Query-oriented Relaxation for Cardinality Assurance

Manasi Vartak
Department of Computer Science, Worcester Polytechnic Institute, Massachusetts, USA
mvartak@wpi.edu

ABSTRACT

Although a large number of queries used in applications ranging from web search to business intelligence have associated cardinality constraints, current database engines have minimal support for ensuring query cardinality. This leads to two main types of problems: the empty result-set problems and the too few/too many problems. Inability to meet query cardinality constraints requires the user to undertake a frustrating trial-and-error process that can be extremely cumbersome and time-consuming. In fact this process wastes system resources without any guarantee of success. A possible strategy for solving the cardinality assurance problem is *query relaxation*. However, obtaining exact cardinality assurance with query relaxation has been proven to be NP-Hard. In this work we present *QRelX*, a novel algorithm for cardinality assurance using query relaxation. QRelX not only relaxes queries efficiently but also ensures that result queries have minimal relaxation. Our algorithm uses QXForm, a query space transformation framework, to enable the relaxation of join and select queries. Following this transformation, QRelX uses layer-based navigation and incremental cardinality estimation to find result queries that meet the expect cardinality but are also close to the original query. Our preliminary experimental results indicate that QRelX is computationally more efficient than traditional cardinality assurance algorithms and it indeed successfully finds minimal relaxation queries.

1. INTRODUCTION

A large number of database queries used every day have implicit or explicit cardinality constraints. For instance, consumer-oriented applications like web search, shopping or travel often have implicit cardinality constraints. For a satisfactory user experience a travel application should not return only one result and similarly a shopping application should not return five thousand results. Explicit cardinality constraints on the other hand are found in domains such as information retrieval, business intelligence and others. In these domains, satisfying the query cardinality can be as essential as answering the query itself. For instance, in order to run a medical study, a research agency may need a fixed number of volunteers satisfying certain health-related criteria. Likewise, a supermarket may need to send out surveys to a fixed number of customers having particular shopping trends. However, in spite of the importance of cardinality constraints, state-of-the-art database systems have minimal support for cardinality assurance.

The lack of query support for cardinality assurance leads to two common types of problems: empty result-set problems [15] and few/many problems [18]. The former type arises from queries that return no results (also called failing queries [20]) while the latter, a superset of the former, arises from queries whose result size is too large or too small for the user's purpose. In both cases the user is offered neither an explanation for the inconsistency in cardinality nor any suggestions for remedying it. The burden of formulating precise queries attaining the required cardinality falls on the user. However, this can be a difficult task because the user seldom has knowledge about the underlying database. Instead he or she has to resort to a frustrating trial-and-error process. Moreover, although there is no guarantee that this process will produce appropriate results, the user needs to try potentially numerous queries. This repeated query execution leads not only to a large increase in the query's response time but also a decrease in server throughput.

Q1 = SELECT * FROM People
WHERE
(25 < age < 75) AND
(BMI > 30) AND
(familyHist = 0) AND
(income < 55000)

Expected Cardinality = 2000

(a) Query Q1

Q2 = SELECT * FROM Flight, Hotel
WHERE
(Flight.price +
7*Hotel.price < 1500) AND
(beachDist < 5)

Expected Cardinality = 20

(b) Query Q2

Age	BMI	familyHist	Income
25<age<70	28	0	65000
25<age<80	27	1	70000
22<age<76	30	0	75000
21<age<77	26	1	65000
24<age<78	29	0	60000

(c) Q1 Alternative Queries

Total Price	beachDist
1800	5.5
1500	6
1900	5.2
1600	7.1
1570	6.5

(d) Q2 Alternative Queries

Figure 1: (a) Q_1 Medical study query selecting volunteers for an obesity study. (b) Q_2 Travel reservation query with constraints on price and distance to the beach

Consider for example Query Q_1 shown in Figure1.a. A medical research organization has received a grant to run an obesity study involving 2000 participants. To determine the connection between obesity and low income, the research agency wants to study peo-

ple with age between 25 and 50 years, BMI (body mass index) greater than 30, having no family history of obesity and having income less than $55000. However, on running this query against the database only 1300 people are found to meet the required criteria. Since the agency has resources to study 2000 people, 700 additional people must be selected for the study. With no additional information about the shortage of results or alternative queries to fix it, the user has to resort to trial-and-error. While formulating the modified query, the user can try various queries. For example, the age range can be expanded to 25 - 70 years, the upper bound on the income can be changed to $65000, etc. Figure 1.c shows some other alternate queries that a user may try. Each modification seeks to enlarge or *relax* the original query to include more results. Even for a simple query like Q_1 with four select predicates we see that there are a very large number (this number is in fact exponential with respect to the query predicates [12]) of alternate queries a user can enter. Continually executing these queries increases query response time and wastes server resources.

Query Q_2 in Figure 1.b gives another example for a query with cardinality constraints. In this query, the user is planning a week-long trip and wants to find a Flight+Hotel deal such that the hotel is close to the beach but the entire trip is not very expensive. Although the user's definitions of "close to the beach" and "expensive" are flexible, the travel database requires the user to set strict parameters. Q_2 shows one such set of parameters. For this example, the cardinality constraint is implicit, so we assume that the user expects about 20 results for this query. However, suppose the user only gets 10 results. The user now has to undertake the cumbersome trial-and-error to formulate an appropriate query.

For both queries Q_1 and Q_2 discussed above, a better - i.e. less frustrating, faster and meaningful - user experience can be obtained by directly providing the user with *alternate queries*. The alternate queries should not only attain the required cardinality but also be as close as possible to the initial query and hence to the original query semantics. Closeness to the original query is an important factor because a user will always prefer alternate queries that have little change from the original one. By automating the query relaxation process, we can improve the user experience, increase user satisfaction, decrease query response time and also save server resources. Towards this goal, this paper presents *QRelX* an incremental query relaxation framework that assures query cardinality while minimizing relaxation.

Before proceeding to the details of the algorithm, we note that while the above scenarios involving Q_1 and Q_2 discuss the case where a query returns *too few* results, similar arguments can be made for the case where a query returns *too many* results. If, for instance, the medical study query Q_1 returns too many results, then the user has to formulate a smaller query by shrinking Q_1's predicates.

Automated Query Relaxation: Query relaxation is the process of selectively loosening a set of query predicates to alter query cardinality [8]. The aim is to carefully control relaxation to provide the user with alternate queries that not only meet the required cardinality but also minimize relaxation. However, employing the technique of query relaxation poses several challenges: (1) the number of combinations of query predicates that can be relaxed is exponential in the number of predicates [12]; (2) the degree of relaxation required is not known beforehand; and (3) the process is computationally expensive since repeated query execution is required to explore the space of relaxed queries. In [3] Bruno et al. discuss the complexity of solving the cardinality assurance problem exactly and prove that the problem is NP-hard. An additional challenge in producing alternate queries lies in minimizing the *relaxation* or amount of change.

Various approaches for tackling the problems of cardinality assurance and query relaxation can be found in the literature. [8, 12, 20, 21] propose methods based on skylines, artificial intelligence and deduction to relax empty result-set queries. However, these techniques mainly focus on solving the empty result-set problem and not on meeting query cardinality. Mishra et al. propose an interactive framework for query refinement in [18]. This method uses information about the underlying database to help users to narrow down the choices for modified queries. However, the authors do not address the problem of relaxing join predicates. Cardinality assurance has also been studied in the content of generating test queries for databases in [3, 19]. The limitation of these methods is not only that they don't relax join predicates but also that they do not focus on minimizing relaxation since query semantics are not relevant for database testing queries. Top-k algorithms have traditionally been used to solve cardinality assurance problems by providing the user with the required number of tuples [6, 7, 9, 10, 14]. The STOP AFTER operator also functions similarly in the case where there are too many results [4]. However, these methods require the formulation of a specialized ranking function and donŠt generate a query characterizing how the desired results were obtained. Further, top-k can also lead to a biased distribution of data in the query results.

Tuple-oriented vs. Query oriented approaches: We classify the above methods like top-k [6] and skyline-based relaxation [12] as *tuple-oriented* methods for query relaxation since their goal is to return the given number of tuples without being concerned about the query that produces them. Methods for generating test queries (like [3]) on the other hand are *query-oriented* because they are concerned with the actual queries that produce the required number of results. However, the utility of the query-oriented approach is not limited to testing. In many queries like Q_1 and Q_2 shown above, providing the user with only the required number of tuples does not suffice. The user also needs to know *why the tuples were selected*, i.e., the **query** that generated them.

Let us consider query Q_1 to contrast the tuple-oriented and query-oriented approaches. First, tuple-oriented methods require much effort to ensure that the algorithm returns tuples that make sense for the given query. For instance, a poorly designed ranking function for top-k or the use of skyline algorithms on Q_1 may give results containing people with BMI less than 30 (and hence not considered obese) or with incomes over $65000 (throwing off the income factor). The query-oriented method on the other hand can assure that all the tuples produced by the given relaxed query will satisfy a fixed set of criteria. Query-oriented methods can also capture user preferences by providing the user with a set of alternate queries and allowing him or her to pick the one that best matches the preferences. Additionally, knowing the query that produced a given set of results is important in scenarios where a user may want to go back and generate a slightly different data set by modifying the query. The second difference between the two approaches relates to queries where we need to ensure that the result tuples are picked uniformly using a consistent set of criteria. Tuple-oriented approaches can cause uncontrolled departure from the original query and lead to the creation of a biased result set. For example, suppose that 700 "closest" neighbors are generated for Q_1 and we find

that the volunteers now satisfy the criteria: age between 25 and 70, BMI greater than 30, having no family history of obesity, and income less than $65000. However, these 700 additional people do not accurately represent the group of people with age 50 to 70, BMI over 30, with no family history and with income between $55000 and $65000. A query-oriented method however returns all the tuples satisfying the given criteria and hence accurately represents the underlying population. The last drawback of using tuple-oriented methods is that the non-uniform selection of result tuples can lead to unreliable results. In the context of Q_1, without a representative sample of people satisfying the given criteria, the conclusions of the study will have low confidence. Lastly, if such a study were to be repeated with a different data set, it is likely that the results may not hold because a biased sample was used in the initial study.

Thus, the advantage of the query-oriented approach over the tuple-oriented one is that it can provide an explanation for the selection of results, preserve the original data distribution and facilitate the process of future query refinement.

Our Contributions: The above discussion shows that while tuple-oriented methods are appropriate for certain applications, a query-oriented approach with its finer grained control over relaxation and thus ability to return more meaningful results is a critical technique needed by many classes of applications. In this work we propose to tackle the *query-oriented* cardinality assurance problem. Given an initial user query, QRelX relaxes the query so that it not only meets the expected cardinality but also has the least change from the initial query. QRelX is based on the key ideas of query space transformation, layer-based navigation and incremental cardinality estimation. Our query space transformation framework enables the relaxation of select as well as join queries. Layer-based navigation is deployed to minimize relaxation, and incremental cardinality estimation reduces computational expenses associated with relaxation. The above principles enable our algorithm to attain the following four goals: (1) Given a query Q^I (initial query) and an expected cardinality C_0, QRelX relaxes the query Q^I to Q^F (final query) such that Q^F satisfies the given cardinality; (2) Q^F minimizes the relaxation with respect to Q^I; (3) QRelX can relax select as well as join predicates; and (4) The process of relaxation from Q^I to Q^F is computationally efficient. By attaining all the four goals for both the empty result-set and few/many problems, we present a comprehensive solution to the query-oriented cardinality assurance problem.

To summarize, our main contributions are:

- We formally define the query-oriented cardinality assurance problem that minimizes query relaxation. We also introduce the classification of the relaxation algorithms into tuple-oriented and query-oriented algorithms.
- We present QXForm, a framework for transforming the initial query space into a relaxation space to relax select and join queries. We design special relaxation mapping functions for this purpose.
- We present QRelX - a novel algorithm for query relaxation that uses layer-based navigation and incremental cardinality estimation for efficient query relaxation. To the best of our knowledge QRelX is the first algorithm proposed for query-oriented join relaxation that ensures query cardinality.
- We propose a novel incremental cardinality estimation algorithm that delays tuple-level computations until absolutely necessary and that removes the need to repeatedly re-evaluate tuples for multiple queries. This reduction in tuple-level computations increases the efficiency of our relaxation technique.
- We present results from experimental studies comparing our method to state-of-the-art methods for cardinality assurance.

Predicate	pFunction	pInterval
(25 < age < 75)	age	(25, 75)
(BMI > 30)	BMI	(30, ∞)
(familyHist = 0)	familyHist	(0, 0)
(income < 55000)	income	(0, 55000)
(Flight.price +) (7 * Hotel.price < 1700)	(Flight.price +) (7 * Hotel.price)	(0, 1500)
(beachDist < 5)	beachDist	(0, 5)

Table 1: Division of query predicates into *pInterval* and *pFunction*

The remainder of this paper is organized as follows: The formal problem definition is presented in Section 2. The proposed method is described in Section 3. In Section 4, evaluation of the method is presented. Section 5 presents the related work, and finally, Section 6 presents our conclusions.

2. PROBLEM DEFINITION

In this section we introduce the notations used in this work and formally define the problem of query-oriented relaxation for cardinality assurance.

2.1 Query Formulation

Consider a query Q over relations R_1, R_2 $\ldots R_k$ comprised of *query predicates* P_1, P_2, $\ldots P_d$, each of which is either a select or join condition. In this work, we limit ourselves to conjunctive queries. Hence query Q can be denoted as:

$$Q = (P_1 \wedge P_2 \wedge \ldots \wedge P_d) \quad (1)$$

Each predicate P_i in the query is assumed to be made up of two parts: the predicate function - *pFunction* and the interval of expected values for the predicate - *pInterval*. *pFunction* is a mathematical function consisting of one or more attributes from the relations R_1, R_2 $\ldots R_n$. In this work we consider *pFunctions* that are monotonic. To illustrate, $pFunction = age$ for the age-predicate of Q_1 while it is $Flight.price + 7 * Hotel.price$ for the price-predicate of Q_2. For each *pFunction*, *pInterval* gives the range of values of *pFunction* that are acceptable for Q. *pInterval* takes the form $\{pInterval_{lower}, pInterval_{upper}\}$ where $pInterval_{lower}$ is the minimum acceptable value of *pFunction* and $pInterval_{upper}$ is the maximum acceptable value. For example, the *pInterval* for the age-predicate of Q_1 is {25, 75} while it is {0, 1500} for the price-predicate of Q_2. Table 2.1 shows the breakdown of the predicates of query Q_1 and Q_2. The above definition of predicates is general and holds for generalized select predicates of the form $(l_1 < k_1 * R_1.x_1 + k_2 * R_2.x_2 \ldots + k_n * R_n.x_n < l_2)$ where x_i is an attribute of relation R_i. For these predicates, *pFunction* = $(k_1 * R_1.x_1 + k_2 * R_2.x_2 \ldots + k_n * R_n.x_n)$ and *pInterval* = $\{l_1, l_2\}$. A slightly different division of predicates is used for joins including equi-joins like $(R_1.x_1 = R_2.x_2)$ and non-equijoins like $(2 * R_1.x_1 < 3 * R_2.x_2)$. For join predicates, *pFunction* takes the form ($\| pFunction_1$ - $pFunction_2 \|$) where $pFunction_1$

and $pFunction_2$ are monotonic functions of attributes belonging to relations $R_1, R_2 \ldots R_n$. The definition of *pInterval* is unchanged and it consists of the minimum and maximum acceptable values of *pFunction*.

2.2 Measuring Query Relaxation

Suppose Q is the original query with predicates $P_1, P_2, \ldots P_d$ such that Q = $(P_1 \wedge P_2 \wedge \ldots \wedge P_d)$ and Q' is a potential relaxed query. Since Q has been relaxed to Q', the *pInterval* of a subset of predicates has been expanded to include more values. Therefore, for a particular predicate P_i, we define the relaxation of Q' with respect to Q as the sum of the difference between the lower bound of P_i in Q and Q' and the difference between the upper bound of P_i in Q and Q'.

$$RelX(Q'_{P_i}) = pInterval_{lower}(Q_{P_i}) - pInterval_{lower}(Q'_{P_i}) + pInterval_{upper}(Q'_{P_i}) - pInterval_{upper}(Q_{P_i}) \quad (2)$$

$$RelX(Q') = \Sigma_{i=1\ldots d} Rel(Q'_{P_i}) \quad (3)$$

where RelX(Q'_{P_i}) is the relaxation of Q' with respect to predicate P_i, while RelX(Q') is the total relaxation of Q'.

Following the same principle, we can symmetrically define the relaxation of a contracted query where the *pIntervals* of predicates have been shrunk to reduce the number of tuples returned.

2.3 Query-Oriented Relaxation

Using the above definitions, we now formally define the problem of query-oriented cardinality assurance. Consider an initial user query Q^I with predicates $P_1, P_2, \ldots P_d$ such that Q^I = $(P_1 \wedge P_2 \wedge \ldots \wedge P_d)$. Let the expected cardinality of Q^I be C_0 while its actual cardinality be C_I such that $C_I < C_0$. In order to increase the cardinality of Q^I, some subset of the query predicates $P_1, P_2, \ldots P_d$ must be expanded to include more results i.e., the *pIntervals* of this subset of predicates must be made bigger to increase the range of accepted values. The query-oriented relaxation problem has two main goals: (1) to modify Q^I to Q^F, the relaxed query, such that Q^F is within a threshold of the expected cardinality C_0, and (2) Q^F minimizes relaxation with respect to Q^I.

The problem of satisfying cardinality constraints exactly is NP-hard [3]. However, for a meaningful user experience, a query-oriented relaxation algorithm is required to be efficient. Therefore, instead of satisfying the query cardinality exactly, we aim to find a query Q^F that has cardinality within a tolerance threshold δ of C_0 (where δ is a tunable parameter). Similarly, finding the answer query with the absolute minimum relaxation would require an exhaustive search. Since this is not feasible nor appropriate for a good user experience, query-oriented relaxation focuses not on finding the query with the absolute minimum relaxation but one that has relaxation within a tolerance threshold of this minimum. We introduce the concept of *IdealRelX* - the Minimum Relaxation in the ideal Case to formalize this notion.

DEFINITION 1. *IdealRelX*

Given a query Q^I with expected cardinality C_0, and a cardinality threshold δ, *IdealRelX* denotes the minimum relaxation that any query having cardinality C_0 +/- δ can have. IdealRelX is thus the minimum relaxation that can possibly be obtained while satisfying the query cardinality.

Based on the above definition, we can formally state the problem of query-oriented relaxation. In general, our goal is not to find the query with relaxation *IdealRelX* but instead to find a query within a tolerance threshold of it.

DEFINITION 2. *Query-oriented Relaxation*

Given a query Q^I with expected cardinality C_0, a cardinality threshold δ and a relaxation threshold γ, compute a query Q^F such that

$$Cardinality(Q^F) = C_0 + / - \delta \quad (4)$$

$$RelX(Q^F) = IdealRelX + / - \gamma \quad (5)$$

3. OUR APPROACH

In this section we describe the QRelX algorithm and its underlying principles.

3.1 Overview

The goal of QRelX is to perform query-oriented relaxation for cardinality assurance. The key ideas underlying QRelX are query space transformation, layer-based navigation and incremental cardinality estimation.

Query space transformation is the process in which the input tables associated with a query are processed and combined to produce an output space or relaxation space. This transformation allows QRelX to relax both select and join predicates. Layer-based navigation is a traversal method that allows the algorithm to navigate the relaxation space in such a manner that relaxed queries with lower relaxation are always evaluated before those with higher relaxation. Layer-based navigation ensures that our algorithm can successfully minimize relaxation. Incremental cardinality estimation is a novel method of cardinality estimation introduced in this work. Incremental estimation allows the algorithm to delay tuple-level computations until they are absolutely necessary. Further this technique ensures that once a tuple has been found to satisfy a given query it is never reevaluated for any other query. The combined result of these three principles is that QRelX can efficiently relax queries that meet the cardinality constraint but also minimize relaxation.

Figure 2 shows the overall architecture of our system. The system consists of two main components: QXForm, the query space transormation module and IncRelX, the query relaxation module. The functions of these modules are descibed as follows:

- QXForm: This module of QRelX is responsible processing the input tables and creating the output or relaxation space. In particular:
  - It processes all input tables and partitions them into multi-dimensional input partitions. These partitions make up the *input space*
  - Once the input space has been created, all combinations of input partitions are considered and *relaxation mapping functions* are used to create corresponding output regions.
  - An *output space or relaxation space* is generated using the output regions and a grid structure is imposed on this space to facilitate the search for relaxed queries.

- IncRelX: This module is responsible for searching the relaxation space created by QXForm for relaxed queries that meet the expected cardinality. It is comprised of the following parts:
  - Layer-based navigation scheme: Since QRelX aims to produce queries that minimize relaxation, layer-based navigation is used to evaluate potential relaxed queries in order of increasing relaxation.
  - Incremental Cardinality Estimation: This algorithm presents a novel way to perform incremental cardinality calculations using previous cardinality information and thus reduce the expenses associated with cardinality estimation.

Each of these modules are discussed in detail in the following subsections.

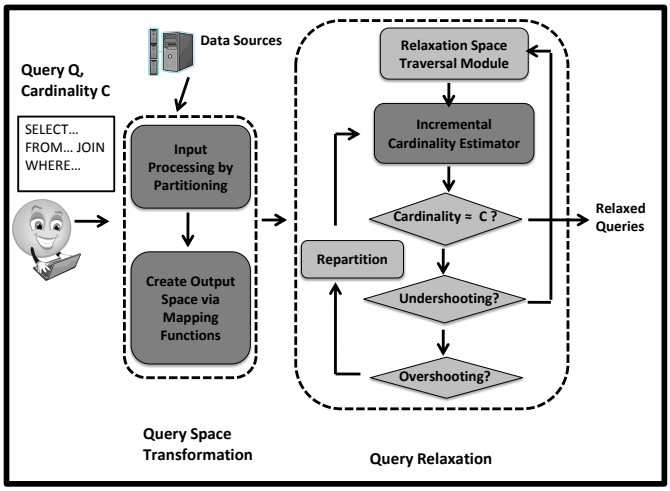

Figure 2: System architecture for QRelX

3.2 Query Space Transformation

The purpose of the QXForm, the query space transformation module is create an *output space or relaxaton space* for finding potential relaxed queries. QXForm takes an input the initial user query $Q = P_1 \wedge P_2 \ldots \wedge P_d$ and the set of tables $R_1, R_2 \ldots R_n$ associated with it. The transformation process consists of three steps:

Input Space: The first step in query space transformation is to use the input tables to create an input space. The input space is an representation of all the input tuples at a higher level of abstraction, namely as input partitions. The purpose of creating this space is to reduce computational expenses by using a higher level of abstraction and to delay tuple-level computations until a latter stage. To create this input space, each table associated with query Q is partitioned and all its tuples are placed into multi-dimensional partitions based on the attributes present in the query predicates. For example, figure 3 (a) and (b) depict the input partitions made for query Q_1. Two tables, namely the Hotel and Flight table, are involved in this query. Since two attributes from the Hotel table - price and distFromBeach - are included in Q_1, the tuples in Hotel are partitioned based on their price and beachDist values. Similarly, the Flight table is partitioned based on price since this is the only Flight table attribute present in the query. The number of divisions associated with each attribute for an input table is tunable paramter k. Further, if each attribute has k divisions, then the maximum number of input partitions for a given input table is k^p where p is the number of attributes from a particular table present in the query.

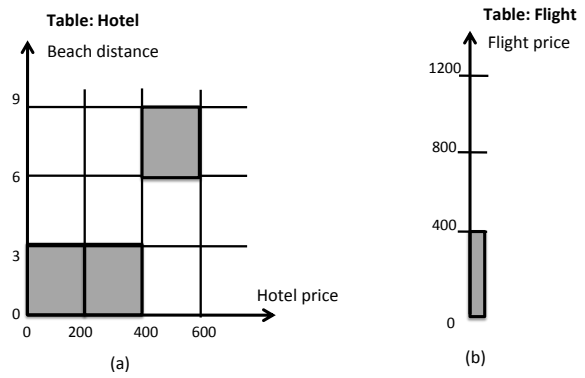

Figure 3: (a) Partitioning of Hotel table (b) Partitioning of Flight table

Result-tuples and Relaxation Mapping Functions: Before we proceed with the description of QXForm, we define the notion of result-tuples. Given a query Q involving relations $R_1, R_2 \ldots R_n$, a *result-tuple* is an ordered set of tuples $(t_1, t_2 \ldots t_n)$ such that tuple t_i belongs to relation R_i for i = 1...n. That is, a result-tuple is any valid combination of one tuple from each relation R_1 to R_n. If Q consists only of one relation R, then result-tuples correspond directly to tuples in R. We also note that the result-tuple concept is a generalization of the joined-tuple concept since all joins are valid with the right amount of relaxation.

Relaxation Mapping Functions: A concept central to the QRelX algorithm is a notion of the "amount of relaxation." Relaxation with respect to relaxed queries has already been defined in Section 2. However, this applies to result-tuples too and this measure of relaxation is essential while constructing the output or relaxation space.

In general, relaxation or *RelX()* corresponding to a variable is the difference between its acceptable value and its actual value, i.e.,

$$RelX(variable) = \| \ acceptable{-}value{-}actual{-}value \ \| \quad (6)$$

In this work, we introduce the concept of *relaxation mapping functions* (or simply *mapping functions*) to measure the relaxation of result-tuples with respect to individual query predicates. Figures 4.a to d show a few examples of relaxation mapping functions. The x-axes in these figures correspond to the actual value of a variable x, while the y-axes measure the relaxation of variable x with respect to the given range of acceptable values. For example, Figure 4.a shows the relaxation mapping function when the acceptable range of values for variable x is [0, 75) and x can take values from [0, ∞). The mapping function RelX() is defined such that while the value of x is within the range [0, 75), there is no relaxation and RelX(X) = 0. However, for (x >= 75), RelX(X) = (x - 75) measures how far the actual value of x is from the acceptable range. For instance, if $x = 85$ then the distance from the acceptable value is (85 - 75) = 10. Figures 4.b, c, d show mapping functions fo different ranges of acceptable values.

Relaxation of a result-tuple: When applied to a result-tuple *t*, a relaxation mapping function measures how far *t* is from a given

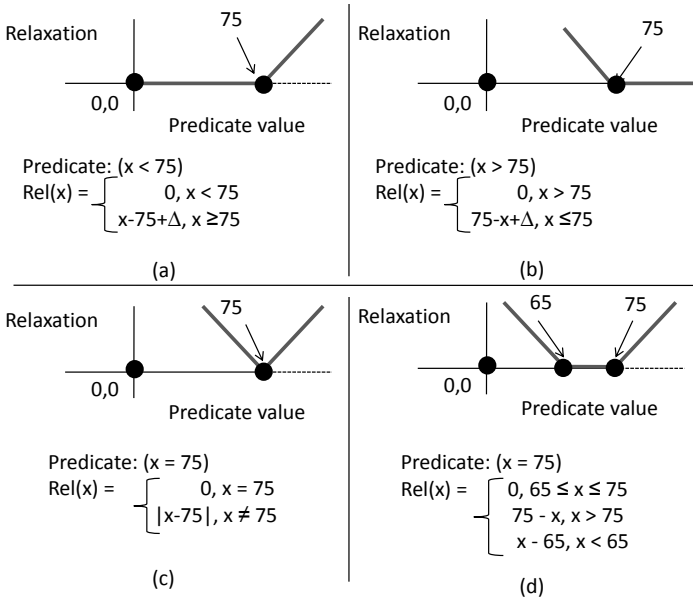

Figure 4: Examples of mapping functions

query predicate. Consider once again Figure 4.a discussed above; the mapping function in this figure can easily be used to calculate the the relaxation of a result-tuple. Suppose our query Q involves the relation R with contains attribute x. Further, suppose that predicate P belonging to query Q is defined such that $P = (R.x < 75)$. For a tuple t belonging to relation R, relaxation of t with respect to P is calculated as follows. Let $t[x]$ denote the value of attribute x for t. For ($0 < t[x] <$ 75), Rel($t[x]$) = 0 since the tuple t satisfies the predicate exactly. However, for ($t[x]$ >= 75), relaxation measures the difference between $t[x]$ and 75 i.e. ($t[x]$ - 75). So if $t[x]$ was 85, then based on the relaxation function, Rel($t[x]$) = 10. This is exactly the mapping function we discussed above.

If we define mapping functions for each predicate of Q then we can calculate the total relaxation of *t* with respect to Q. If Q is a query with predicates $P_1, P_2, \ldots P_d$ such that Q = ($P_1 \wedge P_2 \wedge \ldots \wedge P_d$) then the total relaxation of result-tuple *t* can be measured as follows:

$$RelX(t) = \Sigma_{i=1\ldots d} RelX(t_{P_i}) \quad (7)$$

where RelX(t_{P_i}) is the relaxation of *t* with respect to predicate P_i and RelX(*t*) is the total relaxation of *t*.

On similar lines, we can define RelXVector(), the relaxation vector of a result-tuple *t* that lists in order the individual relaxations of a result-tuple with respect to each query predicate.

$$RelXVector(t) = (RelX(t_{P_1}), RelX(t_{P_2}), \ldots RelX(t_{P_d})) \quad (8)$$

Relaxation Space: The relaxation mapping functions described above provide a powerful means to measure the relaxation of result-tuples. We use these mapping functions to create a *relaxation space* or *output space* that measures the relaxation of all possible result-tuples. By representing result-tuples according to their relaxation vectors, we can traverse the relaxation space to easily find potential relaxed queries. The relaxation space is a d-dimensional space with the original query Q lying at the origin. Further, each dimension of this space corresponds to the relaxation of result-tuples with respect to each of the d predicates of Q.

The relaxation space represents two kinds of information. First, every point in the relaxed output space is a relaxed query that is a potential answer. Second, the output space also represents the relaxation of all result-tuples that can be generated from the input tables. This information about result-tuples is stored in terms of *output regions*. An output region is an abstract data structure corresponding a unique combination of input partitions. It is a region in the output space such that all result-tuples from the given combination of input partitions lie within that region, i.e., an output region delimits the relaxation of result-tuples generated from the associated input partitions.

The process of creating the output space consists of two steps: first, to create output regions, and second, to create a grid in the output space that assigns output regions to grid cells. The first step provides us information about the location of result-tuples while the second facilitates the process of cardinality estimation.

Creating Output Regions: Since a tuple from one table can com-

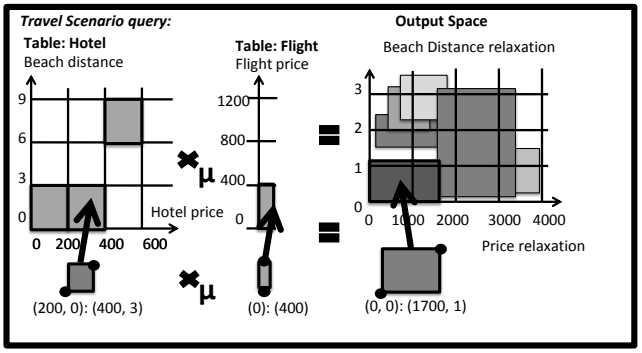

Figure 5: Creating an output space

bine with any other tuple from another table in Q (with the right amount of relaxation), we have to consider all combinations of the input partitions while creating output regions. For each combination of input partitions, we use one mapping functions per query predicate of Q to calculate the maximum and minimum relaxation that any result-tuple from these partitions could have. To actually compute these limits on relaxation, QXForm assumes that result-tuples called *pseudo-result-tuples* are present at the upper-right and lower-left points of each input patition. Once we know these result-tuples, mapping functions can be used to calculate the corresponding relaxations.

Figure 5 shows an example of this process for the running query example Q_2. The figure shows the partitioning of the Flight and Hotel tables along with the output space. Once input partitions have been created, one partition is picked from each table to calculate the minimum and maximum bounds for relaxation of result-tuples. In the current example, the pseudo-result-tuples are (200, 0) and (400, 3) for the Flight table partition while they are (0), (400) for the Hotel table partition.

Consider the first predicate of Q_2, (Flight.price + 7*Hotel.price < 1500). For the given pair of input partitions, the value of this predicate ranges from 1400 to 3200. But the expected value is less than 1500. Hence, the minimum and maximum relaxation with re-

spect to this predicate is the interval distance between (0, 1500) and (1400, 3200) which is (0, 1700). A similar calculation is done for the relaxation of the distance from the beach. For this predicate, we get (0, 1) as the range of relaxations. Thus we obtain an output region with the upper-right corner at (1700, 1) and the lower left corner at (0, 0). The figure shows the other five output regions created from the remaining combinations of input partitions. Note that while the output regions may overlap, they never share result-tuples.

Creating the output grid: The second step in creating the output space is to create a grid that stores the distribution of output regions over grid cells. This data is required while estimating cardinality and for optimization. For instance, consider a query located at point (1000, 1) in the output space. Based on the grid shown in the figure, we know that only the output region with lower bound (0, 0) and upper bound (1700, 1) needs to be examined for estimating query cardinality. The grid structure thus succeeds in reducing the computational expenses. To create the output grid, each dimension of the output space is partitioned according to a stepsize that reflects the preference of the predicate corresponding to that dimension. For ease of computation, we will assume that all all the predicates have equal preference and hence, all dimensions have a unit stepsize.

The following terminology is used to describe the output grid: a *grid point* is any point that lies at a point of intersection in the grid; a *gird cell* corresponding to a grid point is the unit orthotope that has the given grid point as its upper right corner; *neighbors* of a grid point are the grid points at a unit distance from the given grid point in only one dimension. For the purpose of our algorithm, we only consider the neighbors of a grid point that have higher relaxation than the given grid point.

3.3 Relaxation Algorithm

In this section we present IncRelX our core query-oriented relaxation algorithm for cardinality assurance. The goals of IncRelX are fourfold: (1) IncRelX is able to relax select as well as join predicates; (2) Given an initial user query Q^I and an expected cardinality C_0, IncRelX relaxes the query Q^I to Q^F (final query) such that Q^F satisfies the given cardinality; (3) Q^F minimizes the relaxation with respect to Q^I; and (4) The process of relaxation from Q^I to Q^F is computationally efficient. The QXForm framework described in Section 3.2 ensures that IncRelX satisfies the first goal of relaxing both select and join conditions. IncRelX satisfies the second goal of cardinality assurance by evaluating various queries in the relaxation space created by QXForm and selecting those that are closest to the expected cardinality. The third goal of minimizing relaxation is met by using a layer-based navigation scheme which ensures that queries with lower relaxation are always evaluated before queries with higher relaxation. The last goal is met by using an incremental cardinality estimation technique which reduces computational expenses by (a) evaluating only those tuples that are likely to satisfy any given query and (b) ensuring that once a tuple has been found to satisfy a query, it is never re-evaluated for any other query. We now describe the IncRelX algorithm beginning with our navigation scheme.

In the scenario where the original query does not meet its associated cardinality constraint, intuitively, the user would prefer a relaxed query that attains the required cardinality but is as close to the original query as possible. This implies that the search for potential relaxed queries in the relaxation space must be done in a way that prefers queries with lower relaxation. Our solution to this problem is to adopt an iterative layer-based navigation scheme to minimize relaxation.

Layer	Queries
0	(0, 0)
1	(0, 1); (1, 0)
2	(0, 2); (1, 1); (2, 0)
3	(0, 3); (1, 2); (2, 1); (3, 0)
k	(0, k); (1, k-1); (2, k-2)…(k-2, 2); (k-1, 1); (k, 0)

Table 2: Composition of layers in the output space

Layer-based traversal, as the name suggests, is an approach where potential relaxed queries are grouped into layers of equal relaxation, and are evaluated based on these layers. Moreover, relaxation layers are traversed in the order of increasing relaxation so that all queries lying in a layer with relaxation *p* are evaluated before queries lying in layer *(p+1)*. The intuition of the layer-based approach is presented in Figure 6. Figure 6.(a) of the figure depicts a two-dimensional relaxation space similar to the one created by running query Q_1. In a two-dimensional space, the total relaxation of a query is the sum of the relaxations on the two axes, and therefore, the relaxation layers are lines having slope -1. Figure 6.a shows relaxation Layers 0, 1, 2 and 3 having total relaxation equal to 0, 1, 2, and 3 units respectively. Layer-based traversal begins evaluation of queries from Layer 0 and then proceeds along Layers 1, 2, and 3. The numbering of the queries reflects the order in which queries are examined in each layer: first, query (0, 0) belonging to layer 0 is evaluated; second, queries (1, 0) and (0, 1) belonging to layer 1 are evaluated; third, queries (2, 0), (1, 1) and (0, 2) belonging to layer 2 are evaluated; next, queries (3, 0), (2, 1), (1, 2) and (0, 3) belonging to layer 3 are evaluated and so on.

Table 3.3 lists in tabular form the queries that are in the above layers of the relaxed space. From the values in the table, we know that a query (r1, r2) belonging to Layer k in the 2-dimensional relaxation satisfies the constraint that r1 >= 0, r2 >= 0 and (r1 + r2) = k.

In general, given a d-dimensional relaxation space, the queries belonging to Layer k can formulated mathematically as follows.

$$Layer(k) = \{(r_1, r_2 \ldots r_d) \mid (r_i >= 0)\ for\ i = 1 \ldots d \\ AND\ (r_1 + r_2 + \ldots + r_d = k)\} \quad (9)$$

This also turns out to be the formulation of the weak number theoretic compositions of the integer k and hence gives us an upper bound on the number of queries in each layer. Using combinatorial arguments, we can prove that such a layer k will contain $\binom{d+k-1}{d-1}$ queries at the most.

Due to the similarities between the layer-based navigation scheme and breadth-first traversal, our system implements the layer-based navigation scheme by using a modified breadth-first traversal strategy. As in traditional BFS, a queue is used to store the algorithmic data. Let us now look at the walk-through example of the layer-based navigation scheme as shown in Figure 6.(b). Step 1 in above figure shows the initial state of the traversal queue containing (0, 0). In Step 2, (0, 0) is popped from the head of the queue and its cardinality evaluated. Following this, its neighbors are computed to be (0, 1) and (1, 0) and added to the queue. In Step 3, (0, 1) is popped and its neighbors (0, 2) and (1, 1) are added to the queue. Finally,

when we come to Step 4, where (1, 0) is popped and its neighbors, (1, 1) and (2, 0) are computed. However, only (2, 0) is added to the queue since (1, 1) is already present in it. This omission of (2, 0) avoids re-examination of already evaluated queries. The overall effect of the traversal algorithm is that the queries are evaluated in the layer-based order depicted in Part (a) of same figure: (0, 0); (0, 1); (1, 0); (0, 2); (1, 1); (2, 0); (0, 3); (1, 2); (2, 1); and (3, 0).

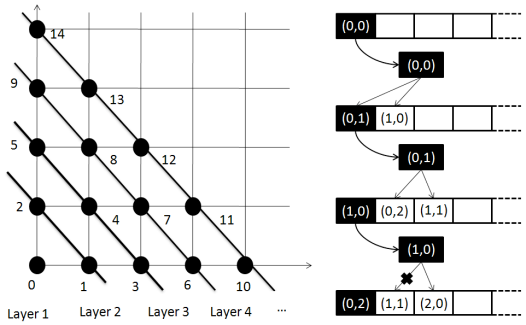

Figure 6: (a) Order in which grid cells are traversed by the *Traverse* algorithm. (b) Walk through example of *Traverse*

Algorithm 3.3 lists the pseudocode for the traversal algorithm. We first pop the element at the head of the queue; this is the query to be evaluated next (Line: 1). Lines 2-8: For loop generates all the neighbors of the most recently popped query by incrementing the query co-ordinates in a dimension-wise manner.
Lines 5-7: Checks if the given neighbor is already present in the queue. The neighbor is added only if the same value is not currently present in the queue.

Algorithm 1 Traverse (Queue q)

```
91: int[] counter = q.pop()
92: for i = 0 to d − 1 do
93:     int[] counterCpy = Copy(counter)
94:     counterCopy[i]++
95:     if !q.Contains(counterCopy) then
96:         q.push(counterCopy)
97: return counter
```

Lemma 1. *Traverse evaluates query Q with relaxation p before evaluating query Q' with relaxation p' if and only if p < p'.*

Proof: The objective of the *Traverse* algorithm is to traverse the output space in such a way that no query with total relaxation r is evaluated before all queries with relaxation r-1 have been evaluated. To prove this, we can model the output space as a graph G(V, E) such that the set of vertices of G is the set of all grid points and E is the set of edges created by connecting each grid point to its neighbors. The *Traverse* algorithm then performs a breadth-first traversal of G, and correctness of breadth-first traversal proves the correctness of *Traverse*.

In summary, the Traverse algorithm ensures that if a relaxed query Q' has relaxation smaller than another relaxed query Q'', the cardinality of Q' is evaluated before Q''. Consequently, this order of query evaluation guarantees that the first query $Q^\star$ found to satisfy the cardinality constraint has the minimum relaxation possible. Moreover, once we find a relaxed query that satisfies the required cardinality, we do not need to evaluate any more queries, and thus we can reduce computational expenses.

3.3.1 Incremental cardinality estimation

One of the main challenges of query relaxation is the computational expense associated with repeated query execution. Whether the refinement is user-driven or automatic, relaxation algorithms require that the cardinality of a large number of queries be estimated. However, traditional relaxation techniques have no memory about prior cardinality estimations and do not reuse previously computed cardinalities to increase computational efficiency. Instead of the repetitive cardinality estimation model, in this work, we propose a novel incremental cardinality estimation model that takes advantage of space partitioning and query containment for performing rapid cardinality calculations. We start with some basic observations of the relaxation space.

Cardinality in the Relaxation Space:

Suppose the original query Q given by Q = ($P_1 \wedge P_2 \wedge \ldots \wedge P_d$) and its expected cardinality is C. As discussed previously, the relaxation space represents not only all possible relaxed queries but also the relaxations of all combined-tuples. As a result, the cardinality of a query Q' corresponding to the point X = (x_1, $x_2 \ldots x_d$) in the relaxation space is equal to the number of combined-tuples included in the orthotope extending from the origin to X. In terms of the relaxations, this orthotope includes all tuples T such that:

$$(0 <= Rel(T_{P_i}) <= x_i) for i = 1 \ldots d \tag{10}$$

Figure 3.3.1.a shows a visual representation of this orthotope for a two-dimensional output space created by Q_1.

Query Containment: The above definition of query cardinality provides us a key insight that can lead to dramatic improvements in cardinality estimation efficiency. Consider the two queries Q' = (r'_1, r'_2) and Q'' = (r''_1, r''_2) shown in Figure 3.3.1.b. Q'' is said to be *contained* in Q' since the rectangle corresponding to Q'' is completely contained inside the rectangle corresponding to Q'. Containment implies that all the combined-tuples belonging to Q'' also belong to Q'. Therefore, if we evaluate the cardinality of Q'' before evaluating Q', then we can avoid the expensive cardinality computation by reusing the already computed cardinality of Q''.

In general, given two queries Q' and Q'' in a d-dimensional space such that Q' = (r'_1, $r'_2 \ldots r'_d$) and Q'' = (r''_1, $r''_2 \ldots r''_d$), we say that Q'' is *contained* in Q' if $r'_i >= r''_i$ for $(i = 1 \ldots d)$. Moreover, the containment also implies that $\Sigma_{i=1\ldots d} r'_i < \Sigma_{i=1\ldots d} r''_i$. In other words, the total relaxation of Q'' is lesser than that of Q' and hence our layer-based navigation scheme will necessarily evaluate Q' before evaluating Q''.

This above set of observations allows us to take advantage of space partitioning and query containment in the relaxation space to formulate the following incremental cardinality estimation model.

Incremental Estimation Model: The principle guiding our Incremental Estimation Model is that once a combined-tuple has been found to satisfy a query Q, that combined-tuple is never re-evaluated for any other relaxed query Q' that contains Q. However, this combined-tuple is incorporated in the cardinality of Q' automatically. The estimation model of IncRelX examines only a small subset of combined-tuples for each query while mainly reusing cardinality values of queries from the previous layer. This approach

ensures that IncRelX does not waste resources in rerunning previous computations. We first describe the framework that allows cardinality to be calculated incrementally.

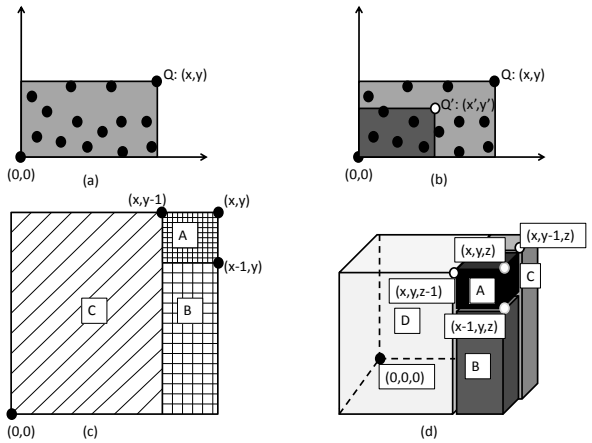

Figure 7: (a) Representation of cardinality in output space, (b) Motivation for incremental cardinality estimation, (c) Decomposition of a 2-D orthotope, and (d) Decomposition of a 3-D orthotope

Decomposition of the Orthotope: Each query in the relaxation space has an associated query orthotope and the first step towards incremental estimation is determining how this orthotope can be divided to reuse cardinality values from the previous relaxation layer. Formally, given a query query Q corresponding to point X = (x_1, $x_2 \ldots x_n$), we seek to partition the query Q's orthotope into smaller sub-orthotopes such that (1) the sub-orthotopes are disjoint, and (2) the upper-right corners of sub-orthotopes correspond to queries from the previous relaxation layer.

Figures 3.3.1.c and d show one such partitioning of two and three dimensional orthotopes. In each of the above figures we see that not only are the sub-orthotopes disjoint, but their upper-right corners also correspond to queries in the previous relaxation layer i.e. the total relaxation of these queries is one unit less than the relaxation of Q. As the figures demonstrate, a two-dimensional orthotope has to be partitioned into three sub-orthotopes to satisfy the above criteria while a three dimensional orthotope requires four sub-orthotopes. To aid the understanding of this concept, the sub-orthotopes have been named to reflect their geometry. Sub-orthotope A is called a "cell", B a "pillar", C a "wall", and D a "block". Moreover, to preserve uniqueness, the sub-orthotopes are associated with the points at their their upper-right corner. Thus, in figure 3.3.1.c, the orthotope or wall corresponding to query (x, y) is composed of the cell of (x, y), the pillar of (x-1, y) and the wall of (x, y-1). Similarly, the orthotope or block of (x, y, z) in part d is composed of the cell of (x, y, z), the pillar of (x-1, y, z), the wall of (x, y-1, z) and the block of (x, y, z-1). In general, a d-dimensional orthotope has to be divided into (d+1) sub-orthotopes to satisfy the above criteria. The previous discussion implies that to estimate cardinality incrementally, we need to calculate the cardinality values of each of the d+1 sub-orthotopes. To illustrate, in a 2-dimensional space, we need to calculate the values of the cell (A), pillar (B) and wall (C) for each query (as shown in Figure 8), while in the 3-dimensional space, we have to calculate the values of the cell (A), pillar (B), wall (C) and block (D) (Figures 3.3.1). In d-dimensional space, the following equations mathematically define the (d+1) sub-orthotopes associated with each query. For instance, the cell denoted by O_1 is the sub-orthotope having unit length in each dimension and having (x_1, $x_2 \ldots x_d$) as its upper-right corner. The pillar, denoted by O_2, has length x_1 in the first dimension while it has unit length in the other (d-1) dimensions. The wall, i.e. O_3, has length x_1 and x_2 in the first two dimensions while it has unit length in the other (d-2) dimensions. The orthotopes are defined as follows in a d-dimensional relaxation space.

$$O_1(cell) = \{(y_1, y_2 \ldots y_n) \mid (x_i - 1 < y_i <= x_i)\ for\ i = 1 \ldots d\} \quad (11)$$

$$O_2(pillar) = \{(y_1, y_2 \ldots y_n) \mid (0 <= y_1 <= x_1)\ AND\ (x_i - 1 < y_i <= x_i)\ for\ i = 2 \ldots d\} \quad (12)$$

$$O_3(wall) = \{(y_1, y_2 \ldots y_n) \mid (0 <= y_1 <= x_1)\ AND\ (0 <= y_2 <= x_2)\ AND\ (x_i - 1 < y_i <= x_i) for i = 3 \ldots d\} \quad (13)$$

$$O_i = \{(y_1, y_2 \ldots y_n) \mid (0 <= y_i <= x_i)\ for\ i = 1 \ldots i\ AND\ (0 <= y_2 <= x_2 - 1)\ AND\ (x_i - 1 < y_i <= x_i)\ for\ i = i+1 \ldots d\} \quad (14)$$

$$O_d + 1 = \{(y_1, y_2 \ldots y_n) \mid (0 <= y_i <= x_i) for i = 1 \ldots d\} \quad (15)$$

Using the above equations, we can formally write the compositions of orthotopes in two and three dimensions Fig.3.3.1.c and d) as follows:

$$O_3(x, y) = O_1(x, y) + O_2(x - 1, y) + O_3(x, y - 1) \quad (16)$$

$$O_4(x, y, z) = O_1(x, y, z) + O_2(x - 1, y, z) + O_3(x, y - 1, z) + O_4(x, y, z - 1) \quad (17)$$

Generalizing the above formulas for a d-dimensional space we get:

$$O_d(x_1, x_2, x_3 \ldots x_d) = O_1(x_1, x_2, x_3 \ldots x_d) + O_2(x_1 - 1, x_2, x_3 \ldots x_d) + O_3(x_1, x_2 - 1, x_3 \ldots x_d) + \ldots + O_{d+1}(x_1, x_2, x_3 \ldots x_d - 1) \quad (18)$$

However, for every point, we not only need to calculate the total cardinality of the orthotope, but we also have to calculate the cardinalities of the other d sub-orthotopes defined above. Referring back to Figures 8 and 3.3.1, we can make the following observations:

- For the two-dimensional relaxation space in Figure 8, we have:
  - Pillar (x, y) = Cell (x, y) + Pillar (x-1, y)
  - Wall (x, y) = Pillar (x, y) + Wall (x, y-1)
- For the three-dimensional relaxation space in Figure 3.3.1, we have:
  - Pillar (x, y, z) = Cell (x, y, z) + Pillar (x-1, y, z)

– Wall (x, y, z) = Pillar (x, y, z) + Wall (x, y-1, z)

– Block (x, y, z) = Wall (x, y, z) + Block (x, y, z-1)

Thus, we begin to see a pattern that can be converted into a recurrence: $O_i(x_1, x_2 \ldots x_d) = O_{i-1}(x_1, x_2 \ldots x_d) + O_i(x_1, x_2 \ldots x_{i-1} - 1) for\ i = 2 \ldots d + 1$. The only sub-orthotope that doesn't have a recurrence is the cell since this is the part of the cardinality that is unique to every query. To calculate the cardinality of the cell, combined-tuples belong to various output regions have to be examined to find those that lie in the given cell. Thus, in summary,

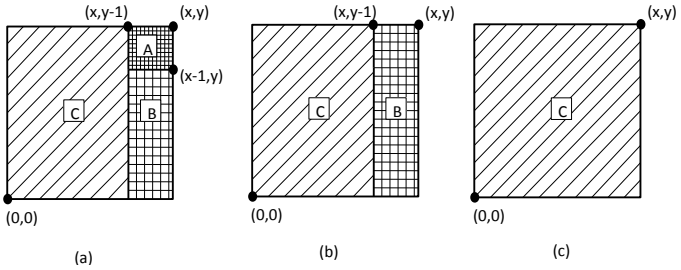

Figure 8: Orthotope recurrences in a two-dimensional space.

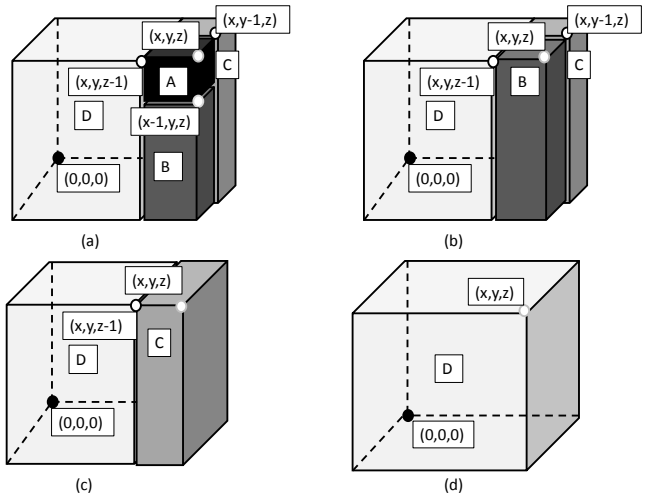

Figure 9: Orthotope recurrences in a three-dimensional space.

we can incrementally calculate the cardinality of queries using the equations below:

$$O_1 = \{(y_1, y_2 \ldots y_n) \mid (x_i - 1 < y_i <= x_i) \; for\ i = 1 \ldots d\} \quad (19)$$

$$O_i(x_1, x_2 \ldots x_d) = O_{i-1}(x_1, x_2 \ldots x_d) + O_i(x_1, x_2 \ldots x_{i-1} - 1 \ldots x_d)\ for\ i = 2 \ldots d + 1 \quad (20)$$

These recurrences ensure that once the cardinality of the cell has been determined, it takes a constant number of steps to calculate the other cardinalities. Algorithm 3.3.1 presents the pseudo-code that performs the incremental cardinality estimation. Note that

Algorithm 2 CardinalityEstimation(int[] upper, Hash h(String, int[]))

```
91: int[d+1] card
92: card[0] = ComputeCellCardinality(upper)
93: for i = 1 to d do
94:     upper[i]–
95:     int[] card_i = h.get(asString(upper))
96:     card[i] = card[i-1] + card_i[i]
97:     upper[i]++
98: h.insert(asString(upper), card)
99: return card[d]
```

the sub-orthotope cardinalities are stored in an array as [O_1, O_2, O_3,... O_{d+1}].

The *CardinalityEstimation* algorithm takes as input the query being evaluated, and a pointer to the recording keeping data structure, which in this case is a hash. The hash stores previously evaluated queries along with the values of the associated (d+1) orthotopes. Briefly, the algorithm functions as follows:
Line 2: Computes the cardinality of the associated cell by examining combined-tuples. The *ComputeCellCardinality* algorithm below is responsible for this evaluaion.
Lines 3-7: Computes cardinalities of the remaining (d+1) orthotopes using the previously defined recurrences.
Line 8: Updates the hash with the given cardinality values

The other part involved in cardinality estimation of IncRelX is the algorithm that computes the cardinality of the cell associated with each query. The goal of this algorithm is to find the combined-tuples tha lie in the query cell and remove them from the pool of combined-tuples so that they are not evaluated again. Algorithm 3.3.1 presents this algorithm. The algorithm functions as follows:
Lines 2-4: Computes the lower bound of the cell
Line 5: Gets the list of output regions that are present in the cell
Line 6: Gets the combined tuples from each of the above regions that have not satisfied a previous query. Output regions are also materialized in this step.
Lines 7-10: Checks the combined-tuples, and removes those that satisfy the current query. Also updates cardinality.

Algorithm 3 ComputeCellCardinality(int[] upper)

```
91: int card = 0
92: int[d] lower = 0, 0... 0
93: for i = 0 to d − 1 do
94:     lower[i] = upper[i] - 1
95: List<OutputRegions> list = GetOutputRegions(upper, lower)
96: List<CombinedTuples> tuples = GetCombinedTuples(list)
97: for t in tuples do
98:     if t.Satisfies(lower, upper) then
99:         tuples.remove(t)
910:        card++
911: return card
```

3.3.2 Overall Algorithm

The cardinality estimation of IncRelX proceeds through the interleaving of the traversal algorithm and the cardinality estimation algorithm. The algorithm begins at the origin and sequentially traverses queries in higher layers. For each query, estimates the cell cardinality of the query and the cardinality of the other d sub-orthotopes. Once the all the cardinalities for a query have been

evaluated, its total cardinality is compared to its expected cardinality. If the cardinality is within δ of the expected cardinality, the algorithm evaluates other queries in the current layer and then returns the answers. If a query undershoots the expected cardinality, the algorithm proceeds to the next higher layer. However, if the query overshoots the expected cardinality, the query cell is repartitioned. Algorithm 3.3.2 shows the pseudo-code for the overall algorithm.

Algorithm 4 IncRelX(int expectedCardinality, int delta)

```
91:  Queue q, Hash h(String, int[]), List<int[]> answers
92:  int[d] counter = 0,0...0
93:  q.push(counter)
94:  counter = Traverse(q, h)
95:  int minimumRelaxationLayer = 0
96:  int currentRelaxationLayer = MAX_INTEGER_VALUE
97:  while (minimumRelaxationLayer >= currentRelaxationLayer)
     do
98:     int card = CardinalityEstimation(counter)
99:     currentRelaxationLayer = GetRelaxation(counter)
910:    if (||card - expectedCardinality|| < δ) then
911:       answers.add(counter)
912:       minimumRelaxationLayer = currentRelaxationLayer
913:    else if (card > expectedCardinality) then
914:       answers.add(Repartition(counter))
```

4. EXPERIMENTS

In this section we describe the preliminary evaluation of the QRelX algorithm.

4.1 System Implementation

For the evaluation of the QRelX algorithm, we implemented in Java the system described in Section 3.1. As shown in Figure 2 of Section 3.1, our system implementation consists of two major components: QXForm, the query space transformation component and IncRelX, the query relaxation component.

The query space transformation component QXForm takes as input the user query along with its associated input tables and processes them to create the relaxation space. In our implementation, the QXForm component is composed of three modules. The first module is responsible for processing the input tables and partitioning them into multi-dimensional input partitions. The second module is responsible for the creation of output regions. It computes all possible combinations of input partitions and constructs an output region for each individual combination. Output region construction is done by using the special interval distance functions that implement relaxation mapping functions. The last module of QXForm creates a grid in the relaxation space. It then assings to each grid cell the set of output regions that overlap with the given cell. This information about the output space is stored in a global hash table **H**. **H** is indexed by keys which are strings representing the bounds of the grid cells. The value associated with each key of **H** is a composite data structure storing (1) a list of references to output regions containing the given grid cell, (2) the d+1 cardinality values required for cardinality estimation, and (3) meta-data about the cell such as whether the cell has already been evaluated.

The second component of our system implements IncRelX, our core relaxation algorithm that meets query cardinality while minimizing the amount of relaxation. This component is also composed of three modules: the traversal module, the cardinality estimation module and the overall module. The traversal module implements the layer-based navigation scheme described in Section 3.3 for minimizing relaxation. Using the global **H** table to run the traversal algorithm, this module iteratively computes the next grid cell to be evaluated. The second relaxation module is responsible for cardinality estimation and implements the Algorithms 3.3.1 and 3.3.1 in Section 3.3. Once again, information from the global hash table **H** is used for rapid cardinality calculations. After the grid cell cardinality has been calculated, the cardinality values stored in **H** are used to calculate the other d cardinalities in a constant number of steps. Additionally, this module uses a supplementary hash-based data structure to store information about materialized output regions. This data structure is required because we do not want to repeatedly materialize the output regions for each grid cell. Moreover, this data structure enables us to only keep track of the result-tuples that haven't already satisfied a previous query. The last module of IncRelX implements the overall algorithm described in 3.3.2 by utilizing the functionality of the traversal module to navigate the relaxation space and cardinality estimation module to efficiently estimate cardinality of queries in the relaxation space.

4.2 Experimental Setup

In order to evaluate our solution, we ran preliminary experiments testing the efficiency of our algorithm for various parameters and compared the performance of our system with the TQGen algorithm proposed by Mishra et al. in [19]. The TQGen algorithm was chosen because this approach has similarities with QRelX and it focuses on cardinality assurance. TQGen however, does not focus on minimizing relaxation with respect to the original query. As a result, our experiment only compare the performance of the two methods. For our experiments, we implemented TQGen in Java for the single cardinality case.

All experiments were run on the TPC-H benchmark data, specifically using data from the PartSupp and LineItem tables. The PartSupp table stored the part key, supplier key, available quantity, supply cost and comments. The LineItem table included sixteen attributes, of which our queries included the attributes order key, part key, supplier key, quantity and extended price. The size of these input tables varied from 1k to 6k tuples for our experiments.

4.3 Experimental Results

Effect of Stepsize on Performance: Our first set of experiments analyzed the effect of input and output (or relaxation) stepsize on the performance of QRelX. Figure 4.3 shows the results of this set of experiments where a fixed query was relaxed to obtain a particular cardinality. The input and output stepsize for this query was varied in turn while keeping the other stepsize constant at 100 units. The x-axis of Figure 4.3 shows the size of the stepsize that is being varied. The y-axis on the other hand shows the time required to relax the given query for that particular stepsize. The two bars for each stepsize respectively show the execution time when one step size is 100 and the other has the value corresponding to the x-axis.

For QRelX, the input stepsize determines the number of input partitions that will be created during query space transformation. It affects the time required in partitioning the input tables but does not affect the quality of the results produced by QRelX. Relaxation stepsize or output stepsize on the other hand denotes the size of grid cells created in the relaxation space. As a result, output stepsize affects not only the time required for generating the output space but also the quality of answers produced. A lower stepsize increases computational expenses involved in creating the relaxation space

but it also allows the algorithm to obtain answers very close to the ideal minimum relaxation. Figure 4.3 shows this inverse relationship between stepsize and execution time. A higher stepsize consistently leads to lower execution time.

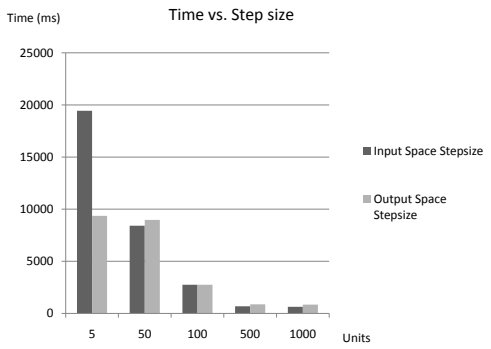

Figure 10: Graph showing the effect of input and output stepsize on QRelX performance.

Performance comparison with TQGen: The second set of experiments we ran compared the efficiency of our QRelX technique to that of TQGen. While TQGen provides cardinality assurance, it does not focus on minimizing relaxation. As a result, the QRelX algorithm always outperformed TQGen in terms of relaxation with respect to the original query. Figure 4.3 shows the results of our comparative study focusing on performance. One fixed data set was used for these experiments, but parameters for the test query were varied to obtain different initial cardinalities. The expected cardinality was also fixed. The x-axis in Figure 4.3 shows the value of the original cardinality relative to the expected cardinality. Values closer to the origin indicate that the query had much lower cardinality compared to the expected cardinality while those away from the origin correspond to values with the original cardianlity close to the expected cardinality. The y-axis of this figure shows the execution time for relaxing each query to meet the expected cardinality. The bars correspond to the time required by QRelX and TQGen respectively. The first trend we observe from this figure is that the cardinality of the original query is inversely related to the time required to run the algorithm, i.e., if the original cardinality is close to the expected cardinality, then the algorithms take a shorter time to find the new query. This observation is to be expected because a higher original cardinality means that the algorithms have to explore a smaller relaxation space to find the new query. The second trend we see in the figure is that QRelX consistently performs better than TQGen for various query cardinalities. Irrespective of the original cardinality QRelX is at least as efficient as TQGen. Since TQGen uses a repetitive execution model, these results support the claim that our incremental query estimation model is more efficient than the traditional repetitive execution model. Moreover, it is important to keep in mind that QRelX also minimizes relaxation while ensuring cardinality.

One drawback we observed during this set of experiments was that TQGen outperformed QRelX when the data was non-uniformly scaled. For example, QRelX performed poorly when one attribute (say A) in the query ranged from 0 to 1 while the other attribute (say B) ranged from 1 to 10000. This discrepancy was not observed when the ranges of attributes were of the same order. The

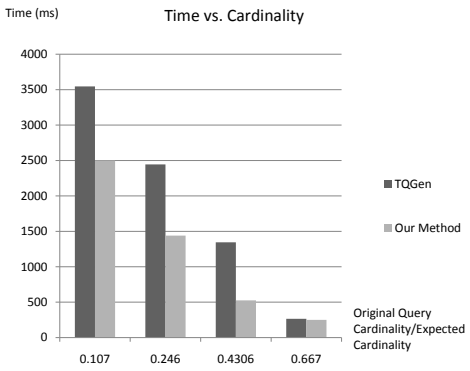

Figure 11: Graph showing the comparison between the performance of TQGen and QRelX for queries with varying cardinality.

above result was also to be expected because initially QRelX used the same output stepsize for all dimensions of the output space. For instance, in the previous example, QRelX could not apply a stepsize of say 0.1 on A and a stepsize of 1000 on B. Following this set of experiments, we modified our algorithm to support varying stepsizes. This modification not only improved performance for QRelX but also enabled QRelX to incorporate user preferences on particular query predicates.

The above set of preliminary experiments thus show that the QRelX algorithm can help reduce the computational expenses of query relaxation while also ensuring minimal relaxation. They also give insight into the performance issues of QRelX that can be considered for future optimizations.

5. RELATED WORK

Several methods to perform query relaxation have been proposed in the literature. In [8] Gaasterland describes a method to relax queries through deduction using semantic information about a database. Muslea et al. proposed a means to use artificial intelligence techniques like Bayesian structures to discover relationships between attributes and use them to relax empty result-set queries [20, 21]. However, these methods do not ensure query cardinality and do not focus on minimizing relaxation. In [15] Luo describes a method to identify empty result-set queries based on previous query executions. Algorithms to relax empty result-set queries are not proposed in this work.

[18] describes a interactive approach to query refinement. Information about the underlying database is used help users narrow the list of alternative queries. The drawback of this method is that the user may not have enough information about the database to meaningfully express his or her preferences. Koudas et al. use skyline algorithms for query relaxation in [12]. Skyline algorithms are useful for empty result-set queries but have limited potential for ensuring cardinality. Query relaxation in terms of approximate query answering has also been studied in the context other data models such as XML queries and semi-structured data [1, 22, 24].

Given a query, top-k algorithms can generate k tuples that best fit a query.Therefore, they represent a tuple-oriented way of ensuring query cardinality while maintaining closeness to the original query. [2, 5–7, 13, 14, 16] describe some of the techniques used for top-

k queries. Most top-k research has focused on selection queries but [9] presents a way to perform top-k over joins. In [10] Kadlag et al. propose a relaxation algorithm that generalizes the top-k idea for range queries involving selection. In [4], Carey et al. proposed the STOP AFTER operator which can help return a fixed number of values in cases where queries return too many results. However, as discussed in section 1 the goal of our algorithm is not to focus of the result-tuples but to provide an alternate query that concisely characterizes the result.

Our work on cardinality assurance is also related to generating test queries for databases. Bruno et al. addressed this question in [3] where they proved the NP-hardness of the problem and proposed a hill climbing approach. Mishra et al. also address the test query generation question in [19]. These algorithms however do not focus on obtaining the closest queries since query semantics are irrelevant for testing purposes.

Lastly, cardinality estimation is an important part of our algorithm and [11, 17, 23] describe some of the recent research in this area. These methods however do not use the incremental estimation approach proposed in this work.

6. CONCLUSION

In this work we proposed a novel algorithm for query-oriented relaxation that can provide cardinality assurance. A query-oriented approach to relaxation can be more beneficial for many applications because it can provide a query characterizing the selected tuples, facilitate future query refinement using this query and also ensure that the query results accurately represent the underlying data.

QRelX, our proposed algorithm for query-oriented relaxation attains two goals: (1) given an initial user query Q_I with expected cardinality C_0, QRelX relaxes query Q_I to Q_F such that Q_F attains the cardinaly C_0, and (2) Q_F minimizes relaxation with respect to Q_I. QRelX attains these goals through the key ideas of query space transformation, layer-based navigation of the relaxed query space and incremental cardinality estimation. QXForm, our framework for query space transformation, uses mapping functions to enable the relaxation of both, select and join predicates. Layer-based navigation enables QRelX to minimize relaxation of Q_F to meet the second goal of QRelX. Lastly, incremental cardinality estimation through IncRelX allows the algorithm to reuse previous cardinality values and rapidly estimate query cardinalities. The combination of the above techniques leads to an efficient query relaxation algorithm to provide cardinality assurance.

We also presented preliminary results demonstrating the performance of our algorithm and a comparison of our approach with TQGen [19]. Further work is required for the comparison of our approach with tuple-oriented strategies for relaxation.

7. REFERENCES

[1] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. In *EDBT*, pages 496–513, 2002.

[2] N. Bruno, S. Chaudhari, and L. Gravano. Top-k selection queries over relational databases. *ACM Transactions on Database Systems*, 27(2):153–187, 2002.

[3] N. Bruno, S. Chaudhari, and D. Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1721–1725, 2006.

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

[5] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD*, pages 346 – 357, 2002.

[6] S. Chaudhari and L. Gravano. Evaluating top-k selection queries. In *VLDB*, pages 397–410, 1999.

[7] S. Chaudhuri and G. Das. Automated ranking of database query results. In *CIDR*, pages 888–899, 2003.

[8] T. Gaasterland. Cooperative answering through controlled query relaxation. *IEEE Expert: Intelligent Systems and Their Applications*, 12(5):48–59, 1997.

[9] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.

[10] A. Kadlag, A. V. Wanjari, J. Freire, and J. R. Haritsa. Cardinality estimation using sample views with quality assurance. In *DAFSAA*, pages 594–605, 2004.

[11] P. Åke Larson, W. Lehner, J. Zhou, and P. Zabback. Cardinality estimation using sample views with quality assurance. In *SIGMOD*, pages 175–186, 2007.

[12] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.

[13] C.-I. Lee and C.-J. Tsai. An efficient approach to extracting and ranking the top k interesting target ranks from web search engines. *Informatica (Slovenia)*, 25(3), 2001.

[14] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. Ranksql: Query algebra and optimization for relational top-k queries. In *SIGMOD*, pages 131–142, 2005.

[15] G. Luo. Efficient detection of empty-result queries. In *VLDB*, pages 1015–1025, 2006.

[16] L. P. Mahalingam and K. S. Candan. Query optimization in the presence of top-k predicates. In *Multimedia Information Systems*, pages 31–40, 2001.

[17] T. Malik, R. C. Burns, and N. V. Chawla. A black-box approach to query cardinality estimation. In *CIDR*, pages 56–67, 2007.

[18] C. Mishra and N. Koudas. Interactive query refinement. In *EDBT*, pages 862–873, 2009.

[19] C. Mishra, N. Koudas, and C. Zuzarte. Generating targeted queries for database testing. In *SIGMOD*, pages 499–510, 2008.

[20] I. Muslea. Online query relaxation. In *SIGKDD*, pages 246–255, 2004.

[21] I. Muslea and T. Lee. Online query relaxation via bayesian causal structures discovery. In *AAAI*, pages 831–836, 2005.

[22] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Approximate xml query answers. In *SIGMOD*, pages 263–274, 2004.

[23] Z. Zhang, Y. Yang, R. Cai, D. Papadias, and A. K. H. Tung. Kernel-based skyline cardinality estimation. In *SIGMOD*, pages 509–522, 2009.

[24] X. Zhou, J. Gaugaz, W.-T. Balke, and W. Nejdl. Query relaxation using malleable schemas. In *SIGMOD*, pages 545–556, 2007.