Stretch ‘n’ Shrink: Resizing Queries to User Preferences

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

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

ABSTRACT

We present *Stretch ‘n’ Shrink*, a query design framework that explicitly takes into account user preferences about the desired answer size, and subsequently modifies the query with user feedback to meet this target. Our system has been prototyped inside an open source data manager, and requires minimal modifications to the database engine.

Categories and Subject Descriptors

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

General Terms

Design, Management

Keywords

Query Relaxation/Contraction, Many/Few Answers Problem

1. INTRODUCTION

There are many database applications with cardinality constraints on the the answer size. For example, one might like the number of results returned by the query to be manually browsable. Alternatively, in a decision support setting, an analyst might submit queries to retrieve data which is to be processed by an external program. In this case, one would like the number of answers to be large enough for meaningful analysis, while at the same time ensuring that the external program is not overwhelmed by too much data. As a third example, testing database applications for scalability across input sizes requires generation of queries that return a specified number of tuples. These three examples have the common feature of having a target *result cardinality* requirement that is less flexible than the actual specification of the query.

The presence of cardinality constraints on the result size is at odds with the standard boolean retrieval model supported by database systems in which constraints are specified on the properties of individual result tuples and not on the result table as a whole. This leads to the well known *many/few answers* problems [2, 5]. Support in current systems for controlling the answer size takes the form of the `STOP AFTER` or `LIMIT` clauses. This approach works only if the original query returns more tuples than necessary. Typically, this clause is used in conjunction with an `ORDER BY` clause which defines a scoring function on the data.

Copyright is held by the author/owner(s).
SIGMOD’08, June 9–12, 2008, Vancouver, BC, Canada.
ACM 978-1-60558-102-6/08/06.

The problem with adopting a scoring/ranking approach towards solving the many/few answers problem is that one needs to define a meaningful scoring function that best captures user preferences. This may be particularly difficult, especially over multiple attributes that are semantically different. For example, if one wants to get a list of old cars which were made before a particular `year` 1970, with a `cost` less than 5000, a *Top-k* based approach [3] would require a scoring function which (semantically speaking), combines time (`year`) and money (`cost`). Even if such a scoring function is defined, it reduces user preferences to a single value, and therefore provides at best indirect control over the final set of results returned.

We describe a system that enables *refinement* of the original query to a new query that satisfies the cardinality constraints on the answer size. The refinement takes the form of modifications to the *range selection predicates* in the query. In particular, our system (SNS) supports the operations of relaxing (*Stretching*) and contracting (*Shrinking*) the ranges defined by the selection predicates. Our techniques can also extend to categorical equality predicates provided that meaningful notions of relaxation/contraction are defined for such predicates. One of the interesting features of this model is that there might be many possible refined queries that satisfy the result cardinality constraint. Consider the following example illustrated in Figure 1 as well:

EXAMPLE 1. *Using the cars query as an example, suppose the original query with the predicates* `year < 1970` *and* `cost < 5000` *returns too few answers. The query could be refined by changing the condition on* `year` *to* `year < 1980`*. Alternatively we could change the condition on* `cost` *to* `cost < 7000`*. Or, both the conditions could be modified together to* `year < 1975` *and* `cost < 6500`*. These three queries (and many others) are all valid refinements of the original query*

2. DESIGN REQUIREMENTS

The problem of generating a query with a given result size is known to be NP-hard [1]. As a result, we relax the semantics of our solution, and return queries that *approximately* satisfy the target cardinality constraint. Our techniques have well defined error guarantees, and are designed to be *interactive* and *flexible*

Interactivity: SNS is proposed as an interactive query design tool. This requires it to have a low response time. To this end, we make heavy use of sampling based cardinality estimation techniques using precomputed random samples and indices. Although this induces an error in the cardinality estimates, the goal of our system is to *approximately* satisfy the target cardinality, while being as flexible as possible.

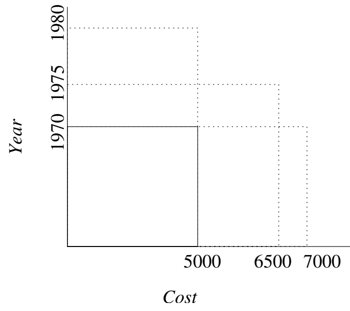

Figure 1: Each of the dashed rectangles represents a valid refinement of the original query

Flexibility: As illustrated in Example 1, a salient feature of this problem is that there might be several possible valid refinements of the original query that approximately satisfy the target cardinality constraint. In order to avoid imposing any restrictions on the choice of the refined query, we design techniques that are *complete* in the sense that they compute all possible refinements of the original query that approximately satisfies the cardinality constraints. We facilitate *interactive exploration* of the space of solutions by casting query refinement as an iterative search-space pruning problem which keeps the user "in the loop".

We now provide an overview of our approach, and illustrate how these requirements of interactivity and flexibility are satisfied.

3. INTERFACE

The SNS system refines queries to meet the user defined cardinality constraint by modifying the range predicates of the given query. To facilitate interactive refinement, we present bounds on how much each predicate can be stretched or shrunk. These bounds are defined by considering each predicate in isolation, and estimating how much it is to be refined to meet the target cardinality constraint. For each predicate $x < C$, this bound is a value C^m such that if we modify the query on *this predicate only*, replacing $x < C$ with $x < C^m$, the resulting query satisfies the target cardinality constraint. We refer to this bound as a *maximal transformation* for the predicate. For Example 1, this corresponds to *year* < *1980* and *cost* < *7000*. Maximal transformations can likewise be defined for the case of too many answers (i.e predicate shrinking) as well. Together with the original query predicates, they bound the space of possible refinements to the original query.

We propose an interactive query refinement procedure that centers on the notion of maximal transformations of the predicates of a query. The original predicate and maximal transformation define the range of possible values to which the query can be refined. At each step of the procedure, a user *refines* a predicate to some value within this range. *Conditional on the user's refinement*, the SNS system recomputes maximal transformations on the remaining unrefined predicates. This process is repeated until all but one of the predicates are refined, after which the system refines the final predicate.

Figure 2 illustrates the SNS system in action. We have submitted a 2 way join query with 3 selection predicates and a target cardinality of 50K tuples. The left panel shows this information along with an estimate of the answer size of the original query. The right panel contains our query refinement interface. For each predicate of the original query, we display a slider showing the range of values to which the predicate may be refined. The user may select an arbitrary slider and set it to a value within the range. This collapses the range to a single point, and also triggers recomputation of the ranges of the unrefined predicates. For instance, in Figure 2, we have selected and fixed a value for the predicate on *d.age*. The predicates on *d.salary* and *c.year* have still not been set, and one can see the ranges for the predicates illustrated by the `Max` and `Min` labels. In the next step, one can select either of these predicates and freeze their value between the lower and upper bounds. The system then determines the value of the final predicate. Alternatively, one can choose to undo the current choice for *d.age*, in which case the system resets the maximal transformations on each predicate to their original values. This interface clearly captures our stated design goal of the system being flexible. We introduce the concept of maximal transformations as a *guide* towards refining the query appropriately. However, to make this system fast and interactive, a number of technical challenges need to be addressed, which we describe in the next section.

4. TECHNOLOGY

The interface described in Section 3 requires us to first compute maximal transformations for each range predicate of the original query. Then, each time a predicate is refined by the user, we recompute maximal transformations for the remaining unrefined predicates. These are the primary operations required by our interface, and we now describe our techniques to make them fast. Due to space limitations, the description of our techniques is necessarily limited.

We utilize sampling based join cardinality estimation techniques [4] to compute the maximal transformations for the original query. Given a query with d range predicates, we generate a query that returns tuples that satisfy *at least* $d - 1$ of the range predicates. We execute this query in a pipelined fashion with a precomputed sample of one of the relations as the outer. This execution proceeds until we have a sufficiently large join result for the purposes of cardinality estimation. We utilize this join result to estimate the maximal transformation for each predicate of the query. This estimation procedure is performed using a binary search on the domain of each predicate, varying the value of the predicate while keeping all other predicates fixed to their original values. This procedure enables us to return maximal transformations for all d predicates, while sampling *only once* from the disk.

Once the original maximal transformations are computed, the interactive query refinement phase proceeds, and we recompute maximal transformations for each predicate. We observe that that each refinement of a predicate *further constrains* the ranges of the other unrefined predicates. Therefore, we generate a query in which each predicate is set to its maximal transformation, and perform a join sampling procedure on this query using a random sample of one relation with indices on the other relations. This sampling procedure is performed *only once* after the maximal transformations of the original query have been computed. We utilize the result of this join sampling procedure to recompute maximal transformations by performing a binary search on the current range of values each predicate can be refined to. To speed up this step, we store the result of the join sampling procedure in an in-memory quadtree index. Each subsequent refinement to a predicate made by the user results in accesses to this quadtree index. However, this is a small in-memory data structure containing the result of our sampling operation, and can therefore be utilized without slowing the system down.

Effectively, we sample from the base tables only twice: once to compute the original maximal transformations, and once to populate the quadtree which is used in the interactive refinement procedure.

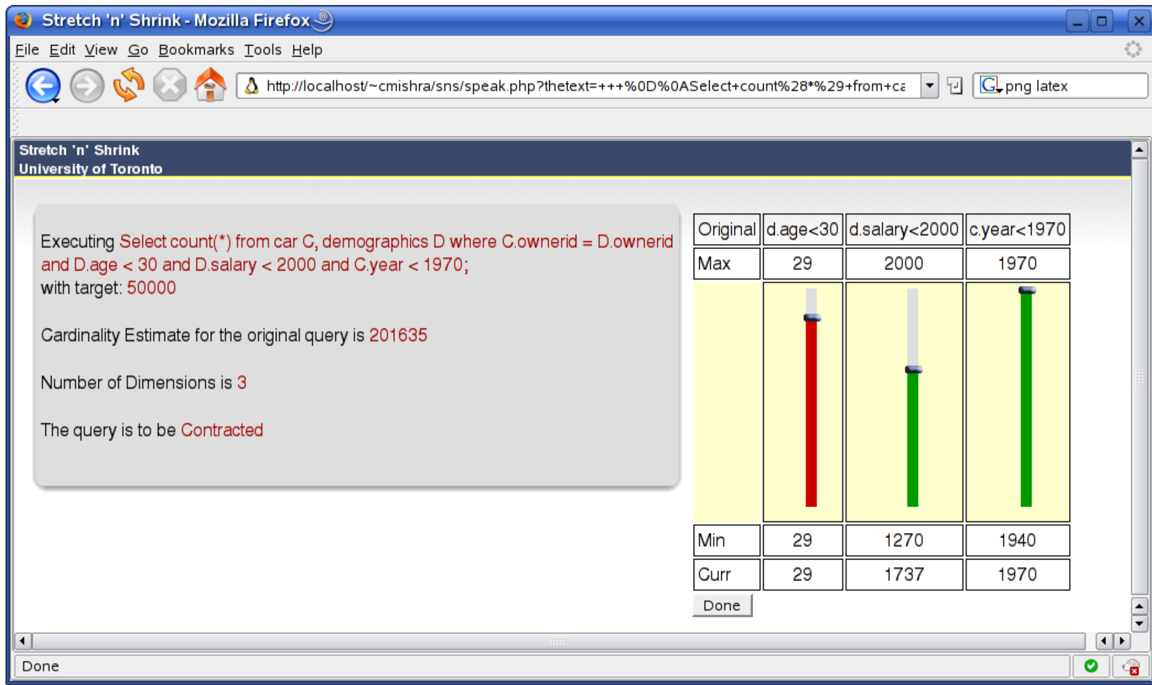

Figure 2: Stretch ‘n’ Shrink in action

5. SYSTEM ARCHITECTURE

The SNS prototype is built as a client server model with the backend as a modified database engine, and a web based frontend.

5.1 Backend

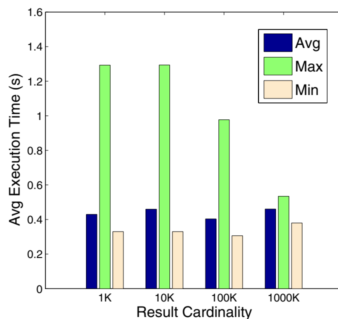

Figure 3: Execution Times

The SNS backend has been prototyped inside the Postgresql 8.0 database engine. We modified the engine to compute maximal transformations for the original query using sampling. The system then runs the sampling procedure to obtain an evaluation layer to be used by the frontend for cardinality estimation. The evaluation layer for cardinality estimation is stored using a quadtree for fast range counting. In addition to the standard interface provided by the database connectivity API, the backend and frontend also communicate through network sockets. Figure 3 describes the Avg, Min and Max execution times of our system for a workload of 50 2-way join queries on tables having approximately 3.6 and 2.5 million tuples. These execution times refer to the time the system takes to reach the interactive query refinement stage. Each step of the refinement with user feedback takes even less time.

5.2 Frontend

The SNS frontend has been constructed as a web interface employing AJAX technology to asynchronously communicate with the backend. The system first asks for the query, and the desired target cardinality. This information is sent to the backend which then constructs the evaluation layer. The backend replies with the maximal transformations which launches the query refinement interface illustrated in Figure 2.

6. DEMONSTRATION DESCRIPTION

In the demonstration, we intend to run a live version of our system with standard test databases loaded in the backend. The demonstration will highlight challenging query instances on databases with skew and inter-table correlations. Participants will be given an opportunity to submit arbitrary SQL queries to the system, and will be provided with further details of the technology.

7. REFERENCES

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

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

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

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

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