Query-Oriented Relaxation for Cardinality Assurance

Manasi Vartak

Worcester Polytechnic Institute

Abstract:

Queries in scientific and business applications often have cardinality constraints. However, current database systems provide minimal support for cardinality assurance. Therefore, users must manually reformulate queries to attain the desired cardinality while remaining close to the original query. But this approach is cumbersome, wastes system resources and offers no guarantee of success. To address the cardinality assurance problem, we propose QRelX, the first framework to efficiently auto-generate refined queries meeting both, cardinality and proximity, criteria. QRelX employs a three phase strategy of Proximity-Based Search, Need-Based Result Generation, and Incremental Cardinality Estimation to preferentially investigate queries close to the original query and minimize computational expenses through result-sharing and delayed tuple-level computations. Our experiments show QRelX to perform up to 2-3 orders of magnitude faster and generate queries with at least 25% smaller refinements compared to state-of-the-art techniques.

Problem and Motivation:

In diverse domains such as business intelligence, decision support and web search, user queries often have cardinality constraints, i.e., constraints on the number of results produced by a query. For instance, a doctor may formulate health-related queries with cardinality constraints to identify a fixed number of patients for drug trials. However, database systems currently provide minimal support for cardinality assurance - they cannot automatically refine queries to ensure result size. This shortcoming frequently leads to queries that return no results [10], too many results [2], or too few results [8].

When a query does not meet the desired cardinality, a user must manually reformulate the query without arbitrarily refining its predicates. We capture these refined query requirements as follows: (1) **Cardinality criteria**: the refined query must attain the desired cardinality, and (2) **Proximity criteria**: the refined query must be close to the original query to preserve its semantics. Manual refinement with these goals is tedious and frustrating because a user rarely knows the characteristics of the underlying data set. Moreover, repeated query execution consumes significant time and effort from the user, wastes system resources and offers no guarantee of success.

Motivating Examples:

EXAMPLE 1. Medical researcher A wants to conduct a study examining the link between obesity and low income. Since A's grant provides him resources to study 2000 volunteers, A formulates query Q1 to select these candidates. However, Q1 can identify only 1300 volunteers.

```
Q1:SELECT * FROM PatientRecords
WHERE (25 < age < 55) AND (income < $55000)
AND (BMI > 30) AND (familyHistory = 0)
```

Due to the deficiency of results, A must manually refine Q1 through a frustrating trial-and-error process where he attempts a large number of refined queries. For instance, some possible refinements for Q1 are: (1) expanding the range to 25-60 years, (2) increasing the income threshold to $65,000, (3) allowing the individual to have a family history of obesity, or (4) any combination of the above. Even for a simple query like Q1, which spans a single table and contains four select predicates, A must try numerous refined queries. Additionally, he must ensure that the refinement doesn't grossly alter the original query, e.g., refining the income threshold to $85,000 may return 2000 results, but the refined query will violate the original low income criteria.

As demonstrated by Example 2, manual query refinement is further complicated when a query combines two or more tables and requires refinement of join predicates.

Consider the data collected by two physiological sensors X and Y that respectively monitor blood oxygen content and blood pressure. Scientist B wants to study the drop in oxygen content when blood pressure goes below the normal threshold. To have statistically significant findings, she requires at least 10,000 pairs of readings that satisfy specific temporal and physiological conditions as defined in Q2. However, B's query only returns 5,000 pairs of readings, and she must manually refine Q2 until it produces 10,000 results.

```
Q2: SELECT * FROM SENSOR X, SENSOR Y WHERE
X.time = Y.time AND X.bloodOxygen < 80
AND Y.systolic < 90 AND Y.diastolic < 60
```

Refinement of Q2 is more challenging than that of Q1 because Q2 can be reformulated by refining only its select predicates, only its join predicate, or a combination of both. We note that join refinement is beneficial for queries like Q2 because it allows approximate matching of timestamps and accommodate naturally occurring physiological and sensor delays.

Problem Definition:

The above scenarios demonstrate that refining queries to meet cardinality and proximity requirements is challenging. We refer to the problem of query refinement with these two goals as the **Proximity-Based Cardinality Assurance** (PBCA) problem. Given a query Q and expected cardinality C, ***PBCA*** seeks to generate a set of optimally refined queries Q_R such that:

1. Cardinality Criteria: For Q' in Q_R, Cardinality (Q') = C
2. Proximity Criteria: There exists no Q'' such that Cardinality(Q'') = C and Q'' is closer to Q than Q' is to Q.

As shown in [16], the proof in [6] can be easily extended to prove that ***PBCA*** is NP-Hard.

Background and Related Work:

While the literature contains several approaches for refining queries, none of the existing techniques simultaneously address the cardinality and proximity criteria. To

identify queries likely to produce no results, [10] proposed a methodology based on the analysis of previous query executions. In contrast, [6,13,14] introduced techniques based on ontological data and Bayesian structures to refine such queries. However, these techniques focus neither on meeting cardinality constraints nor on maintaining query proximity.

[9] proposed a method to refine queries using skyline algorithms. This approach however suffers from the following shortcomings. First, [9] can produce results with unacceptable departures from original predicates, e.g. for query Q1, [9] can return a volunteer with income of 100K (refinement=45K) solely because his BMI=31 satisfies the original predicate. Second, altering the core algorithm for cardinality assurance is expensive. Third, [9] cannot extend to the too-many results case.

Recently, [12] introduced an interactive refinement approach to ensure query cardinality. [12] limits query refinements using information about the underlying database. But, once again, manual refinement can be frustrating because users have insufficient information about the S database. Additionally, [12] cannot produce alternate queries or refine joins.

In the context of database testing, [1, 11] have proposed techniques to generate test queries satisfying cardinality constraints: [1] proposed a heuristic hill-climbing strategy, while [11] proposed techniques based on binary search and divide-and-conquer policies. However, both these techniques disregard proximity. Further, as our experiments demonstrate, repeated query execution makes them inefficient compared to QRelX.

To solve the too many results problem, [2] introduced the STOP AFTER operator. In contrast, [4,8] proposed to apply a ranking function and generate the Top-k results. Yet, ranking-based approaches have several drawbacks: (1) For each query, the user must amalgamate multiple predicates into a singular scoring function. (2) These techniques cannot refine joins.(3) Such methods produce only the refinement results and not queries characterizing these results, information which is crucial for decision support. For example, when a bank chooses customers for special offers, it must know not only the customers that were selected (the results), but also the reasons for their selection (the query).

In summary, current techniques lack the ability to simultaneously satisfy cardinality and proximity requirements for refined queries; QRelX is the first framework to address this challenging problem [15].

Uniqueness of Approach:

Since query refinement depends on cardinality and proximity, QRelX provides seamless support for addressing both concerns while minimizing overall computational expenses. QRelX minimizes the number of queries and tuples examined and ensures that queries lying close the original query are preferentially investigated. Specifically, QRelX employs a three-phase strategy: (1) **Proximity-Based Search** generates and investigates queries ordered by distance from the original query; (2) **Need-based Result Generation** minimizes tuple-level computations; and (3) **Incremental Cardinality Estimation** exploits result-sharing to reuse previous cardinality estimates.

Proximity-based Search (PBS):

To find a set of queries Q_R satisfying the refinement criteria, QRelX adopts a proximity-based approach which explores queries in order of increasing refinement. This technique eliminates the need to explore the entire set of refined queries and permits QRelX to terminate as soon as a query meeting the cardinality constraint is found.

We now define the representation of refined queries and the distance metric used to order them.

Representation:

Consider a conjunctive query Q with predicates P1, P2...Pn such that all Pi are select or join conditions of the form (R.x < bound) or (R.x=T.x). For Q, a refined query Q' is any query with predicates P'1, P'2...P'n such that each predicate (P'i=Pi) or (P'i alters the bound on Pi). Consider for example, the select queries Q_S and Q'_S given below. Q'_S is a valid refinement of Q'_S since it only changes the bounds on attributes R.x and R.y

$$Q_S = (R.x < 30) \text{ AND } (R.y < 50)$$
$$Q'_S = (R.x < 65) \text{ AND } (R.y < 55)$$

Any refined query Q' is expressed in terms of refinements made to the original query predicates. To illustrate, the query Q'_S is represented as the vector (35,5) since the first predicate has been relaxed by 35 units, while the other by 5. Similarly, the original query Q_S is expressed as (0,0). Based on this representation, the distance of a refined query Q' from Q is defined as the L_1 norm of the vector representation of Q', i.e. the sum of the predicate refinements, e.g., Q'_S is at a distance of (35+5=40) units from Q_S, while Q_S is 0 units from itself.

For brevity, we focus on predicates of the form (R.x < bound) and (R.x=T.y), and refinements where predicate bounds are increased. [16] describes how the same techniques can be extended for complex predicates such as (R.x+2*R.y < 50) or (R.x < T.y), and more flexible forms of refinement.

Algorithm:

To aid PBS in exploring queries in order of distance, we introduce a **Refined Query Space** (RQS) representing all refined queries in terms of their predicate refinement vectors. Further, since an exhaustive search of RQS would be prohibitively expensive, we impose a grid-structure and only examine queries lying on the grid. If no grid query is found to satisfy the cardinality constraint, we further partition each cell for a fine-grained search. Figure 1.a shows the RQS for Q_S.

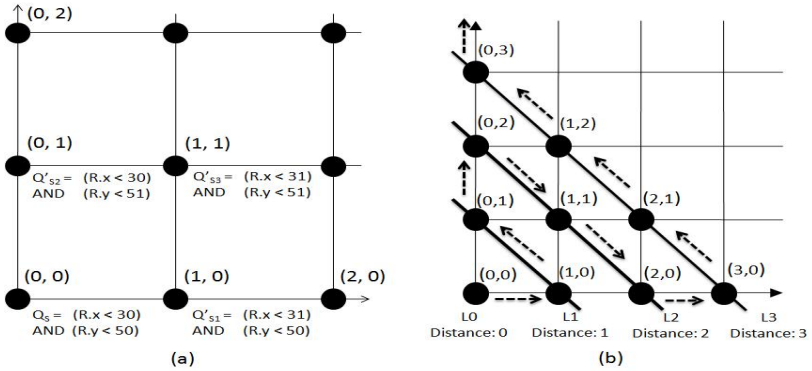


Figure 1: (a) RQS for Q'S, (b) Proximity-Based Search of RQS

Once the RQS has been created, PBS groups grid queries into layers lying at equal distance from the original query. For instance, queries Q'_{S1} and Q'_{S2} shown below are both at distance 1 from Q_S, and hence lie in the same proximity-layer.

$$Q'_{S1} = (R.x < 31)\ AND\ (R.y < 50)$$
$$Q'_{S2} = (R.x < 30)\ AND\ (R.y < 51)$$

PBS then performs a layer-based search where it sequentially explores queries in layers at increasing distance. As indicated by the arrows in Figure 1. b, the queries in L0 are investigated first, followed by those in L1, L2, L3 and so on. This traversal technique ensures that queries in layer **k** are investigated before those in layer **k+1**. Consequently, when PBS finds the first query having cardinality at least C, the search can terminate because we are guaranteed that no query lying closer to Q meets the expected cardinality.

Due to the grid structure of RQS, we are not guaranteed to find the query lying at the absolute minimum distance from Q (this would in fact need an exhaustive search of the RQS); instead we find a query that is within distance $\mathbf{d_0}$ of the absolute minimum where $\mathbf{d_0}$ is a linear function of the grid-cell size and hence a tunable parameter. Similarly, there may exist no query in RQS that exactly meets the cardinality constraint. Therefore, we allow the cardinality to vary within a threshold $\mathbf{c_0}$ of the expected cardinality C.

Need-based Result Generation:

Need-based Result Generation minimizes the tuple-level computations performed while estimating cardinality for each query traversed by PBS. Unlike the brute-force method, QRelX avoids an exhaustive evaluation of input tuples by determining beforehand the subset of tuples likely to satisfy the current query and only examining these tuples. For this purpose, QRelX constructs an abstract result space grouping tuples into *regions* based on attribute values. QRelX can then determine the regions likely to satisfy the current query and only evaluate tuples in these regions. Further, to make the result space compatible with RQS, all tuples and regions are expressed in terms of their predicate refinements, e.g., a tuple with R.x=45 and R.y=75 is expressed as (15,25) because Q_S must be refined by (15,25) units to include the given tuple. Similarly, any tuple satisfying Q_S is represented by (0,0).

As illustrated in Figure 2, QRelX follows a simple algorithm to create regions of result tuples. Consider the select-join query Q_{SJ} defined below.

$$Q_{SJ} = \texttt{SELECT * FROM R, T WHERE R.x=T.x AND T.y<=50}$$

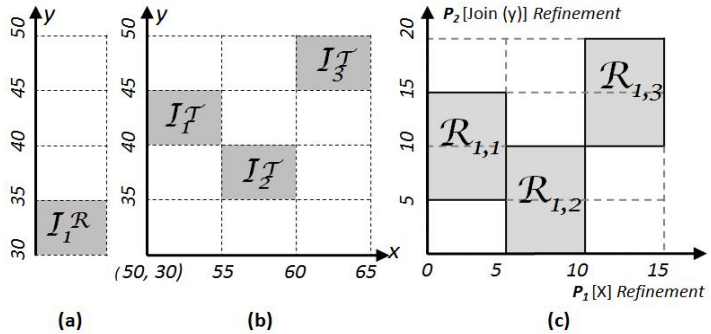


Figure 2: (a) Partitions for Table R, (b) Partitions for Table T, (c) Abstract Result Space

First, each query table is partitioned based on its attributes present in the original query. For Q_{SJ}, table R is partitioned based on R.y, while T is partitioned based on T.x and T.y (Refer Figure 2.a and b). Next, for every combination of input partitions, we compute the minimum and maximum predicate refinements for tuples belonging to these partitions. To illustrate, consider the input partitions I_1^R and I_1^T and their resulting region $R_{1,1}$ in Figure 1.c: For I_1^R, R.y lies between [30,35], while for I_1^T T.y lies between [40,45]. If a pair of tuples from these partitions was to be joined, we would have to refine the join predicate at least by 10 units (to give |R.y-T.y| <= 10) and at most by 15. Similarly, since T.x lies between [50,55], we would have to refine the select predicate by at least 0 and at most 5 units. The minimum and maximum refinements respectively determine the bounds of $R_{1,1}$.

Once all regions in the abstract space have been determined, we overlay the refined query space and abstract result space point for point (possible because both these space are represented as predicate refinements). Figure 3.a shows this overlay for query Q_{SJ}. **The cardinality of a refined query Q'_{SJ} represented as (x', y') in the hybrid space is the number of tuples lying inside the orthotope extending from the origin to (x',y'). We call this orthotope the *query orthtope***. To illustrate, in Figure 3.b, the query orthotope for Q'_{SJ1} is $OF_1Q'_{SJ1}E_1$, and that for Q'_{SJ2} is $OF_2Q'_{SJ2}E_2$. For evaluating the cardinality of query Q'_{SJ2}, we determine the regions lying partially or fully inside its orthotope and only examine tuples from these regions. Thus, we are guaranteed that no tuple-level computation is performed until a region is

found to contribute to the query being currently investigated.

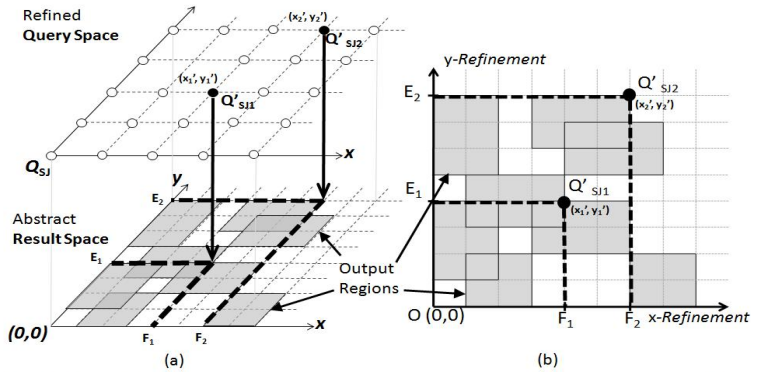


Figure 3: (a) Overlay of RQS and abstract result space for Q_{SJ}, (b) Query orthotopes for Q'_{SJ1} and Q'_{SJ2}

Incremental Cardinality Estimation (ICE):

While need-based query result generation can successfully reduce tuple-level computations during query evaluation, it requires that each refined query be executed independently. Thus, even if a tuple has been found to satisfy some query Q', it is re-evaluated for all other refined queries in the RQS. To address this drawback, QRelX introduces a novel incremental cardinality estimation (ICE) scheme that exploits result-sharing between previously investigated queries and significantly reduces computational expenses. ICE ensures that once a result tuple has been evaluated for one query, it is never re-evaluated for any other query. The following principle guides our incremental scheme:

Query Containment: A query Q' is said to be contained in query Q'' if the query orthotope of Q' is completely contained within that of Q''. For instance, the query Q'_{SJ1} in Figure 3.b is contained in query Q'_{SJ2} since $OF_1Q'_{SJ1}E_1$, the orthotope of Q'_{SJ1}, is contained within $OF_2Q'_{SJ2}E_2$, the orthotope of Q'_{SJ2}. Containment implies that once we determine the tuples lying in the orthotope of Q'_{SJ1}, we are not required to re-evaluate these tuples for Q'_{SJ2}; we must only examine the tuples lying outside $OF_1Q'_{SJ1}E_1$ but within $OF_2Q'_{SJ2}E_2$. Further, we observe that since query $Q'_{SJ1}=(x'_1,y'_1)$ is contained within $Q'_{SJ2}=(x'_2,y'_2)$: ($x'_1<=x'_2$), ($y'_1<=y'_2$), and Q'_{SJ1} is closer to Q_{SJ} than Q'_{SJ2} is to Q_{SJ}. Thus, Q'_{SJ1} lies in a closer proximity-layer than Q'_{SJ2} and will be examined before Q'_{SJ2}. PBS therefore guarantees that before any refined query Q'' is traversed, all queries Q' contained in Q'' will already be traversed, and we can reuse their cardinality information through the sharing of results.

To understand result-sharing, reconsider the query orthotope associated with every refined query Q'. We construct a set of queries that are contained in Q' and can be combined to constitute the entire query orthotope. In an n-dimensional RQS, we construct (n+1) such sub-queries. For ease of exposition, we describe sub-queries through their query orthotopes. Figures 4 and 5 show the set of sub-queries constructed for a 2-D and 3-D RQS respectively. The first sub-query (A) is a unit "cell" (square in 2-D and cube in 3-D) with Q' at its upper-right corner; the second (B) is a unit-width "pillar" (rectangle in 2-D and unit length-and-width parallelopiped in 3-D) with Q' at its upper-right corner; the third (C) is a "wall" that is the entire query orthotope in 2-D and a unit width parallelepiped in 3-D; and the fourth (D) is a "block" covering the entire query orthotope for 3-D.

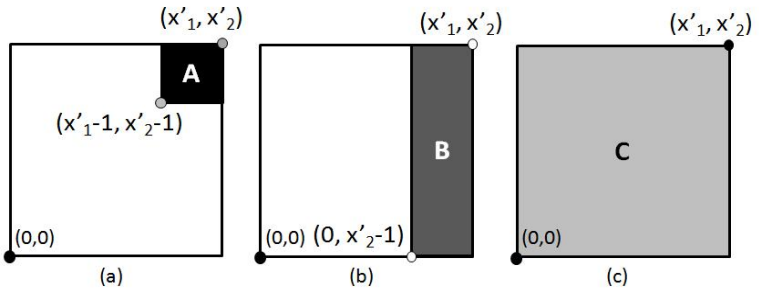


Figure 4: (a) Cell orthotope in 2-D, (b) Pillar orthotope in 2-D, (c) Wall orthotope in 2-D

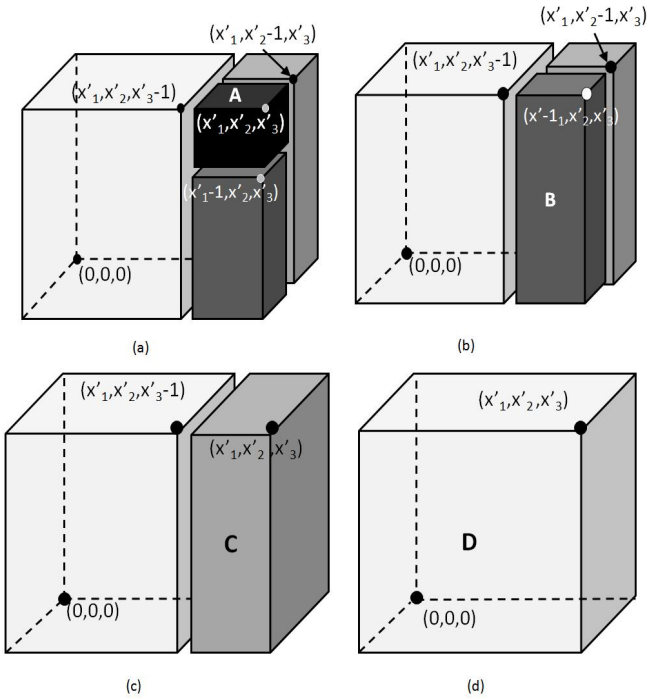


Figure 5: (a) Cell orthotope in 3-D, (b) Pillar orthotope in 3-D, (c) Wall orthotope in 3-D, (d) Block orthotope in 3-D

For n dimensions, the n+1 orthotopes ($O_1 ... O_{n+1}$) can be formally defined in terms of their lower- and upper-bounds. The upper-bound for all orthotopes is the query under consideration. The "cell" (O_1) has a lower-bound which is a unit length away from the upper-bound on all dimensions. The "pillar" (O_2) has a lower-bound with the first dimension equal to 0 and all others one unit away from the upper-bound. For the j-th orthotope, the first j-1 dimensions of the lower-bound are 0 while the remaining are a unit away from the upper-bound. O_{n+1} is the complete query orthotope. Formally:

$$1.\ O_1 = ((x_1-1, x_2-1 \ldots x_d-1) : (x_1, x_2 \ldots x_d))$$
$$2.\ O_2 = ((0, x_2-1 \ldots x_d-1) : (x_1, x_2 \ldots x_d))$$
$$3.\ O_j = ((0, 0 \ldots x_j-1 \ldots x_d-1) : (x_1, x_2 \ldots x_d))$$
$$4.\ O_{n+1} = ((0, 0, \ldots 0) : (x_1, x_2 \ldots x_d))$$

To exploit result sharing opportunities, we decompose every query orthotope in terms of the sub-orthotopes defined above. For example, Figure 6 shows the decomposition of a 2-D query orthotope into 3 sub-query orthotopes: (1) "cell" orthotope with upper bound (x'_1,x'_2) (A), (2) a "pillar" orthotope with the upper bound (x'_1-1,x'_2) (B), and (3) a "wall" orthotope with upper bound (x'_1,x'_2-1) (C). Similarly, Figure 7 shows the decomposition of a 3-d query orthotope into 4 sub-orthotopes "cell", "pillar", "wall" and "block".

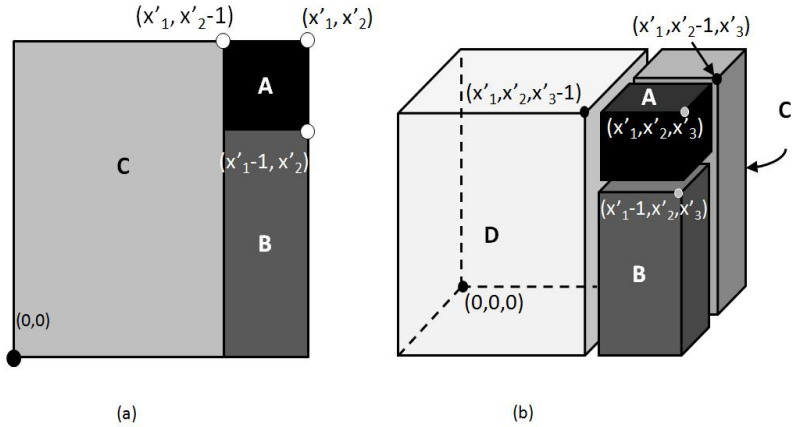


Figure 6: (a) Decomposition of a 2-D orthotope, (b) Decomposition of a 3-D orthotope

In general, an n-dimensional orthotope requires (n+1) orthotopes to completely cover it:

1. 2-d Query Orthotope:
   $O_3(x',y') = O_1(x',y') + O_2(x'-1,y') + O_3(x',y'-1)$
2. 3-d Query Orthotope:
   $O_4(x',y',z') = O_1(x',y',z') + O_2(x'-1,y',z') + O_3(x',y'-1,z') + O_4(x',y',z'-1)$
3. n-dimensional Query Orthotope:
   $O_{n+1}(x'_1,x'_2...x'_n) = O_1(x'_1,x'_2...x'_n) + O_2(x'_1-1,x'_2...x'_n) + O_3(x'_1, x'_2-1...x'_n) + O_{n+1}(x'_1, x'_2...x'_n-1)$

Thus, if we pre-computed cardinalities of the (n+1) sub-orthotopes, cardinality of the query Q' is the mere addition of their cardinalities. Further, we observe that the only part of the query orthotope unique to a query is the "cell"; all other parts are shared with previously investigated queries. Thus, in terms of the abstract result space, for every query, we must only evaluate tuples from regions lying in the "cell" orthotope.

The orthotope decompositions described above assume that we have already computed the cardinalities of the (n+1) sub-queries. Therefore, we introduce a novel recursive methodology to calculate the cardinalities of all sub-queries in constant time. Reconsider Figure 6, and observe the relationship between sub-orthotopes. For the 2-d orthotope, the Pillar(x'_1,x'_2) is the sum of the Cell(x'_1,x'_2) and Pillar(x'_1-1,x'_2). Similarly, the Wall(x'_1,x'_2) is the sum of the Pillar(x'_1,x'_2) and Wall(x'_1,x'_2-1). We find similar recurrences in 3-D. In general:

2-D Recurrences:
1. $\text{Pillar}(x'_1, x'_2) = \text{Cell}(x'_1, x'_2) + \text{Pillar}(x'_1-1, x'_2)$
2. $\text{Wall}(x'_1, x'_2) = \text{Pillar}(x'_1, x'_2) + \text{Wall}(x'_1, x'_2-1)$

3-d Recurrences:
1. $\text{Pillar}(x'_1, x'_2, x'_3) = \text{Cell}(x'_1, x'_2, x'_3) + \text{Pillar}(x'_1-1, x'_2, x'_3)$
2. $\text{Wall}(x'_1, x'_2, x'_3) = \text{Pillar}(x'_1, x'_2, x'_3) + \text{Wall}(x'_1, x'_2-1, x'_3)$
3. $\text{Block}(x'_1, x'_2, x'_3) = \text{Wall}(x'_1, x'_2, x'_3) + \text{Block}(x'_1, x'_2, x'_3-1)$

d-dimensional Recurrences:
1. $O_i(x'_1...x'_d) = O_{i-1}(x'_1...x'_d) + O_i(x'_1, x'_i-1...x'_d)$

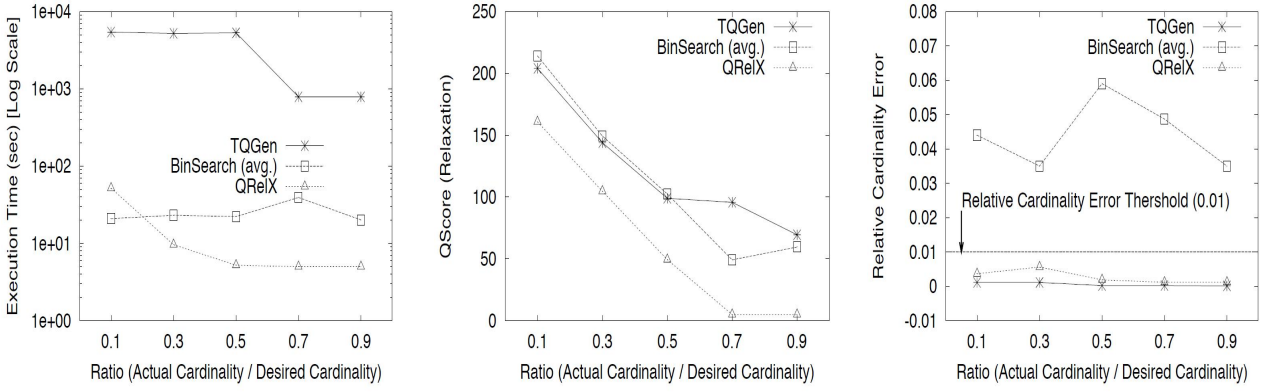
Thus, to calculate the cardinality of any query, we first determine the cardinality of its "cell" sub-orthotope (by examining the regions overlapping it) and then use the above recurrences to compute its full cardinality in a constant number of steps.

The QRelX Algorithm:

Overall, the QRelX algorithm proceeds in three steps: (1) An RQS is created and proximity-based search is initiated; (2) For every query, need-based result generation and incremental cardinality estimation are used to efficiently compute query cardinality; (3) The calculated cardinality is compared with C to determine if it falls within the tolerance threshold $\mathbf{c_0}$. If so, QRelX explores other queries at the same distance and then terminates its search. If a query produces too few results, QRelX proceeds to other queries in that layer and successive layers. In contrast, if a query produces too many results, QRelX repartitions the appropriate grid cells and performs a fine-grained search.

Results and Conclusion:

We studied the efficacy of QRelX on the TPC-H benchmark data and synthetic data. We used a suite of sixty queries to run comparative studies with the binary search (BinSearch) and TQGen algorithms proposed in [6]. BinSearch sequentially performs binary searches on each predicate to find a query minimizing the cardinality error. TQGen, in contrast, uses a divide-and-conquer policy. We note that unlike QRelX, neither of these algorithms can refine join predicates. All the above systems were implemented in Java and measurements were obtained on a workstation with AMD 2.6GHz Dual Core CPUs.



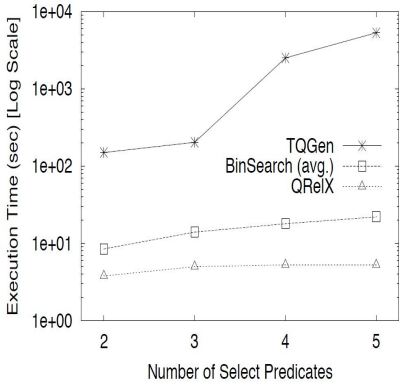


Figure 7: Comparison of (a) execution time, (b) refinement, (c) relative cardinality error for varying C_{actual}/C ratios. (d) Comparison of execution time for varying number

Refining Only Select Predicates:

Figures 7.a, b, c respectively show the average execution time, cardinality error and amount of refinement for queries produced by all three techniques. These measurements were taken for a single table with 100k tuples and queries with 5 predicates. We vary the value of the ratio C_{actual}/C, where C_{actual}=Cardinality of original query, because the ratio quantifies the relative amount of refinement. In Figure 7.a we observe that QRelX performs 2-3 orders of magnitude faster than TQGen and 77% faster than BinSearch. The cardinality error in Figure 7.b indicates that both TQGen and QRelX are within the cardinality error threshold=0.01%, but BinSearch gives unpredictable results because it depends on the order of predicate refinement. Finally, in Figure 7.c, we see that QRelX has on average 25% lower refinement than other techniques, thus outperforming BinSearch and TQGen for all three metrics.

Figure 9.d shows the behavior of the three methods for queries with varying number of predicates (C_{actual}/C=0.5). We observe that the execution time for TQGen increases exponentially because of a proportional increase in the number of queries executed. BinSearch, in contrast, compares favorably to QRelX, but suffers from the drawback of repeated query execution. Increase in the number of query predicates does not affect QRelX significantly because it employs need-based result generation and incremental cardinality estimation.

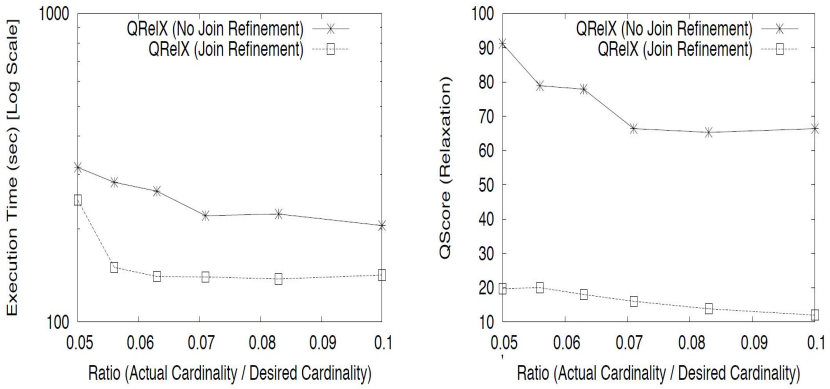


Figure 8: Comparison of (a) execution time and (b) refinement for QRelX join refinement enabled and disabled

Refining Both Join and Select Predicates

We compared the effect of allowing and disallowing QRelX join refinement for queries on synthetic data (two tables with 10k tuples each and join selectivity=0.01). As shown in Figures 11.a and b, allowing join refinement reduces the total refinement by up to 80% and affords a speedup of 20% for small values (0.05-0.1) of C_{actual}/C. Join refinement is thus demonstrated to be an attractive option when the original query is far from the expected cardinality.

Conclusion:

In this work, we investigated the problem of proximity-based cardinality assurance (PBCA) and proposed QRelX, the first framework to generate refined queries that meet the cardinality constraint and lie close to the original query. QRelX employs a combination of Proximity-Based Search, Need-Based Result Generation, and Incremental Cardinality Estimation to attain these twin goals. Our experiments demonstrate that compared to other techniques, QRelX leads to a speed-up of up to 2-3 orders of magnitude and generates refined queries with at least 25% smaller refinements.

Acknowledgements:

I'd like to thank Venkatesh Raghavan, collaborator, and Prof. Elke Rundensteiner, advisor for this project, for their contribution to the work. This work was made possible by the CRA-W CREU Grant for 2009-10.

References:

1. N. Bruno, et al., "Generating Queries with Cardinality Constraints for DBMS Testing," IEEE TKDE, vol. 18, no. 12, pp. 1721-1725, 2006.
2. M. J. Carey and D. Kossmann, "On Saying 'Enough Already!' in SQL," in SIGMOD, 1997, pp. 219-230.
3. S. Chaudhari and L. Gravano, "Evaluating Top-k Selection Queries," in VLDB, 1999, pp. 397-410.
4. S. Chaudhuri and G. Das, "Automated Ranking of Database Query Results," in CIDR, 2003, pp. 888-899.
5. S. Chaudhuri, et al., "Robust Cardinality and Cost Estimation for Skyline Operator," in ICDE, 2006, pp. 64-73.

6. T. Gaasterland, "Cooperative Answering through Controlled Query Relaxation," IEEE Expert: Intelligent Systems and Their Applications, vol. 12, no. 5, pp. 48-59, 1997.
7. I. F. Ilyas, et al., "Supporting Top-k Join Queries in Relational Databases," in VLDB, 2003, pp. 754-765.
8. A. Kadlag, et al., "Cardinality Estimation using Sample Views with Quality Assurance," in DAFSAA, 2004, pp. 594-605.
9. N. Koudas, et al., "Relaxing Join and Selection Queries," in VLDB, 2006, pp. 199-210.
10. G. Luo, "Efficient Detection of Empty-result Queries," in VLDB, 2006, pp. 1015-1025.
11. C. Mishra, et al., "Generating Targeted Queries for Database Testing," in SIGMOD, 2008, pp. 499-510.
12. C. Mishra and N. Koudas, "Interactive Query Refinement," in EDBT, 2009, pp. 862-873.
13. I. Muslea and T. Lee, "Online Query Relaxation via Bayesian Causal Structures Discovery," in AAAI, 2005, pp. 831-836.
14. I. Muslea, "Online Query Relaxation," in SIGKDD, 2004, pp. 246-255.
15. M. Vartak, et al., "QRelX: Generating Meaningful Queries that Provide Cardinality Assurance," in SIGMOD (Demonstration), 2010.
16. M. Vartak, et al., "Query-oriented Refinement for Cardinality Assurance", (in submission)