 0$ i.e the resulting bounding query Q^b has no predicate on dimension i. For the example predicate, if $N(2) = 20K$, $N(1) = 60K$ and $N(0) = 200K$, and the target cardinality is $50K$, then the level of the maximal relaxation is 1 i.e the maximal relaxation is the predicate `country='US'`.

Given a query Q with numeric and/or categorical predicates, *SnS* simultaneously computes the maximal relaxations/contractions along all dimensions of Q with a single invocation of $SSE(Q^e)$ using the procedures outlined above. This generates the bounding query Q^b which is utilized in the second phase of our framework, as described next.

5. PHASE 2: QUERY REFINEMENT

Phase 1 of *SnS* returns a bounding query Q^b which bounds all solutions to the query refinement problem. This section describes indexing structures and navigation schemes for interactively exploring the search space defined by Q^b in order to generate refinements of the original query. We first describe these techniques for numeric predicates in Section 5.1 and for categorical predicates in Section 5.2 before combining the techniques in Section 5.3.

5.1 Numeric Predicates

In this section, we assume that the query has only numeric predicates of the form $x_i < C_i$. We first state certain properties of the space enclosed by Q^b, before describing an indexing structure which exploits these properties. We then describe the navigation scheme supported by *SnS* for numeric predicates, and illustrate how the index supports the scheme.

If the original query Q is to be relaxed, let variables $C_i^b = C_i^m$ and $C_i^s = C_i$. If it is to be contracted, let $C_i^b = C_i$ and $C_i^s = C_i^m$. Effectively, C_i^b corresponds to the predicate along dimension i ($x_i < C_i^b$) of the bounding query Q^b, while C_i^s corresponds to the smaller of C_i and C_i^m.

In the following we use the terms rectangle and hyperrectangle interchangeably. The bounding query Q^b corresponds to a d dimensional hyperrectangle $O^b : \forall_i C_i^l < x_i < C_i^b$

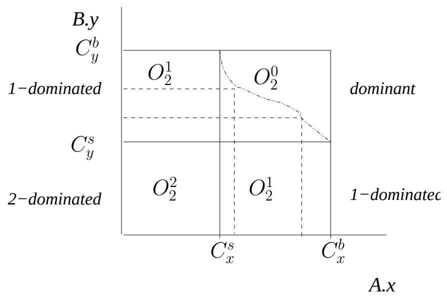

Figure 2: Query Refinement: Phase 2
Every point on the curve in O_2^0 dominates k points

where C_i^l is the lower bound of the domain of x_i.

Consider the d hyperplanes $\langle x_i = C_i^s \rangle$. Each hyperplane splits O^b into 2 halves, and therefore the hyperplanes along the d dimensions result in 2^d smaller rectangles. Out of these 2^d rectangles, one is of particular interest.

Definition 6. ***[Dominant rectangle]*** *(O_d^0): This is the rectangle enclosing the region $\forall_i : C_i^s < x_i < C_i^b$.*

Property 1. *Every solution to the query refinement problem is defined by a set of selection predicates corresponding to a point inside the dominant rectangle O_d^0.*

This property holds because C_i^s corresponds to the maximal contraction along a dimension.

Definition 7. *[**l-dominated rectangle**] (O_d^l) Any rectangle for which there exists l dimensions $L = \{j_1 \dots j_l\}$ such that for any point $x \in O_d^l$*

$$x \in O_d^l \rightarrow \forall_{i \in L} x_i < C_i^s$$

is an l-dominated rectangle.

Unlike the dominant rectangle, an l-dominated rectangle is not unique. For instance, there are 2 1-dominated rectangles in Figure 2.

Property 2. *Any point in O_d^0 dominates any point in O_d^l along at least l dimensions i.e is larger on l dimensions.*

We next describe our index structures, which exploit these properties to optimize space requirements.

5.1.1 Index Structure

Algorithm 2 QuadTree Insertion

```
QTInsert(Tree,tup)
if (Tree.isroot) N++
Tree.elements++
UpdateMinMax(Tree,tup);
if (Tree.isLeaf == false) then
    Child = ComputeChild(tup)
    QTInsert(Child,tup)
    if (Tree.elements < $\alpha N/2$) then
        Merge(Tree,Children)
    end if
else
    Insert(tup)
    if (Tree.elements > $\alpha N$) then
        Split(Tree)
    end if
end if
```

In Phase 2, *SnS* invokes $SSE(Q^b)$ to generate an ϵ-approximation E_{Q^b}. Our procedure constructs an in-memory quadtree on E_{Q^b} to support fast range counting. The quadtree structure is derived from the adaptive spatial partitioning tree introduced in [17]. Algorithm 2 describes our quadtree insertion algorithm *QTInsert*. *QTInsert* maintains a target fraction α, such that no leaf in the tree may contain more than an α fraction of the tuples seen (N). If a leaf contains more than αN tuples, it is split into 2^d children. Similarly, if the leaf descendents of an intermediate node together contain less than $\alpha N/2$ tuples, then the subtree at the node is collapsed into a leaf node. This can happen due to incoming tuples increasing N. At each node of the quadtree, *QTInsert* maintains the *min* and *max* values along each dimension over all the tuples in the leaves of the subtree at the node

The space requirements of the quadtree described above can be optimized further. Property 1 asserts that all solutions to the query refinement problem must lie within the dominant rectangle O_d^0. Given a d dimensional point p' corresponding to a query Q', *SnS* requires the quadtree index to return the number of tuples in E_{Q^b} that are dominated by p'. Property 2 asserts that p' must dominate every point p^l in O_d^l along l dimensions. Therefore, for any tuple t_{p^l} represented by a point p^l in O_d^l, one only requires the remaining $d - l$ dimensions to check whether p^l is dominated by p' (i.e whether $t_{p^l} \in Q'$). Since it is possible to discard the l dominated dimensions, we modify the quadtree index to exploit this property by initially splitting the root node of the tree along the d hyperplanes $\langle x_i = C_i^s \rangle$. This optimization results in significant space savings. For instance, the quadtree does not keep any attributes of tuples in the d-dominated rectangle O_d^d, only maintaining a counter for this node.

5.1.2 Navigation Scheme

We now describe the navigation scheme supported by the *SnS* framework. Each dimension i of the original query is initially associated with a range (C_i^s, C_i^b) defined by the original query predicate, and the associated maximal transformation. Our goal is to support an interactive refinement procedure which enables specification of one's preferences on the choice of values of the refined predicates, within the constraints defined by these ranges.

The interactive refinement procedure proceeds in the form of *rounds* between the user and the *SnS* framework. Each round consists of the following two steps:

- The user selects an arbitrary predicate $\mathcal{P}_j : x_j < C_j$ and refines it to a new predicate $\mathcal{P}'_j : x_j < C'_j$ such that $C_j^s \leq C'_j \leq C_j^b$ i.e C'_j lies within the range for dimension j.
- *SnS* recomputes the ranges (C_i^s, C_i^b) for each *unrefined dimension i* conditional on the user's current set of predicate refinements.

Each predicate refinement further constrains the ranges of the remaining unrefined predicates. This process is repeated for $d-1$ rounds, at which point the final unrefined predicate is fully constrained, and can be computed automatically. We illustrate this interactive refinement procedure through the following example:

Example 6. *Consider a query with predicates `year < 1960` and `age < 25` and `salary < 3000` and suppose it returns* *too few answers. In Phase 1, SnS computes maximal relaxations (*1980, 40, 7000*) for* `year, age, salary` *respectively. In Phase 2, one may first refine the predicate on* `age` *to* `age < 30`*. SnS in turn recomputes the ranges of the remaining predicates as* `year:` (1960, 1975) *and* `salary:`(3000, 6400)*. These ranges are smaller than the ranges specified by the initial set of maximal relaxations due to the relaxation of the predicate of* `age`*. Given these ranges, one may refine the predicate on* `year` *to* `year < 1970`*. SnS computes the best choice of the final predicate, relaxing it to* `salary < 4100`*. The final refined query has predicates* `year < 1970` *and* `age < 30` *and* `salary < 4100`*.*

Algorithm 3 Numeric Predicate Refinement

```
 1: INumRef()
 2: for all Dimensions $i$ do
 3:     $C_i^s = min(C_i, C_i^m)$; $C_i^b = max(C_i, C_i^m)$
 4:     $C_i' = C_i$
 5: end for
 6: if $\exists i : C_i^m = \infty$ then
 7:     RecomputeLowers()
 8: end if
 9: NumUnRef = $d$
10: while NumUnRef > 0 do
11:     GetUserRef()
12:     NumUnRef −−
13:     for all UnrefinedDimensions $i$ do
14:         $C_i^m = MaxTrans(C_1', \ldots, C_{i-1}', ?, C_{i+1}', \ldots, C_d')$
15:         if Contraction then $C_i^s = C_i^m$ else $C_i^b = C_i^m$
16:         if NumUnRef = 1 then $C_i' = C_i^m$; return;
17:     end for
18: end while

19: RecomputeLowers()
20: for all Dimensions $i$ do
21:     temp = $MaxTrans(C_1^b, \ldots, C_{i-1}^b, ?, C_{i+1}^b, \ldots, C_d^b)$
22:     if temp > $C_i^s$ then $C_i^s$ = temp;
23: end for
```

Algorithm 3 describes the numeric predicate refinement component (*INumRef*) of the *SnS* framework. *INumRef* initially defines a range (C_i^s, C_i^b) for each predicate (as per Property 1). Each predicate refinement made by the user (*GetUserRef*), results in a recomputation of the ranges for each unrefined predicate (lines 13-17). This is performed by calling the *MaxTrans* procedure (Algorithm 1), setting the unrefined dimension i as unknown ('?'), with all remaining dimensions j set to either the original $C_j' = C_j$ (if unrefined) or the refined C_j' values. When only one unrefined dimension i remains (line 16), *INumRef* automatically refines the associated predicate to its maximal transformation C_i^m.

The version of *MaxTrans* utilized by *INumRef* differs from the description in Algorithm 1 in two minor ways. First, it performs a binary search over the limited range (C_i^s, C_i^b). The second difference is that the associated cardinality estimation module *CardEst* now utilizes the quadtree index on E_{Q^b} through the function *QTQuery* outlined as Algorithm 4. Given a d dimensional point p, *QTQuery* computes the number of tuples in the quadtree dominated by p. *QTQuery* also uses the *min* and *max* values associated with each node of the quadtree to avoid unnecessary tree traversals.

Recomputing lower bounds for relaxations: The above discussion assumes that it suffices to recompute only the maximal transformation C_i^m in each round of the refinement process. While this holds true in general, for certain special cases of query relaxation the lower bound may be too tight to achieve the target cardinality due to highly selective predicates. This is indicated by maximal relaxations for

Algorithm 4 Index Querying Procedure

```
QTQuery(QuadTree Tree,Point p)
if (DominatesOnAllDims(p,Tree.max)) then
    return Tree.elements
else if (DominatesOnOneDim(Tree.min,Point p)) then
    return 0
end if
if (Tree.isLeaf) then
    return CountDominatedPoints(p,Tree.elementsArray)
else
    dom = 0
    for all (Child $\in$ Tree.Children) do
        dom+=QTQuery(Child,p)
    end for
end if
return dom
```

some dimensions being set to infinity. In this case, *INumRef* recomputes the lowerbounds by calling the *RecomputeLowers* procedure (Alg 3 lines 19-23). *RecomputeLowers* invokes *MaxTrans* for each dimension i, with all remaining dimensions j set to their maximal relaxations C_j^b. If the returned value exceeds the current lower bound C_i^s, then the lower bound is modified.

Error Guarantees: Our navigation scheme enables a refinement of all but one of the predicates as per user preferences. The final predicate is however refined by *INumRef*. Let the maximum frequency of a value in dimension i in the bounding query Q_b be m_i. If i is the final dimension refined by *SnS*, then in the worst case, the resulting refined query will have an absolute error of $\frac{m_i}{2}$ with respect to the target cardinality. This error is separate from the errors due to cardinality estimation, and is an artifact of our flexible navigation scheme. This also suggests that it is best to leave attributes with uniform distributions and many distinct values as the final unrefined dimension, since these would be expected to have a low value of m_i.

5.2 Hierarchical Categorical Predicates

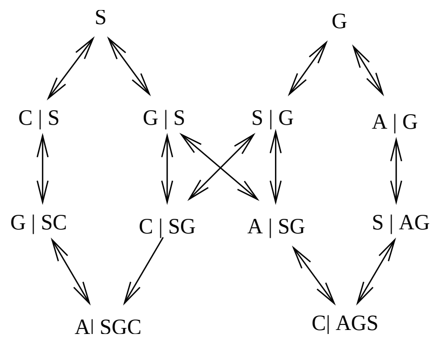

Figure 3: Navigation paths for 2 hierarchies (S,C) and (G,A)

In this section, we describe the indexing structures and navigation schemes supported by *SnS* for refining queries with hierarchical categorical predicates only. Since these predicates are equality predicates on categorical domains, one cannot utilize quadtrees and adopt a range shrinking navigation scheme as with numeric predicates. Instead, the *SnS* framework deploys techniques based on materializing different *navigation paths* on a data cube [15].

Consider a query Q with d hierarchical categorical dimensions. Q can be relaxed by *roll-up* and *expansion* operations applied in an arbitrary order on the upper levels of the categorical hierarchies. Likewise, Q can be contracted by *shrinking* and *drill-down* operations applied in an arbitrary order on the lower levels of the hierarchy. We refer to any feasible sequence of such relaxation or contraction operations for refining Q as a *navigation path* or *NavPath*. For example, given two two-level hierarchies (`State(S), City(C)`) and (`Genre(G), Artist(A)`), the possible *NavPaths* for relaxation/contraction are illustrated in Figure 3. In the figure, an entry along a navigation path (*NavEntry*) of the form $G|SC$ represents a relaxation/contraction operation on dimension G, with predicates on S and C unchanged, and no predicate on A.

5.2.1 Index Structure

Algorithm 5 Constructing CFDs

```
List NavPaths = CreateNavPaths()

HistInsert(Tuple Tup)
List HistEntries = NULL
for all NavEntry $Nav \in$ NavPaths do
    childAttr = ExtractChildAttr($Nav$,Tup)
    HistEntry parent = GetParent(Entries,$Nav$)
    if hasChild(parent,childAttr) then
        HistEntry child = GetChild(parent, childAttr);
    else
        HistEntry child = AddChild(parent,childAttr);
    end if
    child.freq++
    Entries.add(child)
end for
```

As with numeric predicates, *SnS* invokes $SSE(Q^b)$ in order to generate an in-memory ϵ-approximation E_{Q^b} for the purposes of cardinality estimation. The goal of our indexing structure is to support fast cardinality estimation for the operations of relaxing or contracting a hierarchical predicate by utilizing E_{Q^b}. *SnS* accomplishes this by maintaining conditional frequency distributions (CFDs) for each *NavEntry* along any possible *NavPath* for relaxation or contraction.

Algorithm 5 describes our procedure for constructing CFDs as exact histograms over E_{Q^b}. Given a query Q with d hierarchies, function *CreateNavPaths* constructs the possible *NavPaths* for query refinement. Having identified the navigation paths, *SnS* invokes function *HistInsert* for each tuple *Tup* produced by $SSE(Q^b)$. *HistInsert* traverses the *NavPaths* in a breadth-first top-down fashion. Each *NavEntry* encountered encodes a particular CFD; for instance $G|S$ represents the frequency distribution of `Genre` conditional on a given value of the `State` predicate. Given a *NavEntry*, *HistInsert* identifies the appropriate parent and child histogram entries (*HistEntry*) and increments the frequency of the child accordingly. For instance, given a tuple with `Genre='Pop'` and `State = 'Fl'`, for *NavEntry* $G|S$, *HistInsert* (a) identifies the *parent HistEntry* for `State = 'Fl'` (b) looks up its children for a *HistEntry* corresponding to `Genre='Pop'`, creating a new *child* if necessary and (c) increments the frequency count of the *child*.

5.2.2 Navigation Scheme

As with numeric predicates, *SnS* supports a navigation scheme for hierarchical categorical predicates which proceeds in the form of *rounds*. The process begins with all predicates as in the original query. In each round:

- The user selects an arbitrary hierarchical predicate, and either *rolls-up/expands* the predicate (in the case of *relaxation*) or *drills-down/shrinks* it (for *contractions*).
- *SnS* identifies the current position of the user along the *NavPaths*, and displays the appropriate set of CFDs for any possible next refinement step, utilizing the index structure.

This process continues until either *(a)* the refined query exceeds the target cardinality (for relaxations) or falls below the target cardinality (for contractions) or *(b)* one identifies a *final predicate*, for which *SnS* computes an appropriate relaxation or contraction to best satisfy the target cardinality. We illustrate this process through the following example featuring only drill-down operations:

Example 7. *Consider a query with initially no predicates which is to be contracted to a target cardinality of* $50K$ *tuples along the hierarchies* (S, C) *and* (G, S). *SnS initially presents frequency distributions on* `State` *e.g ('FL', 10M), ('CA', 7M),..., and* `Genre` *e.g ('rock', 5M), ('pop', 3M), ... One may then select* `State = 'FL'`. *In response, SnS presents frequency distributions on* `city` *e.g ('Miami', 800K), ('Tampa', 400K), ... and* `genre` *e.g. ('rock', 2M), ('pop', 1M), ... These distributions are conditional on the selection of* `State='FL'`. *This process is repeated, with one possibly selecting* `genre='Rock'` *and* `artist='Coldplay'`. *The final predicate for attribute* `city` *is computed by SnS as* `city='Miami' OR city='Tampa' OR city='Alachua'`.

Identifying the appropriate set of CFDs to display is straightforward At each round of the refinement process, the user is at some *NavEntry*, with the current set of predicate refinements corresponding to an associated *HistEntry hist*. Therefore for roll-ups, *SnS* needs to display frequencies of the *parents* of *hist*. Similarly, drill-downs require displaying frequency distributions of the *children* of *hist*. Likewise, expansion and shrinking require displaying the frequency distributions of the *siblings* of *hist* and of its *children* respectively.

We now describe how *SnS* refines the final predicate.

Refining the final predicates: Suppose the final predicate to be refined is on attribute x_i with possible values $C_i^1, \ldots, C_i^n$ with associated (through *HistEntries*) conditional frequency estimates $f_i^1, \ldots, f_i^n$ respectively. The goal of this step is to *disjunctively* select a subset $J = j_1, \ldots, j_r$ of these n values such that $\sum_{j \in J} f_i^j \approx k$. This is the subset-sum problem, which is known to be NP-hard. Although polynomial time approximation schemes exist for this problem [18, 25], we implement a greedy 2-optimal approximation algorithm [25] described as Algorithm 6. This approximation guarantee is a worst-case guarantee, and in practice the greedy algorithm works extremely well.

Procedure *FinalPred* (Algorithm 6) takes as input a list of possible values $C_i^1, \ldots, C_i^n$ for the final predicate sorted in decreasing order of their frequency f_i^j. *FinalPred* greedily adds new values to the current set (*CurrSet*) of values unless the associated sum *CurrSum* exceeds the target cardinality. The procedure ensures that *CurrSum* is guaranteed to be $\leq k$. Additionally, *FinalPred* maintains the greedy set *CurrBigSet* with minimal error which has a sum *CurrBigSum* $> k$. *FinalPred* returns either of *CurrSet* or *CurrBigSet* having minimum error.

To summarize, *SnS* supports a navigation scheme based on roll-ups/expansions or drill-downs/shrinking along mul-

Algorithm 6 Refining the Final Predicate

```
FinalPred(SortedList values)
CurrSet = NULL; CurrSum = 0; CurrBigSet = NULL; CurrBigSum = 0;
for all $(C_i^j, f_i^j) \in$ values do
  if CurrSum $+f_i^j \le k$ then
    CurrSet.add($C_i^j$)
    CurrSum $+= f_i^j$
  else if | CurrSum $+f_i^j - k$| < | CurrBigSum $-k$| < then
    CurrBigSet = CurrSet + $C_i^j$
    CurrBigSum = CurrSum $+f_i^j$.
  end if
end for
return minErr(CurrSet,CurrBigSet)
```

tiple navigation paths for a set of hierarchies. It supports these operations by maintaining conditional frequency distributions along all such possible navigation paths

5.3 Combining the techniques

In the previous two sections, we have described index structures and navigation schemes for queries with only numeric, and only hierarchical categorical predicates. In this section, we illustrate how these techniques can be combined for queries with both numeric and categorical predicates.

5.3.1 Index Structure

The goal of the index structure is to efficiently support cardinality estimation for range and equality queries over multidimensional categorical and numeric data. The index is built on the ϵ-approximation E_{Q^b} generated by the *SSE* procedure.

SnS combines the quadtree index (for numeric predicates) and navigation path index (for categorical hierarchies) by adding a quadtree to each histogram entry (*HistEntry*) to represent the numeric dimensions of the tuples corresponding to the *HistEntry*. This is accomplished by adding an extra function call *QTInsert(child.Tree,Tup)* after line 13 of the *HistInsert* procedure in Algorithm 5. Thus, for instance, the numeric attributes for all tuples with `state='FL'` are indexed by a quadtree associated with the corresponding *HistEntry*. Additionally, *SnS* maintains a global quadtree which indexes all tuples in the bounding query. This index structure suffices to provide fast cardinality estimates for *any* $Q' \subseteq Q^b$.

5.3.2 Navigation Scheme

The navigation scheme for queries with both numeric and categorical predicates remains essentially unchanged. As before, refinement proceeds in rounds. In each round:

- The user selects either a numeric or categorical predicate and refines it to a new value.
- *SnS* responds by updating the ranges for unrefined numeric predicates, and the appropriate CFDs for the categorical predicates.

Suppose a numeric predicate $P_i : x_i < C_i$ is refined to a new value C_i'. *SnS* recomputes the ranges of the remaining unrefined predicates by calling the *MaxTrans* procedure (Algorithm 1). For the categorical predicates, the relevant CFDs are updated by calling the *QTQuery* procedure (Algorithm 4) for the quadtrees associated with *each HistEntry*. Likewise, a similar procedure is adopted when a hierarchical categorical predicate is relaxed or contracted.

6. EVALUATION

In this section, we describe an implementation and experimental evaluation of our refinement framework

We have instantiated the *SSE* procedure in the Postgresql 8.0 database system, and implemented the *SnS* framework as a Java based frontend. *SnS* communicates with *SSE* through the JDBC layer and network sockets. Each query Q submitted for refinement results in two invocations of the *SSE* layer, once for the extended query Q^e, and once for the bounding query Q^b. Our instantiation of *SSE* utilizes precomputed random samples on base relations. In our evaluation, the *SSE* procedure is halted when it generates an ϵ-approximation of size 5000 tuples, with a sampling threshold of at most 5%. Varying the sample size (provided it isn't too low) did not have a significant affect on the cardinality estimates.

We conducted experiments on three different test databases. The primary test database is a 2.5 GB database generated using the DMV data generator [24]. The DMV dataset consists of 4 tables. The table sizes of *Owner (O)* and *Car (C)* are 2.5 million tuples; the size of *Demographics (D)* is approximately 3.6 million tuples; and the size of *Accidents (A)* is approximately 10.7 million tuples. The dataset was generated with the correlations flag set on. This produces many interesting correlations between columns on different tables. which makes cardinality estimation highly challenging. Additionally, we also performed experiments on 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 [9].

To simulate user interaction with the system, we implemented an external function which randomly refines predicates (within the constraints imposed by the framework) at each round of the refinement process. In our experiments we plot the relative error of the refined queries, which is defined as

$$Err = \frac{|RefinedQueryCard. - TargetCard.|}{TargetCard.}$$

The experimental evaluation was conducted on a lightly loaded machine running Suse Linux with 4 GB memory and 3.60GHz clock speed.

6.1 Accuracy Experiments

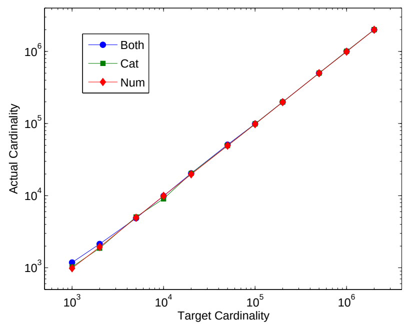

Figure 4: Single Query Varying Target

Figure 4 describes the results of an experiment using the

DMV database in which we fixed the original query, and varied the target cardinality. We generated 3 initial queries. The query marked as `Num` is a 3 table join query with 3 numeric range predicates, and original cardinality of approximately 600K. The query marked as `Cat` is a 3 table join query with 2 categorical predicates, each a part of a 3 level hierarchy. Its original cardinality is approximately 250K. The query marked as `Both` is a 3 table join query with 3 numeric and 2 categorical predicates and original cardinality approximately 125K. We varied the target cardinality from 1K (low selectivity) to 2M (high selectivity) tuples and plot the cardinality of the queries generated by our system with respect to the target cardinality. As can be seen, our technique generates queries that approximately satisfy the target cardinality constraints for a wide range of target and initial query cardinalities.

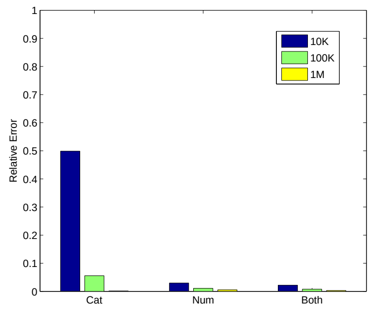

Figure 5: Multiple Query Varying Targets

In our next experiment, we generated three random workloads of 50 queries each defined on the DMV database having numeric (`Num`), categorical (`Cat`), and both numeric and categorical (`Both`) predicates respectively. Each query in each workload is refined to target cardinalities of 10K, 100K and 1M tuples. We plot the average errors for each workload-target cardinality combination in Figure 5. As can be seen, the average errors are low except for the case of queries with categorical predicates only, and a low target cardinality (10K). This is primarily due to the underlying data distribution which prevents contraction to the target cardinality. For instance, consider a query with predicates `make = 'Porsche' and country = 'GM'`, and cardinality 186K tuples, which is to be contracted to 10K tuples. Given two categorical hierarchies (`make, model, color`) and (`country, state, city`), the best that *any query refinement algorithm* can do on the underlying DMV database is to contract the query to 40K tuples (`model = 'Carrera' and color = 'red' and city = 'Berlin' and state = 'Berlin'`). Further contraction of this query is not possible.

In our next experiment, we examine the effects of the size of the domains on which categorical predicates are defined. For the experiment shown in Figure 6, we generated 50 queries with 3 categorical predicates defined on domains with 4, 20 and 197 distinct values on the DMV database. Each query in the workload has a cardinality less than 100K. We invoke *SnS* to relax each query to target cardinalities: 100K, 200K and 300K tuples. We consider each predicate to form a single level hierarchy, resulting in $3! = 6$ possible navigation paths for query relaxation. Each query is relaxed by rolling up along every possible navigation path, with the

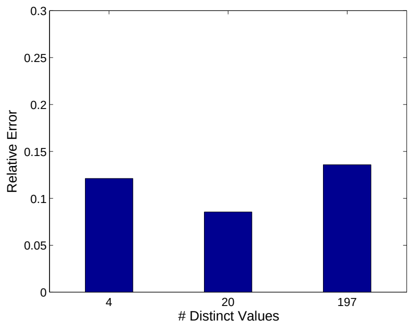

Figure 6: Varying # Distinct Values for relaxation

final predicate selected using the subset sum procedure *FinalPred* (Algorithm 6). In Figure 6, we plot the average error with respect to the domain size of the final predicate.

As can be seen from Figure 6, the average errors for small (4) and large (197) sized domains are higher. For small domains, this higher error is due to the fact that disjunctively adding or removing an additional predicate can significantly change the query cardinality. This leads to a coarse degree of control over the cardinality of the refined query. At the other extreme, for large domains, the *FinalPred* procedure has a much finer degree of control on query cardinality. However attributes with a large number of distinct values often have many low frequency values, for which cardinality estimation may be difficult. This illustrates an interesting tradeoff for categorical domains, in which we need to balance the finer degree of control provided by large domains, with the associated higher relative errors in cardinality estimates for the many low frequency values present. We note that a similar tradeoff does not arise in numeric predicates, since one does not require cardinality estimates for each distinct value within a range, but for the range as whole.

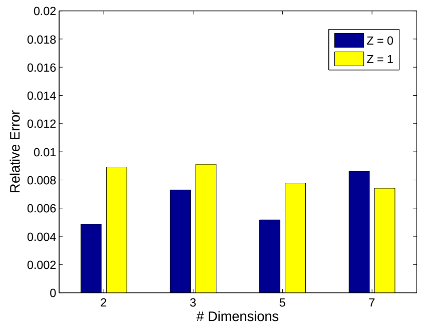

Figure 7: Varying dimensions and skew

In our next experiment illustrated in Figure 7, we examine the effects of varying the number of predicates (dimensions), and the skew of the underlying data on the accuracy of our refinement procedures for numeric predicates. We generate workloads of 5 table joins defined on TPCH tables, with 2, 3, 5 and 7 numeric predicates, with each workload having 50 queries. The target cardinality was fixed at 100K tuples. We plot the average error for each workload over both the uniform (Z=0) and skewed (Z=1) data. As can be seen, the errors are uniformly low ($< 1\%$) and are not significantly affected by the number of dimensions or skew of the data.

6.2 Overheads

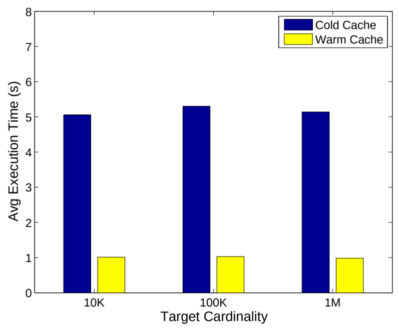

Figure 8: Execution Times

We measure the execution times of our technique by generating a workload of 10 queries on the DMV database, with each query having 3 numeric and 2 categorical hierarchical predicates. Each query was refined to 10K, 100K and 1M tuples. We measure the execution times on cold and warm caches. The execution time is the time to complete the entire refinement procedure, from the initial query specification to the final generation of the refined query. Figure 8 demonstrates that the average execution times are low (approx. 5s on cold caches) and independent of the target cardinality. This execution time is primarily concentrated on the execution of the *SSE* procedure. Once the *SSE* procedure instantiates the indexing structures, each round of the refinement process takes few ms to execute, demonstrating that our framework can support an interactive refinement interface with low response times.

7. CONCLUSIONS

In this paper, we have introduced a new model for solving the *many/few answers* problem. We have presented an interactive query refinement framework, and outlined the challenges, and our solutions for practically realizing such a framework. Our experimental evaluation of an implementation of this framework in a real database system demonstrates the utility of our approach.

8. REFERENCES

[1] S. Agrawal, S. Chaudhuri, G. Das, and A. Gionis. Automated ranking of database query results. *CIDR*, 2003.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. *ICDE*, 2001.

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

[4] M. J. Carey and D. Kossmann. On saying ”enough already!” in sql. *SIGMOD*, pages 219–230, 1997.

[5] S. Chaudhuri. Generalization and a framework for query modification. *ICDE*, 1990.

[6] S. Chaudhuri, N. N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. *ICDE*, 2006.

[7] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. *VLDB*, 2004.

[8] S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling Over Joins. *SIGMOD*, 1999.

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

[10] W. W. Chu and Q. Chen. A structured approach for cooperative query answering. *TKDE*, 1994.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.

[12] M. Fontoura, V. Josifovski, R. Kumar, C. Olston, A. Tomkins, and S. Vassilvitskii. Relaxation in text search using taxonomies. *VLDB*, 2008.

[13] P. Godfrey. Skyline cardinality for relational processing. *FoIKS*, 2004.

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

[15] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. *SIGMOD*, 1996.

[16] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *SoCG*, 1986.

[17] J. Hershberger, N. Shrivastava, S. Suri, and C. D. Tóth. Adaptive spatial partitioning for multidimensional data streams. *Algorithmica*, 46(1), 2006.

[18] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, 1975.

[19] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB*, 2003.

[20] Y. Ioannidis and S. Christodoulakis. Optimal Histograms for Limiting Worst-Case Error Propagation in the Size of Join Results. *ACM Transactions on Database Systems, Vol. 18, No. 4*, pages 709–748, Dec. 1993.

[21] A. Kadlag, A. V. Wanjari, J. Freire, and J. R. Haritsa. Supporting exploratory queries in databases. *DASFAA*, 2004.

[22] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. *VLDB*, 2006.

[23] G. Luo. Efficient detection of empty-result queries. *VLDB*, 2006.

[24] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust Query Processing Through Progressive Optimization. *SIGMOD*, 2004.

[25] S. Martello and P. Toth. Worst-case analysis of greedy algorithms for the subset-sum problem. *Math. Programming*, 28(2), 1984.

[26] C. Mishra, N. Koudas, and C. Zuzarte. Generating targeted queries for database testing. *SIGMOD*, 2008.

[27] V. N. Vapnik and A. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280, 1971.