8-2013

Computing Immutable Regions for Subspace Top-k Queries

Kyriakos MOURATIDIS
Singapore Management University, kyriakos@smu.edu.sg

Hwee Hwa PANG
Singapore Management University, hhpang@smu.edu.sg

DOI: https://doi.org/10.14778/2535568.2448941

Computing Immutable Regions for Subspace Top-k Queries*

Kyriakos Mouratidis
School of Information Systems
Singapore Management University
kyriakos@smu.edu.sg

HweeHwa Pang
School of Information Systems
Singapore Management University
hhpang@smu.edu.sg

ABSTRACT

Given a high-dimensional dataset, a top-k query can be used to shortlist the k tuples that best match the user's preferences. Typically, these preferences regard a subset of the available dimensions (i.e., attributes) whose relative significance is expressed by user-specified weights. Along with the query result, we propose to compute for each involved dimension the maximal deviation to the corresponding weight for which the query result remains valid. The derived weight ranges, called *immutable regions*, are useful for performing sensitivity analysis, for finetuning the query weights, etc.

In this paper, we focus on top-k queries with linear preference functions over the queried dimensions. We codify the conditions under which changes in a dimension's weight invalidate the query result, and develop algorithms to compute the immutable regions. In general, this entails the examination of numerous non-result tuples. To reduce processing time, we introduce a pruning technique and a thresholding mechanism that allow the immutable regions to be determined correctly after examining only a small number of non-result tuples. We demonstrate empirically that the two techniques combine well to form a robust and highly resource-efficient algorithm. We verify the generality of our findings using real high-dimensional data from different domains (documents, images, etc) and with different characteristics.

1. INTRODUCTION

Consider dataset $\mathcal{D}$ where every tuple $\mathbf{d}_\alpha$ is a vector $\langle d_{\alpha 1}, d_{\alpha 2}, \ldots, d_{\alpha m}\rangle$ in an m-dimensional space $[0,1]^m$. Specified a query vector $\mathbf{q} = \langle q_1, q_2, \ldots, q_m\rangle$ in this space, the score of a tuple $\mathbf{d}_\alpha$ is defined as the dot product $\mathcal{S}(\mathbf{d}_\alpha, \mathbf{q}) = \mathbf{q} \cdot \mathbf{d}_\alpha$. The top-$k$ result $\mathcal{R}(\mathbf{q})$ of this query is a list of the k tuples with the highest scores in $\mathcal{D}$. An update to any query weight q_j may induce a change in the composition of $\mathcal{R}(\mathbf{q})$ or the ordering among its members, in which case we say that $\mathcal{R}(\mathbf{q})$ is perturbed; otherwise $\mathcal{R}(\mathbf{q})$ is preserved. For each dimension $j \in [1, m]$, we define the *immutable region* as the widest range of q_j values that preserve $\mathcal{R}(\mathbf{q})$, assuming that all other weights q_i for $i \neq j$ are kept constant.

*Supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office.

Published in Proceedings of the VLDB Endowment:
39th VLDB 2013, August 26-30, Riva del Garda,
Trento, Italy.
http://www.vldb.org/pvldb/vol6/p73-mouratidis.pdf

Figure 1: Running Example

To illustrate, consider dataset $\mathcal{D} = \{\mathbf{d_1}, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_4\}$ in Figure 1, which is indexed by inverted lists $\mathcal{L}_1$ and $\mathcal{L}_2$. Each tuple $\mathbf{d}_\alpha = \langle d_{\alpha 1}, d_{\alpha 2}\rangle$ is a vector in two-dimensional space, and the score of $\mathbf{d}_\alpha$ with respect to a query vector $\mathbf{q} = \langle q_1, q_2\rangle$ is given by function $\mathcal{S}(\mathbf{d}_\alpha, \mathbf{q}) = \sum_{j=1}^{2} q_j \times d_{\alpha j}$. For $\mathbf{q} = \langle 0.8, 0.5\rangle$, the top-2 result is $[\mathbf{d}_2, \mathbf{d}_1]$. Any change in query weight q_1 inside the range $(q_1 - \frac{16}{35}, q_1 + 0.1)$ has no effect on the result (provided that q_2 retains its value). As this is the widest range where this holds, we call it the immutable region for q_1. We define the immutable region for query weight q_2 similarly; that is $(q_2 - \frac{1}{18}, q_2 + 0.5)$.

An application of immutable regions is iterative query refinement. In a text retrieval setting, for example, many document and Web search engines are based on the vector space model [1]. Documents and queries are represented as vectors in term-space, and the similarity between a document and a user query is approximated by their cosine distance, equivalent to the dot product of their respective vectors. If a query result does not satisfy the user's information needs, she may iteratively adjust the weights in the query vector (e.g., raise the emphasis on a certain keyword). Naturally, the user wants to avoid trying several minuscule adjustments that produce no visible impact on the result. On the other hand, neither does she (always) want to perform a large jump in a query weight that alters the result entirely. Instead, it would be useful to compute the immutable regions for each dimension, thus specifying to the user the exact weight values where the result changes. Figure 1 illustrates a slide-bar interface to control weights q_1 and q_2. The horizontal marks at positions l_j and u_j on each bar indicate the immutable region for the corresponding dimension. The marks at l_1 and u_1 on the left slide-bar could be used to guide an adjustment to q_1. Likewise for q_2 and positions l_2, u_2 on the right slide-bar.

Another application is sensitivity analysis. Consider a system like tripadvisor.com, where users may browse accommodation options (e.g., hotels) in a city they plan to visit. In addition to price per night, the system also maintains for each hotel its average scores on cleanliness, value for money, convenience of location, service, etc, based on guest reviews. Assume that a user's decision criteria

are price per night, cleanliness, and service. A common decision making strategy is to specify a weight for each criterion of interest, and apply a linear scoring function on the corresponding hotel attributes in order to shortlist the k most preferred options. Along with the top-k recommendation, it would also be useful to provide a sensitivity measure on the decision variables. In this context, the immutable regions serve to profile the robustness of the recommendation to deviations in the stated user preferences. A narrow immutable region for cleanliness and a wide one for service suggest that the result is more sensitive to the former. Thus, a compromise (or raise of standards) in cleanliness is more likely to alter the top-k recommendation than reconsidering service expectations.

Immutable regions, as defined so far, indicate weight ranges within which no perturbation (i.e., reordering, or inclusion of a hitherto non-result tuple) occurs in $\mathcal{R}(\mathbf{q})$. A generalization is to compute regions for up to ϕ perturbations in $\mathcal{R}(\mathbf{q})$, for some $\phi \geq 0$. Consider the example in Figure 1 again, and assume that $\phi = 1$. As explained previously, with $\phi = 0$ the immutable region for query weight q_1 is $(q_1 - \frac{16}{35}, q_1 + 0.1)$, which preserves the top-2 result $[\mathbf{d}_2, \mathbf{d}_1]$. If q_1 moves to the right of this range and into $(q_1 + 0.1, q_1 + 0.2)$, the result changes to $[\mathbf{d}_1, \mathbf{d}_2]$. On the other hand, if q_1 moves into $(q_1 - 0.55, q_1 - \frac{16}{35})$, $\mathcal{R}(\mathbf{q})$ becomes $[\mathbf{d}_2, \mathbf{d}_3]$. Keeping q_1 within the union of ranges $(q_1 - 0.55, q_1 - \frac{16}{35}) \cup [q_1 - \frac{16}{35}, q_1 + 0.1] \cup (q_1 + 0.1, q_1 + 0.2)$ ensures that there are no more than $\phi = 1$ perturbation in $\mathcal{R}(\mathbf{q})$, as long as q_2 is fixed. Furthermore, the exact query result in each of these ranges is also available.

Contributions: We consider subspace top-k queries. These queries score tuples in $\mathcal{D}$ via a linearly weighted function on the tuples' coordinates (values) in a subset of the data dimensions (attributes). Such queries are common in high-dimensional datasets [17, 23]. We codify the conditions under which deviations in a query weight invalidate the top-k result, and formulate immutable regions based on these conditions. We collectively refer to our problem as immutable region computation, but our goal is to additionally report the specific perturbations (i.e., new query results) at the bounds of the immutable regions. We observe that the cost of computing immutable regions is dominated by the examination of non-result data (i.e., tuples in $\mathcal{D} \backslash \mathcal{R}(\mathbf{q})$) to ensure that none of them supplant the current result tuples. To reduce this cost, we introduce the Candidate Pruning and Thresholding Algorithm (CPT). CPT incorporates two complementary techniques – pruning and thresholding.

The first technique, pruning, builds on the insight that when a query weight varies, there exist two subsets of $\mathcal{D} \backslash \mathcal{R}(\mathbf{q})$ in which the tuples always maintain their relative score order. We prove that only a small number of the leading candidates in the two subsets are capable of influencing the immutable regions, thus allowing the remaining tuples to be eliminated from consideration. The number of leading candidates needed is determined by ϕ, and is independent of k, $|\mathcal{D} \backslash \mathcal{R}(\mathbf{q})|$, and the data/query dimensionality.

The second technique, thresholding, assesses the 'potential' of candidates to qualify for $\mathcal{R}(\mathbf{q})$ due to weight changes. It processes them in descending order of potential, and terminates when all remaining candidates are guaranteed not to enter the result in the current immutable region. The challenge here lies in quantifying the potential, as well as in deriving a safe termination condition that will allow un-processed candidates to be disregarded.

Using both synthetic and real datasets (of different types and characteristics), we evaluate CPT under various settings. The results confirm the effectiveness of the pruning and thresholding techniques, and show that combined they reduce the number of examined tuples by 2 to well over 500 times compared to a baseline approach. This leads to vast improvements in I/O and CPU cost.

TA operation	t_1	t_2	threshold	$\mathcal{R}(\mathbf{q})$	$\mathcal{C}(\mathbf{q})$
1. Initialization	0.8	0.8	1.04	[ ]	[ ]
2. Process d_1 on $\mathcal{L}_1$; $\mathcal{S}(\mathbf{d}_1, \mathbf{q}) = 0.8$	0.7	0.8	0.96	$[d_1]$	[ ]
3. Process d_3 on $\mathcal{L}_2$; $\mathcal{S}(\mathbf{d}_3, \mathbf{q}) = 0.48$	0.7	0.6	0.86	$[d_1, d_3]$	[ ]
4. Process d_2 on $\mathcal{L}_1$; $\mathcal{S}(\mathbf{d}_2, \mathbf{q}) = 0.81$	0.1	0.6	0.38	$[d2, d_1]$	$[d_3]$
5. $\mathcal{S}(\mathbf{d}_1, \mathbf{q}) \geq \mathcal{S}(\mathbf{t}, \mathbf{q})$; Termination					

Figure 2: TA Execution in Running Example

2. RELATED WORK

Top-k Search: Given a collection $\mathcal{D}$ of m-dimensional tuples $\mathbf{d}_\alpha$ and a scoring function $\mathcal{S}$, a top-k query returns the k tuples in $\mathcal{D}$ with the highest scores $\mathcal{S}(\mathbf{d}_\alpha, \mathbf{q})$. Top-$k$ queries have been studied in the context of relational databases [12], as well as similarity search in multimedia repositories [6], ranking in the presence of expensive predicates [4], joins [25, 11], uncertain or probabilistic data [10, 14], etc. To accelerate processing, several approaches have been proposed, including pre-computation [5, 9] and indexing [22, 24]. Methods also exist for maintaining the top-k result over dynamic datasets [27, 15], where (unlike our setting) the data are updated but not the scoring function. In reverse top-k queries [26], given a set of scoring functions, the problem is to determine for a specific tuple which of the functions would include it in their result.

Among top-k methods, we elaborate on the *threshold algorithm* (TA) [8] for queries with a monotone function $\mathcal{S}$, e.g., of the form $\mathcal{S}(\mathbf{d}_\alpha, \mathbf{q}) = \mathbf{q} \cdot \mathbf{d}_\alpha$ for data vector (tuple) $\mathbf{d}_\alpha$ and query vector $\mathbf{q}$. In TA, m lists keep $\mathcal{D}$ sorted with respect to each of the m dimensions in descending order. TA probes the lists (*sorted access* from top to bottom) in a round-robin fashion. For each tuple $\mathbf{d}_\alpha$ encountered in a list, its complete vector is fetched via *random access* (either in the remaining lists or in an external file holding the entire data vectors) to compute its score $\mathcal{S}(\mathbf{d}_\alpha, \mathbf{q})$. The k tuples with the highest scores encountered during processing are kept, in descending order of their scores, in a (tentative) result $\mathcal{R}(\mathbf{q})$. Let t_i be the sorting key of the next tuple in the i-th list. The search terminates when the k-th score in the result is no smaller than the score of the (fictitious) tuple $\mathbf{t} = \langle t_1, t_2, ..., t_m \rangle$, which plays the role of a threshold. On termination, $\mathcal{R}(\mathbf{q})$ contains the top-k result. Figure 2 traces the execution of TA in the example of Figure 1. List $\mathcal{C}(\mathbf{q})$ contains all non-result tuples encountered by the algorithm – it is not used by TA but its role will be discussed shortly. This corresponds to *random access* TA (we focus on this variant due to its superior performance over the *no random access* version).

In existing literature, the closest work to our problem is [20]. Although not its main objective, that study formulates a side-problem (termed STB) where, given a query vector $\mathbf{q}$ and a dataset $\mathcal{D}$, the goal is to compute the maximal radius ρ around $\mathbf{q}$ (in the query vector space) where the top-k result remains the same. The radius ρ is used as a measure for sensitivity analysis. Consider the two-dimensional vector $\mathbf{q} = \langle q_1, q_2 \rangle$ in Figure 3, and assume that $k = 1$. Each non-result tuple $\mathbf{d}_\beta$ defines a half-plane in the query vector space where $\mathbf{d}_\beta$ scores higher than the current result tuple. Radius ρ is the smallest distance between $\mathbf{q}$ and any of these half-planes, essentially defining a circle where the result is preserved.

Our formulation, on the contrary, isolates each query weight and devises an immutable region for it, while all other weights remain fixed; in a real multivariate preference application, the user would normally consider one weight adjustment at a time. Figure 3 illustrates the immutable regions IR_1 and IR_2 on the two query dimen-

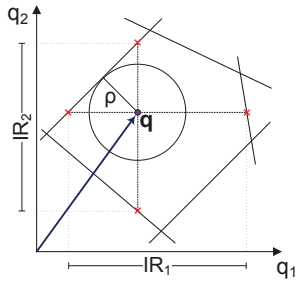


Figure 3: Query Vector Space

sions. STB does not cater for iterative query refinement – moving $\mathbf{q}$ outside the circle with radius ρ in Figure 3 does not necessarily induce a result perturbation. Our framework not only computes the exact immutable regions per query dimension, but it also outputs the new result past them. Furthermore, it supports more than one refinement, reporting all immutable regions and respective results for up to $\phi > 0$ perturbations. Finally, STB requires scanning all non-result tuples to compute the imposed half-planes (or half-spaces in higher dimensions). This is similar to the baseline solution to our problem (in Section 4) and leads to poor performance. Our advanced techniques reduce the number of processed non-result tuples and achieve a vast speed-up.

Given the visualization in Figure 3, one could suggest computing the exact polyhedron in query space that bounds the validity of the current result. Although this is possible in two or three dimensions, the complexity of half-space intersection explodes with dimensionality. In m dimensions the complexity of the polyhedron is $\Omega(n^{\lfloor m/2 \rfloor})$, where n is the dataset cardinality [2]. This implies not only a prohibitive computation cost, but also an inability to effectively construct (not to mention visualize) the faces of the polyhedron. We therefore isolate query dimensions, and derive immutable regions for each of them individually.[1]

Safe Regions: Safe region techniques are used in moving object databases in order to avoid frequent index maintenance and result re-computation, and to reduce the communication overhead imposed by location updates. Given a set of queries (e.g., ranges), each data object has an associated safe region. As long as the object remains within this spatial region, it is guaranteed not to alter the result of any query. An object issues a location update to the processing server only if it exits its safe region [19]. In the context of moving nearest neighbor (NN) queries, while the query point (typically, the querying user) remains within the Voronoi cell of its current NN, no query re-computation is necessary [28, 16]. An alternative approach to achieve the same goal is to provide the user with more NNs than requested, so that she can locally update the result without contacting the server [21]. Safe region techniques are tailored to spatial queries and are inapplicable to top-k processing.

3. PROBLEM DEFINITION

The problem input includes a dataset $\mathcal{D}$, a query vector $\mathbf{q}$, and a parameter k. Each tuple $\mathbf{d}_\alpha \in \mathcal{D}$ is a vector $\langle d_{\alpha 1}, d_{\alpha 2}, \ldots, d_{\alpha m} \rangle$ in m-dimensional space $[0,1]^m$. The query vector comprises m weights $q_j \in [0,1]$, where $j \in [1,m]$. In high-dimensional datasets, user preferences typically involve a subset of the dimensions, hence most of the query weights are expected to be zero. Without loss of generality and for ease of presentation, we assume that $q_j > 0$ for $j \in [1, qlen]$, and $q_j = 0$ for $j \in (qlen, m]$, for some $qlen$ smaller than m. We call $qlen$ the *query length*, and the dimensions with non-zero weights the *query dimensions*. The score of $\mathbf{d}_\alpha$ with respect to $\mathbf{q}$ is given by the dot product $\mathcal{S}(\mathbf{d}_\alpha, \mathbf{q}) = \mathbf{q} \cdot \mathbf{d}_\alpha$.

[1]Our immutable regions could support concurrent modifications in multiple weights. Referring to Figure 3, it can be easily seen that the convex hull of the axis-parallel projections of $\mathbf{q}$ on the surface of the validity-preserving polyhedron (shown as red crosses) lies fully inside the polyhedron and, thus, preserves the result (albeit, being only a subpart of the polyhedron). The immutable region for each dimension essentially defines the two respective projections.

Symbol	Meaning
$\mathcal{D}$	Database (set of all data tuples)
$\mathbf{d}_\alpha$	A tuple $\langle d_{\alpha 1}, d_{\alpha 2}, \ldots, d_{\alpha m} \rangle$ in $[0,1]^m$ space
$\mathbf{q}$	A query vector $\langle q_1, q_2, \ldots, q_m \rangle$ in $[0,1]^m$ space
$qlen$	# of query dimensions (i.e., # of non-zero weights)
q_j	Weight of j-th dimension
$\mathcal{S}(\mathbf{d}_\alpha, \mathbf{q})$	Score of $\mathbf{d}_\alpha$ with respect to $\mathbf{q}$
$\mathcal{R}(\mathbf{q})$	List of top-k result tuples for $\mathbf{q}$, in decreasing score
$\mathcal{C}(\mathbf{q})$	List of candidate tuples, in decreasing score
$\mathcal{L}_j$	Inverted list on j-th dimension
$\mathcal{IR}_j$	Immutable region for j-th dimension
l_j, u_j	Lower and upper bound of $\mathcal{IR}_j$
ϕ	Number of tolerable result permutations

Table 1: Notation

Since tree indices fail in high dimensionality [24], we create an inverted list $\mathcal{L}_j$ for each dimension. $\mathcal{L}_j$ contains entries of the form $\langle d_\alpha, d_{\alpha j} \rangle$ where $d_{\alpha j}$ is the j-th coordinate of tuple $\mathbf{d}_\alpha$, and d_α is a pointer into an external file that contains the entire $\mathbf{d}_\alpha$ tuple. $\mathcal{L}_j$ is sorted in decreasing $d_{\alpha j}$ order. The inverted lists and the external file of tuples are stored on disk. Posed a query with vector $\mathbf{q}$, the threshold algorithm (random access TA, described in Section 2) is used to compute the result $\mathcal{R}(\mathbf{q})$ that comprises the k tuples with the highest scores. $\mathcal{R}(\mathbf{q})$ is a list $[\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_k]$ sorted in decreasing score order. Unlike the conventional TA, we keep in a *candidate list* $\mathcal{C}(\mathbf{q})$ all the tuples encountered but not included in the top-k result, in decreasing score order. Figure 2 illustrates the formation of $\mathcal{C}(\mathbf{q})$ as TA executes.

Our problem is to derive for each query dimension $j \in [1, qlen]$ an *immutable region* $\mathcal{IR}_j$. $\mathcal{IR}_j$ is defined as the widest range of q_j values that preserve $\mathcal{R}(\mathbf{q})$, assuming that all remaining weights q_i for $i \neq j$ are fixed. This means that (1) every pair of consecutive $\mathbf{d}_\alpha, \mathbf{d}_{\alpha+1}$ tuples in $\mathcal{R}(\mathbf{q})$ continues to satisfy the condition $\mathcal{S}(\mathbf{d}_\alpha, \mathbf{q}) \geq \mathcal{S}(\mathbf{d}_{\alpha+1}, \mathbf{q})$, $1 \leq \alpha < k$; and (2) for every $\mathbf{d}_\beta \in \mathcal{D} \backslash \mathcal{R}(\mathbf{q})$, the condition $\mathcal{S}(\mathbf{d}_k, \mathbf{q}) \geq \mathcal{S}(\mathbf{d}_\beta, \mathbf{q})$ still holds. Henceforth, for simplicity, we represent $\mathcal{IR}_j$ relative to q_j, i.e., in terms of the deviation δq_j. For instance, the immutable region $(q_1 - \frac{16}{35}, q_1 + 0.1)$ in Figure 1 is expressed as $(-\frac{16}{35}, 0.1)$. Each immutable region $\mathcal{IR}_j$ serves to guide the user on the minimum adjustment to the corresponding query weight q_j that is necessary to induce a change in the query result. After refining $\mathbf{q}$, the user could submit it as a fresh query to the server to produce a new top-k result $\mathcal{R}'(\mathbf{q})$ and a new set of immutable regions $\mathcal{IR}'_j$.

The above formalization focuses on $\phi = 0$, i.e., when the user is interested in weight ranges that entirely preserve the result. However, in many cases the user may want to iteratively refine and resubmit the query until satisfied, or she may wish to alter the result more aggressively than inducing a single perturbation per adjustment. Therefore, for responsiveness and scalability reasons, it is desirable to compute multiple immutable regions (and the top-k result for each of them) in an one-off process. Specifically, we may produce regions and results for up to $\phi \geq 0$ successive query refinements, where ϕ is a user-specified or application-dependent parameter. Table 1 summarizes frequently used notation.

4. SCAN – A PRELIMINARY SOLUTION

In this section, we describe a preliminary technique to derive immutable regions, called *Scan*. Assuming that the top-k result has already been computed by TA, *Scan* executes in three phases. First, it derives an interim immutable region for each query dimension that preserves the relative order *within the result* $\mathcal{R}(\mathbf{q})$. In the second phase, it refines the immutable region to ensure that the result elements are not displaced by any tuple *in the candidate list* $\mathcal{C}(\mathbf{q})$. In the third and final phase, it considers tuples that are *neither in the result nor in the candidate list*. Processing builds on Lemma 1.

LEMMA 1. *Consider tuples* $\mathbf{d}_\alpha, \mathbf{d}_\beta \in \mathcal{D}$ *such that* $\mathcal{S}(\mathbf{d}_\beta, \mathbf{q}) \leq \mathcal{S}(\mathbf{d}_\alpha, \mathbf{q})$ *for query* $\mathbf{q}$. *A change in one of the weights* q_j *in* $\mathbf{q}$ *by some* $\delta q_j \in [-q_j, 1 - q_j]$ *preserves the inequality* $\mathcal{S}(\mathbf{d}_\beta, \mathbf{q}) \leq \mathcal{S}(\mathbf{d}_\alpha, \mathbf{q})$ *if and only if*

$$\delta q_j (d_{\beta j} - d_{\alpha j}) \leq \mathcal{S}(\mathbf{d}_\alpha, \mathbf{q}) - \mathcal{S}(\mathbf{d}_\beta, \mathbf{q}) \tag{1}$$

Specifically,

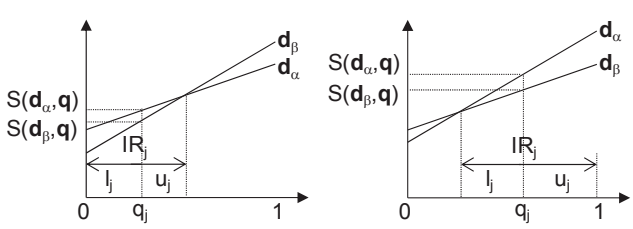
(a) If $d_{\beta j} > d_{\alpha j}$, *then*

$$\delta q_j \in \left[-q_j, \frac{\mathcal{S}(\mathbf{d}_\alpha, \mathbf{q}) - \mathcal{S}(\mathbf{d}_\beta, \mathbf{q})}{d_{\beta j} - d_{\alpha j}}\right) \tag{2}$$

(b) If $d_{\beta j} < d_{\alpha j}$, *then*

$$\delta q_j \in \left(\frac{\mathcal{S}(\mathbf{d}_\alpha, \mathbf{q}) - \mathcal{S}(\mathbf{d}_\beta, \mathbf{q})}{d_{\beta j} - d_{\alpha j}}, 1 - q_j\right] \tag{3}$$

Case (a) is illustrated in Figure 4(a), where the x-axis corresponds to q_j values and the y-axis to the scores of the tuples. At the current q_j value (see vertical dotted line at q_j), $\mathbf{d}_\alpha$ scores higher than $\mathbf{d}_\beta$. Since $d_{\beta j} > d_{\alpha j}$, the score of $\mathbf{d}_\beta$ rises at a steeper incline, which enables it to overtake $\mathbf{d}_\alpha$ at $q_j + u_j$, where $u_j = \frac{\mathcal{S}(\mathbf{d}_\alpha,\mathbf{q}) - \mathcal{S}(\mathbf{d}_\beta,\mathbf{q})}{d_{\beta j} - d_{\alpha j}}$. For q_j inside interval $[0, q_j + u_j)$, the two tuples maintain their relative order. Referring to the running example in Figure 1 and considering the first query dimension, tuple $\mathbf{d}_2$ retains its lead over $\mathbf{d}_1$ as long as δq_1 is no larger than $u_1 = 0.1$. Case (b) is illustrated in Figure 4(b), where $l_j = \frac{\mathcal{S}(\mathbf{d}_\alpha,\mathbf{q}) - \mathcal{S}(\mathbf{d}_\beta,\mathbf{q})}{d_{\beta j} - d_{\alpha j}}$. In the context of Figure 1, this case applies to $\mathbf{d}_1$ keeping ahead of $\mathbf{d}_3$ for values of q_1 larger than $q_1 + l_1 = q_1 - \frac{16}{35}$ (i.e., when δq_1 is no smaller than $l_1 = -\frac{16}{35}$).



(a) $d_{\beta j} > d_{\alpha j}$ (b) $d_{\beta j} < d_{\alpha j}$

Figure 4: Immutable Region $\mathcal{IR}_j$ for q_j

Given a top-k query, suppose that TA produces result $\mathcal{R}(\mathbf{q})$ and candidate list $\mathcal{C}(\mathbf{q})$. Let t_j be the sorting key (i.e., $d_{\alpha j}$ value) of the tuple immediately after the last tuple processed in $\mathcal{L}_j$. The termination condition of TA ensures that $\sum_{j=1}^{m} q_j \times t_j \leq \mathcal{S}(\mathbf{d}_k, \mathbf{q})$ – recall that $\mathcal{S}(\mathbf{d}_k, \mathbf{q})$ is the score of the last result tuple.

Phase 1: In Phase 1 of *Scan*, we derive an interim immutable region $\mathcal{IR}_j = (l_j, u_j)$ based solely on $\mathcal{R}(\mathbf{q})$, i.e., we compute the widest q_j range where the relative order among result tuples is preserved. Let ϵ_j^α $(1 \leq \alpha < k)$ denote the range of δq_j over which $\mathbf{d}_\alpha$ remains ahead of $\mathbf{d}_{\alpha+1}$ in $\mathcal{R}(\mathbf{q})$, i.e., $\mathcal{S}(\mathbf{d}_\alpha, \mathbf{q}) \geq \mathcal{S}(\mathbf{d}_{\alpha+1}, \mathbf{q})$. If $d_{\alpha+1,j} > d_{\alpha j}$, we have $\epsilon_j^\alpha = \left[-q_j, \frac{\mathcal{S}(\mathbf{d}_\alpha,\mathbf{q}) - \mathcal{S}(\mathbf{d}_{\alpha+1},\mathbf{q})}{d_{\alpha+1,j} - d_{\alpha j}}\right)$ by Formula 2. If $d_{\alpha+1,j} < d_{\alpha j}$, then $\epsilon_j^\alpha = \left(\frac{\mathcal{S}(\mathbf{d}_\alpha,\mathbf{q}) - \mathcal{S}(\mathbf{d}_{\alpha+1},\mathbf{q})}{d_{\alpha+1,j} - d_{\alpha j}}, 1 - q_j\right]$ by Formula 3. In the end, we compute $\mathcal{IR}_j = \bigcap_{\alpha=1}^{k-1} \epsilon_j^\alpha$. Phase 1 is summarized by Algorithm 1.

Algorithm 1 Check for Reorderings within $\mathcal{R}(\mathbf{q})$

1: Let $\mathcal{IR}_j = (l_j, u_j)$ where $l_j = -q_j$ and $u_j = 1 - q_j$
2: **for** $\alpha = 1$ to $k - 1$ **do**
3: **if** $(d_{\alpha+1,j} > d_{\alpha j})$ **then**
4: $u_j = \min\left(u_j, \frac{\mathcal{S}(\mathbf{d}_\alpha,\mathbf{q}) - \mathcal{S}(\mathbf{d}_{\alpha+1},\mathbf{q})}{d_{\alpha+1,j} - d_{\alpha j}}\right)$
5: **else if** $(d_{\alpha-1,j} < d_{\alpha j})$ **then**
6: $l_j = \max\left(l_j, \frac{\mathcal{S}(\mathbf{d}_\alpha,\mathbf{q}) - \mathcal{S}(\mathbf{d}_{\alpha+1},\mathbf{q})}{d_{\alpha+1,j} - d_{\alpha j}}\right)$

Phase 2: Starting from the interim region $\mathcal{IR}_j$ produced by the above algorithm, Phase 2 further constrains it so that $\forall \delta q_j \in \mathcal{IR}_j$, none of the candidates $\mathbf{d}_\beta$ in $\mathcal{C}(\mathbf{q})$ are able to overtake the last result tuple, $\mathbf{d}_k$. Let ϵ_j^β $(k < \beta \leq k + |\mathcal{C}(\mathbf{q})|)$ denote the range of δq_j values within which $\mathbf{d}_k$ remains ahead of $\mathbf{d}_\beta$ in score, i.e., the relationship $\mathcal{S}(\mathbf{d}_\beta, \mathbf{q}) \leq \mathcal{S}(\mathbf{d}_k, \mathbf{q})$ is upheld. If $d_{\beta j} > d_{kj}$, $\epsilon_j^\beta = \left[-q_j, \frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q}) - \mathcal{S}(\mathbf{d}_\beta,\mathbf{q})}{d_{\beta j} - d_{kj}}\right)$ by Formula 2. If $d_{\beta j} < d_{kj}$, then $\epsilon_j^\beta = \left(\frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q}) - \mathcal{S}(\mathbf{d}_\beta,\mathbf{q})}{d_{\beta j} - d_{kj}}, 1 - q_j\right]$ by Formula 3. Eventually, we set $\mathcal{IR}_j = \mathcal{IR}_j \cap \bigcap_{\beta=k+1}^{k+|\mathcal{C}(\mathbf{q})|} \epsilon_j^\beta$ (where the original version of $\mathcal{IR}_j$ is the immutable region derived in Phase 1). This procedure mirrors Algorithm 1, skipping line 1, with β iterating from $k + 1$ to $k + |\mathcal{C}(\mathbf{q})|$ in line 2, and $\mathbf{d}_k, \mathbf{d}_\beta$ replacing $\mathbf{d}_\alpha$ and $\mathbf{d}_{\alpha+1}$, respectively, in lines 3 to 6.

Phase 3: In Phase 3 we refine $\mathcal{IR}_j$ by examining whether any tuple outside $\mathcal{R}(\mathbf{q})$ and $\mathcal{C}(\mathbf{q})$ may still enter the result when δq_j lies within the current $\mathcal{IR}_j$ and, if so, shrink the immutable region accordingly. We begin by noting that $\mathcal{IR}_j$ in the general case spans both negative and positive values; i.e., by definition $l_j \leq 0$ and $u_j \geq 0$. We distinguish two cases when considering a tuple $\mathbf{d}_\beta$ in $\mathcal{D} \backslash \mathcal{R}(\mathbf{q}) \backslash \mathcal{C}(\mathbf{q})$:

- If $d_{\beta j} > d_{kj}$, then tuple $\mathbf{d}_\beta$ can overtake $\mathbf{d}_k$ only if q_j increases, i.e., for some $\delta q_j > 0$. Hence, $\mathbf{d}_\beta$ may only affect the upper bound u_j of the immutable region (but not the lower).

- Conversely, if $d_{\beta j} < d_{kj}$, tuple $\mathbf{d}_\beta$ can overtake $\mathbf{d}_k$ only if $\delta q_j < 0$. Thus, $\mathbf{d}_\beta$ may only influence the lower bound l_j of the immutable region, but not u_j.

Having made the above distinction, we now need a mechanism to identify those tuples $\mathbf{d}_\beta \in \mathcal{D} \backslash \mathcal{R}(\mathbf{q}) \backslash \mathcal{C}(\mathbf{q})$ that may alter either the upper or the lower bound of the immutable region (because, clearly, scanning every tuple outside $\mathcal{R}(\mathbf{q})$ and $\mathcal{C}(\mathbf{q})$ is impractical).

Consider the first case and focus on u_j. If TA had encountered the entry of $\mathbf{d}_k$ in the j-th inverted list via sorted access, any tuple $\mathbf{d}_\beta$ with $d_{\beta j} > d_{kj}$ (i.e., any tuple that precedes $\mathbf{d}_k$ in $\mathcal{L}_j$) would have already been processed during TA execution and thus included in either $\mathcal{R}(\mathbf{q})$ or $\mathcal{C}(\mathbf{q})$. In this case, u_j is finalized without any further consideration in Phase 3. Alternatively (i.e., if TA retrieved coordinate d_{kj} via random access in the external file), we proceed as follows. Let $\overline{\mathcal{S}}(\mathbf{d}_k, \mathbf{q}) = \mathcal{S}(\mathbf{d}_k, \mathbf{q}) + u_j \times d_{kj}$. We resume TA

(i.e., continue scanning the inverted lists from where top-k computation stopped), until $\sum_{i \neq j} q_i \times t_i + (q_j + u_j) \times t_j \leq \overline{\mathcal{S}}(\mathbf{d}_k, \mathbf{q})$. For each $\mathbf{d}_\beta$ encountered in the process, we update $\mathcal{IR}_j$ by Formula 2, setting $u_j = \min\left(u_j, \frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q}) - \mathcal{S}(\mathbf{d}_\beta,\mathbf{q})}{d_{\beta j} - d_{kj}}\right)$. When the above termination condition is met, any non-encountered tuple is guaranteed not to affect u_j.

Consider the second case and lower bound l_j. Let $\underline{\mathcal{S}}(\mathbf{d}_k, \mathbf{q}) = \mathcal{S}(\mathbf{d}_k, \mathbf{q}) + l_j \times d_{kj}$. To retrieve tuples that could affect l_j, we resume TA and stop when $\sum_{i \neq j} q_i \times t_i + (q_j + l_j) \times t_j \leq \underline{\mathcal{S}}(\mathbf{d}_k, \mathbf{q})$. For each encountered tuple $\mathbf{d}_\beta$, we update $\mathcal{IR}_j$ by Formula 3, setting $l_j = \max\left(l_j, \frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q}) - \mathcal{S}(\mathbf{d}_\beta,\mathbf{q})}{d_{\beta j} - d_{kj}}\right)$. Once the termination condition is met, non-encountered tuples are guaranteed not to enter the result for any $\delta q_j \geq l_j$.

Algorithm 2 summarizes Phase 3. Observe that finalizing l_j and u_j can be performed simultaneously, with a combined termination condition given in line 4.[2] This condition also guarantees the overall correctness of *Scan*, ensuring that all tuples that can potentially affect the result in the j-th immutable region have been processed. *Scan* is executed for each query dimension in turn to produce all immutable regions $\mathcal{IR}_j$, $j \in [1, qlen]$. Note that whenever a tuple is encountered in Phase 3 for some query dimension, it is inserted into $\mathcal{C}(\mathbf{q})$ in line 6, so that it is processed in Phase 2 for the next query dimension.

Algorithm 2 Check for New Candidates in $\mathcal{D} \backslash \mathcal{R}(\mathbf{q}) \backslash \mathcal{C}(\mathbf{q})$

1: Suppose that $\mathcal{IR}_j = (l_j, u_j)$
2: $\overline{S} = \mathcal{S}(\mathbf{d}_k, \mathbf{q}) + u_j \times d_{kj}$
3: $\underline{S} = \mathcal{S}(\mathbf{d}_k, \mathbf{q}) + l_j \times d_{kj}$
4: **while** ($\sum_{i \neq j} q_i \times t_i + (q_j + l_j) \times t_j > \underline{S}$) or ($\sum_{i \neq j} q_i \times t_i + (q_j + u_j) \times t_j > \overline{S}$) **do**
5: Resume TA to produce the next candidate $\mathbf{d}_\beta$
6: Insert $\mathbf{d}_\beta$ into $\mathcal{C}(\mathbf{q})$
7: **if** ($d_{\beta j} > d_{kj}$) **then**
8: $u_j = \min\left(u_j, \frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q}) - \mathcal{S}(\mathbf{d}_\beta,\mathbf{q})}{d_{\beta j} - d_{kj}}\right)$
9: $\overline{S} = \mathcal{S}(\mathbf{d}_k, \mathbf{q}) + u_j \times d_{kj}$
10: **else if** ($d_{\beta j} < d_{kj}$) **then**
11: $l_j = \max\left(l_j, \frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q}) - \mathcal{S}(\mathbf{d}_\beta,\mathbf{q})}{d_{\beta j} - d_{kj}}\right)$
12: $\underline{S} = \mathcal{S}(\mathbf{d}_k, \mathbf{q}) + l_j \times d_{kj}$

Figure 5 demonstrates the various phases of *Scan* in our running example, after the application of TA (i.e., continuing from Figure 2). The figure illustrates the comparisons among tuples and the evolution of immutable regions for both query dimensions.

An extension of *Scan* to the general case of $\phi \geq 0$ can be achieved easily by computing the immutable regions for several successive perturbations in the result. Specifically, after deriving the original immutable region $\mathcal{IR}_j$, we may conceptually move q_j to u_j in order to force the associated result perturbation, and re-apply *Scan* in a one-way fashion. This means that we need to compute only the upper bound of the next immutable region $\mathcal{IR}'_j$, since its lower bound coincides with the current u_j. The situation is symmetric for the immutable regions to the left of $\mathcal{IR}_j$. Overall, we need to perform this one-way procedure ϕ times to the left, and ϕ times to the right of the original immutable region. Although conceptually simple, dealing with $\phi > 0$ in *Scan* is costly due to this iterative re-processing of the immutable region request. That is one of the drawbacks of *Scan* which are overcome by our advanced algorithm, CPT, described in the next sections.

[2]To simplify presentation, the pseudo-code assumes that d_{kj} was retrieved by TA via random access in the external file, and thus processing is required for u_j (as opposed to the sorted access case).

Phase 1: Check the result tuples in $\mathcal{R}(\mathbf{q})$
To maintain $\mathcal{S}(\mathbf{d}_1, \mathbf{q}) \leq \mathcal{S}(\mathbf{d}_2, \mathbf{q})$,
$\mathcal{IR}_1 = [-0.8, 0.1)$
$\mathcal{IR}_2 = (-\frac{1}{18}, 0.5]$

Phase 2: Check the candidate tuples in $\mathcal{C}(\mathbf{q})$
To maintain $\mathcal{S}(\mathbf{d}_3, \mathbf{q}) \leq \mathcal{S}(\mathbf{d}_1, \mathbf{q})$,
$\mathcal{IR}_1 = \mathcal{IR}_1 \cap (-\frac{16}{35}, 0.2] = (-\frac{16}{35}, 0.1)$
$\mathcal{IR}_2 = \mathcal{IR}_2 \cap [-0.5, \frac{2}{3}) = (-\frac{1}{18}, 0.5]$

Phase 3: Check the tuples in $\mathcal{D} \backslash \mathcal{R}(\mathbf{q}) \backslash \mathcal{C}(\mathbf{q})$
For $\mathcal{IR}_1$:
$\overline{S} = \mathcal{S}(\mathbf{d}_1, \mathbf{q}) + 0.1 \times 0.8 = 0.88$
$\underline{S} = \mathcal{S}(\mathbf{d}_1, \mathbf{q}) - \frac{16}{35} \times 0.8 = 0.43$
Test: $((0.8 - \frac{16}{35}) \times 0.1 + 0.5 \times 0.6 = 0.33 < \underline{S})$ and
$((0.8 + 0.1) \times 0.1 + 0.5 \times 0.6 = 0.39 < \overline{S})$
$\Rightarrow$ No need to resume TA to examine new tuples
For $\mathcal{IR}_2$:
$\overline{S} = \mathcal{S}(\mathbf{d}_1, \mathbf{q}) + 0.5 \times 0.32 = 0.96$
$\underline{S} = \mathcal{S}(\mathbf{d}_1, \mathbf{q}) - \frac{1}{18} \times 0.32 = 0.78$
Test: $((0.8 \times 0.1 + (0.5 - \frac{1}{18}) \times 0.6 = 0.35 < \underline{S})$ and
$((0.8 \times 0.1 + (0.5 + 0.5) \times 0.6 = 0.68 < \overline{S})$
$\Rightarrow$ No need to resume TA to examine new tuples

Figure 5: Execution of *Scan* on the Running Example

An important remark regards reporting the top-k result along with each immutable region. This is achieved easily while the immutable regions are being formed. Take the first immutable region (i.e., $\phi = 0$) in the j-th dimension for example. For each bound of $\mathcal{IR}_j$ (i.e., for each of l_j and u_j) we record the latest processed tuple that updated its value. Let $\mathbf{d}_\beta$ be the tuple recorded for u_j. If $\mathbf{d}_\beta$ is already in $\mathcal{R}(\mathbf{q})$, then the result in the region immediately to the right of $\mathcal{IR}_j$ contains the same tuples, reordered so that $\mathbf{d}_\beta$ overtakes the one preceding it. On the other hand, if $\mathbf{d}_\beta$ does not belong to the current $\mathcal{R}(\mathbf{q})$, then the new top-k result is formed by replacing the last (i.e., the k-th) tuple in $\mathcal{R}(\mathbf{q})$ with $\mathbf{d}_\beta$. Computing the top-k result in different regions is similar in the techniques described next, thus the discussion focuses only on deriving the immutable regions themselves.

Unlike our formulation as stated so far, in certain applications reorderings among result tuples may not be of interest to users and applications, i.e., such reorderings should not be counted as result perturbations. Instead, only changes in the *composition* of $\mathcal{R}(\mathbf{q})$ would be considered as valid perturbations, i.e., inclusions of new tuples into the result. *Scan*, as well as our advanced techniques in subsequent sections, extend trivially to this scenario by simply skipping Phase 1 and initializing the immutable region to its widest possible form (i.e., $l_j = -q_j$ and $u_j = 1 - q_j$) before Phase 2 starts. Although we evaluate performance in this scenario too (in Section 7.4), unless otherwise specified, in the following we assume the original problem formulation.

5. CPT – PRUNING AND THRESHOLDING

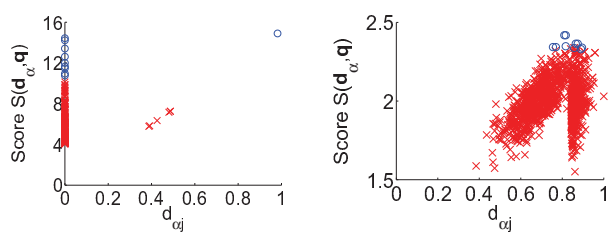
In *Scan* the first and third phases are relatively inexpensive. Specifically, running Algorithm 1 on $\mathcal{R}(\mathbf{q})$ (Phase 1) is fast because the number of result tuples k is typically small, e.g., $k = 10$. Phase 3 is also inexpensive, because after Phases 1 and 2, the immutable regions $\mathcal{IR}_j$ are already very tight, so only a few additional tuples from the sorted lists would satisfy the condition in line 4 of Algorithm 2. In contrast, Phase 2 is the bottleneck, because $\mathcal{C}(\mathbf{q})$

is one or two orders of magnitude larger than $\mathcal{R}(\mathbf{q})$. We verify this intuition empirically in Section 7. Based on that observation, we focus on Phase 2, and introduce an algorithm to quickly prune the inconsequential candidate tuples in $\mathcal{C}(\mathbf{q})$, before processing the remaining ones with a thresholding technique. The resulting algorithm is named CPT, for *Candidate Pruning and Thresholding*. In the following we address the $\phi = 0$ case, before generalizing to $\phi > 0$ in Section 6.

5.1 Candidate Pruning

Here we describe the first component of CPT, i.e., candidate pruning. To illustrate the rationale behind it, we run a query $\mathbf{q}$ with equal weights on four randomly chosen search terms (i.e., query dimensions) on the WSJ corpus, described in detail in Section 7.1. Figure 6(a) plots the score of the top-10 result tuples in $\mathcal{R}(\mathbf{q})$ as blue circles and the candidate tuples of $\mathcal{C}(\mathbf{q})$ as red crosses, versus their coordinates in the first query dimension. The charts for the other three dimensions show an identical trend and are omitted for brevity. The key observation is that all the result and candidate tuples in this example appear at the front of one of the four inverted lists, and have zero coordinates in the other three dimensions.

Consider the first query dimension in Figure 6(a). As q_1 increases, all tuples on the y-axis retain their original scores because they have zero coordinate in the first query dimension. In contrast, every tuple on the slope experiences a rise in score that is proportional to its first coordinate. Eventually, the lowest blue circle (i.e., result tuple) that lies on the y-axis will be replaced in the result by the red cross (candidate) that is highest on the slope, followed by the next-highest red cross on the slope, and so on. Conversely, as q_1 decreases, the blue circles on the slope experience a reduction in score; eventually, they will be replaced by the top red cross that lies on the y-axis, then the second red cross on the y-axis, and so on. Consequently, to compute the immutable region (for $\phi = 0$), among the red crosses on the slope, it suffices to consider only the one with the largest coordinate (in the first query dimension). Likewise, among the red crosses on the y-axis, we need to take into account only the one with the highest score.



(a) WSJ Data (b) Correlated Data

Figure 6: Result and Candidate Tuples

Clearly, Figure 6(a) demonstrates a special (yet common) situation where the result and candidate tuples have non-zero values in only one of the query dimensions. However, the key observation that underlies our pruning technique holds also in the general case, where there is a third type of result and candidate tuples that have non-zero coordinate values in two or more query dimensions (apart from those on the slope and on the y-axis). Specifically, for each query dimension $j \in [1, qlen]$, the candidate tuples in $\mathcal{C}(\mathbf{q})$ can be partitioned into:

- $\mathcal{C}_j^0 = \{\mathbf{d}_\beta | \mathbf{d}_\beta \in \mathcal{C}(\mathbf{q}) \text{ and } d_{\beta j} = 0\}$; the tuples in $\mathcal{C}_j^0$ are inside $\mathcal{C}(\mathbf{q})$ (i.e., were encountered by TA) due to their coordinates in some query dimension(s) other than j.

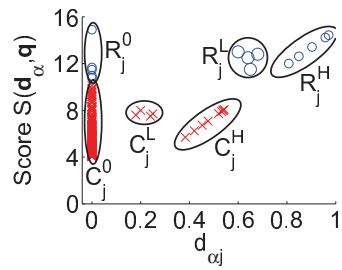


Figure 7: Partitions of $\mathcal{R}(\mathbf{q})$ and $\mathcal{C}(\mathbf{q})$

- $\mathcal{C}_j^H = \{\mathbf{d}_\beta | \mathbf{d}_\beta \in \mathcal{C}(\mathbf{q}) \text{ and } d_{\beta i} = 0 \; \forall i \neq j\}$; the tuples in $\mathcal{C}_j^H$ are in $\mathcal{C}(\mathbf{q})$ solely due to their j-th coordinate.
- $\mathcal{C}_j^L = \{\mathbf{d}_\beta | \mathbf{d}_\beta \in \mathcal{C}(\mathbf{q}), d_{\beta j} > 0 \text{ and } \exists i \neq j \text{ such that } d_{\beta i} > 0\}$; the tuples in $\mathcal{C}_j^L$ are in $\mathcal{C}(\mathbf{q})$ due to the combined contributions from dimension j and at least one other query dimension.

The three sets are illustrated in Figure 7. $\mathcal{C}_j^0$ corresponds to candidates on the y-axis, $\mathcal{C}_j^H$ on the slope, and $\mathcal{C}_j^L$ to candidates neither on the y-axis nor on the slope. The figure also illustrates a similar partitioning of the result tuples.

As we see in Figure 6(a), $\mathcal{C}_j^0$ and $\mathcal{C}_j^H$ could be sizable. By pruning $\mathcal{C}_j^0$ and $\mathcal{C}_j^H$, we may effectively reduce the processing time to derive $\mathcal{IR}_j$. Unlike *Scan*, CPT eliminates many candidates without consideration based on the following lemmata.

LEMMA 2. *Consider a query $\mathbf{q}$, its top-k result $\mathcal{R}(\mathbf{q})$, and candidate tuples $\mathcal{C}(\mathbf{q})$. For any $j \in [1, qlen]$, the lower bound l_j of immutable region $\mathcal{IR}_j$ cannot be affected by any tuple in $\mathcal{C}_j^H$. Also, l_j can be affected by only one tuple in $\mathcal{C}_j^0$; namely, the one with the highest current score.*

PROOF. First, we show that no candidates in $\mathcal{C}_j^H$ may enter the result as q_j decreases. Let $\mathbf{d}_\beta$ be a tuple in $\mathcal{C}_j^H$, and $\mathbf{d}_\alpha$ be any member of $\mathcal{R}(\mathbf{q})$. If $d_{\beta j} > d_{\alpha j}$, $\mathbf{d}_\beta$ undergoes a larger reduction in score than $\mathbf{d}_\alpha$, and cannot overtake it. If $d_{\beta j} \leq d_{\alpha j}$, $\mathbf{d}_\beta$ cannot score higher than $\mathbf{d}_\alpha$ for any decrease in q_j, because by definition $\mathbf{d}_\beta$ has zero values in all other query dimensions, and therefore all its coordinates of interest are smaller than those of $\mathbf{d}_\alpha$.

Regarding the second part of the lemma, lowering q_j reduces the score of result tuples $\mathbf{d}_\alpha$ with non-zero $d_{\alpha j}$ values. As reductions in q_j may cause result tuples to drop out of $\mathcal{R}(\mathbf{q})$, the candidates $\mathbf{d}_\beta$ in $\mathcal{C}_j^0$ will qualify for inclusion in order of their current score (which is independent of q_j since their $d_{\beta j}$ values are zero). Hence, only the top-scoring tuple in $\mathcal{C}_j^0$ may affect the lower bound l_j. □

LEMMA 3. *Consider a query $\mathbf{q}$, its top-k result $\mathcal{R}(\mathbf{q})$, and candidate tuples $\mathcal{C}(\mathbf{q})$. For any $j \in [1, qlen]$, the upper bound u_j of immutable region $\mathcal{IR}_j$ cannot be affected by any tuple in $\mathcal{C}_j^0$. Also, u_j can be affected by only one tuple in $\mathcal{C}_j^H$; namely, the one with the highest value in the j-th dimension.*

PROOF. If q_j increases, the result tuples' scores will either increase or remain constant. The scores of all tuples in $\mathcal{C}_j^0$, on the other hand, do not change (as they are independent of q_j) and therefore they cannot enter the top-k result.

Regarding $\mathcal{C}_j^H$, as q_j increases, the candidates $\mathbf{d}_\beta \in \mathcal{C}_j^H$ may enter the result. Since their score increases proportionally to their $d_{\beta j}$ value, the tuple with the highest $d_{\beta j}$ in $\mathcal{C}_j^H$ will enter the result first. Therefore, it is the only tuple in $\mathcal{C}_j^H$ that could affect u_j. □

Lemmata 2 and 3 suggest that $\mathcal{IR}_j$ computation needs a single tuple from each $\mathcal{C}_j^0$ and $\mathcal{C}_j^H$, which significantly reduces processing

time. Note that pruning could be performed on the fly during TA execution, by maintaining in $\mathcal{C}(\mathbf{q})$ only the top-scoring $\mathcal{C}_j^0$ tuple encountered so far, and only the $\mathcal{C}_j^H$ tuple with the highest j-th coordinate. This lowers the memory overhead for keeping $\mathcal{C}(\mathbf{q})$.

5.2 Candidate Thresholding

Pruning reduces the size of $\mathcal{C}_j^0$ and $\mathcal{C}_j^H$. However, $\mathcal{C}_j^L$ (i.e., the remaining constituent of $\mathcal{C}(\mathbf{q})$) may also be sizable, especially when the query dimensions are correlated. To illustrate, we run a query $\mathbf{q}$ with equal weights on four randomly chosen dimensions using a dataset with positively correlated coordinates; the dataset is described in detail in Section 7.1. Figure 6(b) plots the scores of result and candidate tuples against the first query dimension – charts for the remaining three dimensions are similar. The top-10 tuples are plotted as blue circles and the candidates as red crosses. In contrast to Figure 6(a), here $\mathcal{C}_j^0 = \emptyset$ and $|\mathcal{C}_j^L| \gg |\mathcal{C}_j^H|$. Note that the members $\mathbf{d}_\alpha$ of $\mathcal{C}_j^H$ lie on the line $\mathcal{S}(\mathbf{d}_\alpha, \mathbf{q}) = q_j \times d_{\alpha j}$ since by definition they have zero values in the other query dimensions.

In this section, we propose a technique that vastly reduces the number of $\mathcal{C}_j^L$ candidates considered. Our approach leverages on a geometric reduction of the problem used in tandem with a thresholding strategy.

We introduce the crux of our technique using Figure 8. The figure plots the score-coordinate space for tuples $\mathbf{d}_\alpha$ in $\mathcal{C}(\mathbf{q})$, i.e., the x-axis corresponds to $d_{\alpha j}$ values and the y-axis to $\mathcal{S}(\mathbf{d}_\alpha, \mathbf{q})$. The horizontal line indicates the score of the k-th result tuple, $y = \mathcal{S}(\mathbf{d}_k, \mathbf{q})$. The tuples in $\mathcal{R}(\mathbf{q})$ lie on or above this line, while candidates lie below it (without loss of generality, assume that there are no ties with $\mathbf{d}_k$). Let (l_j, u_j) demarcate the (interim) immutable region $\mathcal{IR}_j$ derived in Phase 1, i.e., after processing the tuples in $\mathcal{R}(\mathbf{q})$. The immutable region is represented by the two solid lines sloping down from $\mathbf{d}_k$. The left line has a gradient of l_j, whereas the gradient of the right line is u_j. Consider a candidate $\mathbf{d}_\beta$ with coordinate $d_{\beta j} > d_{kj}$, as illustrated in the figure. By Formula 2 it follows that if $\mathbf{d}_\beta$ lies below the right bound line (corresponding to u_j) it cannot overtake $\mathbf{d}_k$ within the current immutable region. The situation is similar for candidates with j-th coordinate smaller than d_{kj}, with the line at gradient l_j playing the role of the bound. In other words, *any candidate below the lines that correspond to l_j and u_j cannot affect the immutable region*. Processing such tuples is unnecessary and we aim to avoid it.

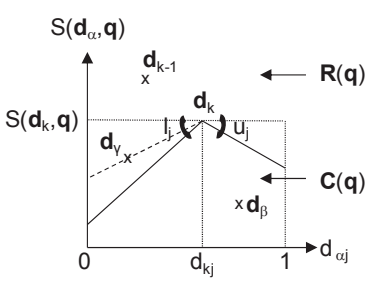


Figure 8: Intuition behind Candidate Thresholding

By the same geometric intuition, *any tuple above the bound lines necessitates shrinking the immutable region*, i.e., raising the corresponding line (depending on whether it is on the left or the right of $\mathbf{d}_k$ in the score-coordinate plane) to pass through it. One such tuple is $\mathbf{d}_\gamma$, which requires updating (raising) l_j to the dashed line. The task in Phase 2 is to raise the bound lines toward the horizontal line $y = \mathcal{S}(\mathbf{d}_k, \mathbf{q})$ just enough to keep all candidates below, thus effectively tightening l_j and u_j in the process. Raising the lines early on in Phase 2 implies disqualifying more candidates from consideration, and hence faster termination. This rationale leads to a second design direction, i.e., to process the candidates in order of their potential to affect $\mathcal{IR}_j$.

Consider the region on the right of $\mathbf{d}_k$ in the score-coordinate plane. This region corresponds to candidates with j-th coordinate larger than d_{kj} and concerns u_j only (not l_j). Candidates with a high potential to affect u_j (i.e., with more chances to raise the right bound line) are those with a large score *and* a large j-th coordinate. This fact motivates our thresholding technique, which takes into account the aforementioned two-fold criterion to decide the processing order among candidates.

Specifically, based on Lemma 3 and the pruning described in Section 5.1, our set of candidates in the right region includes $\mathcal{C}_j^L$ appended by the tuple in $\mathcal{C}_j^H$ with the highest j-th coordinate. Observe that some of the tuples in $\mathcal{C}_j^L$ do not fall in the right region, but this is an issue we ignore for now. $\mathcal{C}_j^L$ is already sorted on score – with the addition of the extra tuple from $\mathcal{C}_j^H$, we form a list of the candidates $\mathcal{SL}_\mathcal{S}$ sorted in decreasing score order. We also form list $\mathcal{SL}_j$ with the candidates sorted on their j-th coordinate (in decreasing order). We probe the two lists in a round-robin fashion. For each popped candidate we apply Formula 2 and lower u_j accordingly. Let $t_\mathcal{S}$ be the score of the next tuple in $\mathcal{SL}_\mathcal{S}$, and t_j be the j-th coordinate of the next tuple in $\mathcal{SL}_j$. These values play the role of thresholds; the slope of every non-considered candidate further down in the lists is lower-bounded by $\frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q})-t_\mathcal{S}}{t_j-d_{kj}}$, i.e., this is the minimum value they can update u_j to. Therefore, we stop probing the lists (i.e., considering new candidates for u_j) when $\frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q})-t_\mathcal{S}}{t_j-d_{kj}}$ rises above the current value of u_j, in which case u_j becomes the final upper bound of Phase 2. Another termination case is when $t_j \leq d_{kj}$, which means that candidates in the right region have been exhausted, and thus we cannot further reduce u_j.

The process is similar for the lower bound l_j of the immutable region. The set of candidates that may affect (raise) l_j in Phase 2 comprises $\mathcal{C}_j^L$ and the top-scoring tuple in $\mathcal{C}_j^0$ (according to Lemma 2). The thresholding process is similar to that for u_j. The difference is that candidates with high potential to update l_j are those with high score but *small* j-th coordinate. Therefore, we use $\mathcal{SL}_\mathcal{S}$ as before, yet follow a reverse access order in $\mathcal{SL}_j$; we access it from bottom to top (i.e., in increasing j-th coordinate order). Let t'_j be the j-th coordinate of the next tuple in $\mathcal{SL}_j$ (i.e., the one with the immediately larger j-th coordinate than the last candidate drawn from the list). Termination occurs when (i) $\frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q})-t_\mathcal{S}}{t'_j-d_{kj}}$ drops below the current l_j value or (ii) $t'_j \geq d_{kj}$.

A way to implement Phase 2 is to perform thresholded processing twice, once for each of l_j and u_j. However, this way we scan (the top part of) $\mathcal{SL}_\mathcal{S}$ twice. Also, when considering candidates for u_j, for instance, some tuples drawn from $\mathcal{SL}_\mathcal{S}$ may fall in the region to the left of $\mathbf{d}_k$ (i.e., the region that affects l_j but not u_j). To save computations we perform both searches concurrently.

This is achieved by a 3-list thresholded probe. $\mathcal{SL}_\mathcal{S}$ is used as is, while $\mathcal{SL}_j$ is treated as two virtual lists:

- $\mathcal{SL}_{j\uparrow}$ for j-coordinates smaller than d_{kj} sorted in ascending order, for which we maintain threshold t'_j.
- $\mathcal{SL}_{j\downarrow}$ for j-coordinates greater than d_{kj} sorted in descending order, keeping track of threshold t_j.

We probe the lists in a round-robin fashion. Candidates drawn from $\mathcal{SL}_{j\uparrow}$ are considered for raising l_j, those drawn from $\mathcal{SL}_{j\downarrow}$ for lowering u_j, while those pulled from $\mathcal{SL}_\mathcal{S}$ are considered for either of the two bounds, depending on whether their j-th coordinate is

Algorithm 3 Candidate Thresholding Method

Let $\mathcal{IR}_j = (l_j, u_j)$ be the interim region from Phase 1.
Set $Search(l_j)$ = ACTIVE; set $Search(u_j)$ = ACTIVE
while ($Search(l_j)$=ACTIVE) or ($Search(u_j)$=ACTIVE) **do**
  Pull top candidate $\mathbf{d}_\beta$ from $\mathcal{SL}_\mathcal{S}$
  if ($d_{\beta j} < d_{kj}$) and ($Search(l_j)$ = ACTIVE) **then**
    $l_j = \max\left(l_j, \frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q})-\mathcal{S}(\mathbf{d}_\beta,\mathbf{q})}{d_{\beta j}-d_{kj}}\right)$
  else if ($d_{\beta j} > d_{kj}$) and ($Search(u_j)$ = ACTIVE) **then**
    $u_j = \min\left(u_j, \frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q})-\mathcal{S}(\mathbf{d}_\beta,\mathbf{q})}{d_{\beta j}-d_{kj}}\right)$
  if $Search(l_j)$ = ACTIVE **then**
    if ($\frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q})-t_\mathcal{S}}{t'_j-d_{kj}} \leq l_j$) or ($t'_j \geq d_{kj}$) **then**
      $Search(l_j)$ = COMPLETE
    else
      Pull top candidate $\mathbf{d}_\beta$ from $\mathcal{SL}_{j\uparrow}$
      $l_j = \max\left(l_j, \frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q})-\mathcal{S}(\mathbf{d}_\beta,\mathbf{q})}{d_{\beta j}-d_{kj}}\right)$
  if $Search(u_j)$ = ACTIVE **then**
    if ($\frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q})-t_\mathcal{S}}{t_j-d_{kj}} \geq u_j$) or ($t_j \leq d_{kj}$) **then**
      $Search(u_j)$ = COMPLETE
    else
      Pull top candidate $\mathbf{d}_\beta$ from $\mathcal{SL}_{j\downarrow}$
      $u_j = \min\left(u_j, \frac{\mathcal{S}(\mathbf{d}_k,\mathbf{q})-\mathcal{S}(\mathbf{d}_\beta,\mathbf{q})}{d_{\beta j}-d_{kj}}\right)$

lower or higher than d_{kj}. The dual termination condition for each of the two searches is checked whenever considering a candidate in its region. If reached for one of the two searches, say for u_j, then $\mathcal{SL}_{j\downarrow}$ is no longer probed, and only the search for l_j continues. Algorithm 3 summarizes the complete thresholding technique. Boolean flags $Search(l_j)$ and $Search(u_j)$ indicate whether the search for l_j and u_j, respectively, is ongoing or completed.

We experimented with alternative probing heuristics, such as pulling candidates from $\mathcal{SL}_\mathcal{S}$ twice as frequently as the other two lists (since $\mathcal{SL}_\mathcal{S}$ is used for two searches, whereas each of the other lists is used for one), as well as heuristics based on estimates of the potential of the 3 tuples at the top of each list. We found round-robin to perform better (or at least comparably well) and prefer it because of its robust performance across all datasets tried.

Since Phase 2 dominates the processing cost of CPT, an important note regards its complexity. Assuming that the query dimensions are independent, [7] proves that the expected cardinality of $\mathcal{C}(\mathbf{q})$ is $O(k^{\frac{1}{qlen}} n^{1-\frac{1}{qlen}})$, which is sublinear to dataset cardinality n. In the worst case, pruning will disqualify no candidate, leaving the entire $\mathcal{C}(\mathbf{q})$ to thresholding. The latter sorts the candidates that remain after pruning. Updating $\mathcal{IR}_j$ for a tuple via Lemma 1 takes constant time. Hence, the total complexity of Phase 2 per query dimension is $O(k^{\frac{1}{qlen}} n^{1-\frac{1}{qlen}} \log(k^{\frac{1}{qlen}} n^{1-\frac{1}{qlen}}))$.

6. CPT EXTENSION TO $\phi > 0$

The description of CPT so far focused on $\phi = 0$, i.e., on immutable regions that allow no result perturbation. Here we extend CPT to $\phi > 0$, starting with Phase 1. For simplicity, we focus on positive deviations $\delta q_j > 0$ and the ϕ immutable regions to the right of the current q_j value. These regions are essentially defined by $\phi+1$ upper bounds u_j^r for $0 \leq r \leq \phi$. Processing is symmetric for negative deviations, unless otherwise specified.

Phase 1: We treat each result tuple as a line in score-coordinate space. Consider Figure 9 where $k = 3$ and $\phi = 3$. Note that

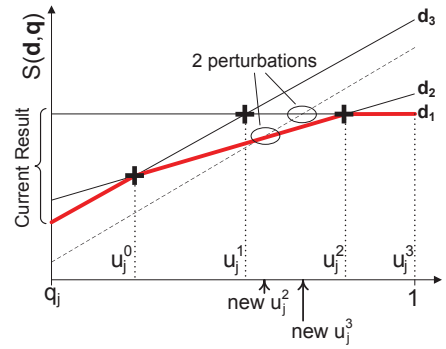


Figure 9: Processing Example when $\phi > 0$ (for $\delta q_j > 0$)

the x-axis starts at the current q_j value and extends to 1, since we focus on positive deviations δq_j. The current top-3 result (at the beginning of the x-axis) includes $\mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3$, shown as the 3 solid lines. The $\phi+1$ leftmost intersections of the result lines correspond to the interim upper bounds u_j^r. The intersections in a set of lines can be found with the plane-sweep algorithm [2]. To save time, not all intersections between result lines need to be computed; as plane-sweep scans the score-coordinate plane from left to right, we stop after encountering the first $\phi+1$ intersections. In our example there are only 3 intersections, indicated by crosses. Since they are fewer than $\phi + 1 = 4$, the interim bound u_j^3 takes its maximum possible value, i.e., $u_j^3 = 1 - q_j$.

Phase 2 (Pruning): In Phase 2 we first disqualify (i.e., prune) candidates based on the following generalization of Lemma 2.

LEMMA 4. *Consider a query* $\mathbf{q}$*, its top-k result* $\mathcal{R}(\mathbf{q})$*, and candidate tuples* $\mathcal{C}(\mathbf{q})$*. For any* $j \in [1, qlen]$ *and* $\phi > 0$*, the immutable regions to the right of* q_j *cannot be affected by any tuple in* $\mathcal{C}_j^0$*. Also, they can be affected by a maximum of* $\phi+1$ *tuples in* $\mathcal{C}_j^H$*; namely, those with the* $\phi+1$ *highest values in the* j*-th dimension.*

PROOF. As q_j increases, the result tuples' scores either increase or remain constant, while the scores of all tuples in $\mathcal{C}_j^0$ remain constant. Hence, no tuple in $\mathcal{C}_j^0$ can enter the result. On the other hand, as q_j increases, the candidates $\mathbf{d}_\beta \in \mathcal{C}_j^H$ may enter the result in order of their j-th coordinate (since their score is equal to $q_j \times d_{\beta j}$, their relative order is preserved). Thus, only the first one could affect u_j^0, only the first two could affect u_j^1, and so on. Overall, only the first $\phi+1$ of them could affect the ϕ immutable regions to the right of q_j. □

Lemma 4 applies to positive deviations in q_j. For negative deviations, pruning is similar: the ϕ immutable regions to the left of q_j are independent of $\mathcal{C}_j^H$, and can be affected by a maximum of $\phi+1$ tuples in $\mathcal{C}_j^0$ (those with the $\phi+1$ highest current scores). The corresponding lemma and its proof are omitted for brevity.

Phase 2 (Thresholding): To explain thresholding, we continue the example in Figure 9. After Phase 1, the *lower envelope* of the three result lines (i.e., the broken line shown in red and bold) indicates the boundary of the result, i.e., it represents the score of the k-th result tuple at different q_j values[3]. If a candidate line intersects the lower envelope, this candidate enters the result at the q_j value of the intersection. Thresholding considers candidates (that pass the pruning criteria) by the same round-robin probing of $\mathcal{SL}_\mathcal{S}$ and

[3]We borrow the term lower envelope from computational geometry. Deriving it over k lines can be done in $O(k \log k)$ time [2].

$\mathcal{SL}_{j\downarrow}$. The difference lies in (i) the update of the immutable regions when candidates are considered and (ii) the termination condition.

Regarding (i), for each candidate tuple pulled from $\mathcal{SL}_{\mathcal{S}}$ or $\mathcal{SL}_{j\downarrow}$ we check whether it intersects the lower envelope (for any $\delta q_j \in [0, u_j^{\phi})$). If so, it enters the result at the point of intersection, and might induce additional perturbations (result reorderings) later. We identify the candidate's intersections with all result lines on or above the lower envelope and update the immutable regions accordingly. In Figure 9, if the dashed line corresponds to a candidate tuple $\mathbf{d}_\beta$, then it causes 2 perturbations, one for each of the intersections marked by ellipses. This lowers u_j^2 and u_j^3 to the x-axis projections of these two intersection points – the new u_j^2 and u_j^3 values are shown by arrows on the x-axis. The lower envelope is updated accordingly in order to reflect the new k-th result tuple at different q_j values, i.e., it now passes through the intersections caused by $\mathbf{d}_\beta$, because $\mathbf{d}_\beta$ is the k-th top tuple in the range between the new u_j^2 and u_j^3. Note that the lower envelope does not extend further than u_j^{ϕ} because perturbations beyond the first ϕ are irrelevant, i.e., the updated envelope extends only until the new u_j^3.

For point (ii), the termination condition also utilizes the lower envelope. Thresholds $t_{\mathcal{S}}$ and t_j (on lists $\mathcal{SL}_{\mathcal{S}}$ and $\mathcal{SL}_{j\downarrow}$, respectively) represent a line that caps the potential of each unseen candidate. This line, termed *threshold line*, is expressed by $y = t_{\mathcal{S}} + t_j \times x$, where y represents the score dimension and x is δq_j. In the context of Figure 9, for example, this line would intersect the y-axis at $t_{\mathcal{S}}$ and have a slope equal to t_j. The threshold line is guaranteed to lie above (the line representing) any non-encountered candidate further down in $\mathcal{SL}_{\mathcal{S}}$ or $\mathcal{SL}_{j\downarrow}$. Therefore, thresholding terminates when the threshold line lies entirely below (i.e., does not intersect) the lower envelope for any $\delta q_j \in [0, u_j^{\phi})$.

Phase 3: Phase 3 considers candidates (and updates the immutable regions accordingly) from deeper in the inverted lists like Algorithm 2, but with a new termination condition. It uses a threshold line expressed by $y = \sum_{i=1}^{qlen} q_i \times t_i + t_j \times x$, where y is the score dimension and x is δq_j. All non-encountered candidate lines lie below the threshold line, and hence Phase 3 (and CPT) terminates when the threshold line is below (i.e., does not intersect) the lower envelope for any $\delta q_j \in [0, u_j^{\phi})$.

7. EXPERIMENTS

Here we evaluate the effectiveness of the pruning and thresholding techniques, both individually and in combination. We first examine the case where $\phi = 0$ (computing a single immutable region) followed by experiments with $\phi > 0$. Finally, we study a scenario where only changes in the result *composition* are considered to be valid perturbations (i.e., reorderings among result tuples are disregarded in immutable region formation).

7.1 Experiment Set-Up

Methods: The CPT algorithm comprises two techniques – pruning and thresholding – which may also be used separately. This gives rise to four alternative methods: (i) *Scan*, described in Section 4, processes all the candidates in $\mathcal{C}(\mathbf{q})$, and provides a baseline to assess the effectiveness of our advanced techniques; (ii) *Prune* enhances *Scan* via pruning (presented in Section 5.1), after which all the remaining candidates in $\mathcal{C}_j^0$, $\mathcal{C}_j^L$ and $\mathcal{C}_j^H$ are examined to tighten $\mathcal{IR}_j$; (iii) *Thres* improves on *Scan* by applying thresholding (introduced in Section 5.2) on all the candidates in $\mathcal{C}(\mathbf{q})$; and (iv) CPT, i.e., the complete Candidate Pruning and Thresholding Algorithm, prunes the candidate list before applying thresholding.

Datasets: We run the methods on two real and one synthetic dataset. The default dataset is WSJ, including 172,891 articles published in the Wall Street Journal from December 1986 to March 1992. Following standard practice in document retrieval, we removed stopwords (i.e., common words like 'the' and 'a' that are not useful for differentiating between documents) and those that appear in only one article. We treated the remaining 181,978 search terms T_j as dataspace dimensions. For each of them, we created an inverted list $\mathcal{L}_j$ of entries $\langle d_\alpha, d_{\alpha j}\rangle$, where d_α is a pointer into an external file that contains the entire $\mathbf{d}_\alpha$ tuple, and $d_{\alpha j}$ is the term frequency (*TF*) of T_j in document $\mathbf{d}_\alpha$, multiplied by the inverse document frequency (*IDF*) of T_j [1]. Queries on WSJ are formed by randomly selecting $qlen$ terms as query dimensions. The weight q_j of each query term is set according to the TF-IDF scheme. This formulation corresponds to classic similarity-based text retrieval. In a real application, immutable regions would provide the user a finer control over the relative importance of the query terms (instead of having to repeat in her query the terms she wishes to stress).

To verify the generality of our approach, we use a second real dataset from a different domain. KB [13] contains 28,452 images, each represented by a 9,693-dimensional feature vector (tuple). An image database can be queried on various features, e.g., average color, texture, image quality, etc. Here, the immutable regions could help the user control/adjust the weights of query features.

The third dataset, ST, is synthetic. Since WSJ and KB have little and moderate correlation among their dimensions, respectively, with ST we explore the case of highly correlated distributions. Such data are a standard benchmark for preference-based querying and occur often in multi-criteria decision making [3]. ST is generated by the 'mvnrnd' function in Matlab, using correlation coefficients of 0.5. This means that ST tuples are clustered along the line from $[0, 0, \ldots, 0]$ to $[1, 1, \ldots, 1]$. ST contains one million tuples in a 20-dimensional space. Queries in KB and ST are formed by randomly selecting query dimensions and their weights.

System Model: Each dataset $\mathcal{D}$ and its inverted lists $\mathcal{L}_j$ are stored on disk. Upon receiving a user query $\mathbf{q}$, the server runs TA to retrieve the top-k result $\mathcal{R}(\mathbf{q})$. Instead of a round-robin strategy, we follow [18] and enhance TA by probing the list $\mathcal{L}_j$ with the largest product $q_j \times d_{\alpha j}$, where $\mathbf{d}_\alpha$ is the last document pulled from $\mathcal{L}_j$. Along with $\mathcal{R}(\mathbf{q})$, the server also retains the list of encountered tuples $\mathcal{C}(\mathbf{q})$ – to conserve memory, it caches only the score of encountered tuples, not their full information. The server runs Redhat Linux, and is equipped with a dual Intel Xeon 3GHz CPU.

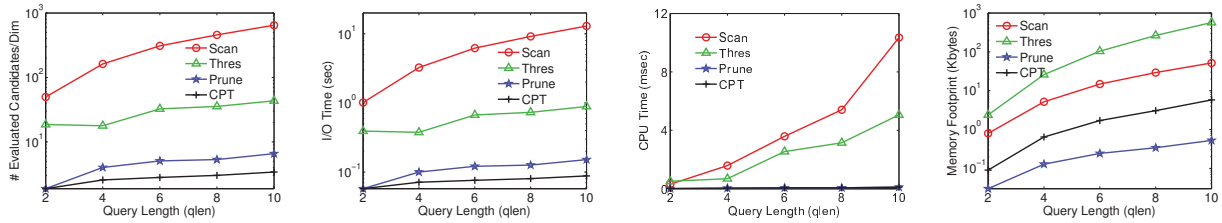
Metrics: Our primary performance metrics include the number of candidates evaluated per query dimension, and the total I/O and CPU costs to compute the immutable regions for all query dimensions. Every reported result is the average over 100 queries.

The CPU measurements by themselves also indicate performance in an alternative setting where the dataset and inverted lists are cached in main memory (instead of disk).

7.2 Experiments for $\phi = 0$

Effect of Query Length: In the first experiment we use WSJ, set $k = 10$, and vary $qlen$ from 2 to 10 (to simulate short web queries as well as longer full-text queries). Phases 1 and 3 are identical in all compared approaches. The total cost of Phase 1 ranges from 60 μsec to 140 μsec, and that of Phase 3 is around 40 msec, which are both at least one order of magnitude smaller than the cost of Phase 2. Since Phase 2 is the main performance determinant, hereafter we focus on the costs of Phase 2 only.

Figure 10 considers Phase 2 in the same experiment. A larger $qlen$ requires TA to search deeper in the query lists. This produces



(a) # Evaluated Candidates (b) I/O Cost (c) CPU Cost (d) Memory Footprint

Figure 10: WSJ Corpus, $k = 10$, varying $qlen$

more candidates in $\mathcal{C}(\mathbf{q})$ and explains the increased I/O and CPU costs for all methods. The effect is more severe for *Scan* which processes *all* tuples in $\mathcal{C}(\mathbf{q})$.

As shown in Figure 6(a), in the WSJ dataset most tuples in $\mathcal{C}(\mathbf{q})$ have a large coordinate in one query dimension, and zero values in all others. Hence, $\mathcal{C}_j^0$ and $\mathcal{C}_j^H$ account for most or even all of the tuples in $\mathcal{C}(\mathbf{q})$. *Prune* is very effective in this case, because it processes a single candidate from each of $\mathcal{C}_j^0$ and $\mathcal{C}_j^H$. Figure 10(a) shows the number of evaluated candidates, i.e., those checked against the k-th result tuple $\mathbf{d}_k$ for potential l_j or u_j update via Lemma 1. At $qlen = 2$, *Prune* processes 1.9 candidates/dimension on average, compared to 49.8 for *Scan*. At $qlen = 10$, the difference widens to 6.6 candidates/dimension for *Prune* and 646.4 for *Scan*. The number of evaluated candidates is the main determinant of CPU cost (Figure 10(c)). Likewise for I/O (Figure 10(b)), since the exact coordinates of evaluated candidates are fetched from disk.

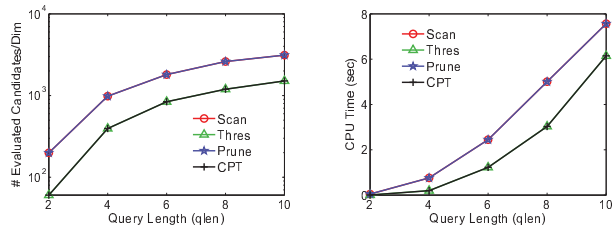
Regarding candidate thresholding, its essence is to focus on high-potential candidates, and (safely) disqualify the rest. This enables *Thres* to reduce the number of evaluated candidates by around 60% at $qlen = 2$, and over 90% at $qlen = 10$, relative to *Scan*. Furthermore, thresholding complements pruning successfully – CPT reduces evaluated candidates by an additional 35% to 50% relative to *Prune*. The reduction in processed candidates explains the I/O and CPU savings (although not visible due to the scale in Figure 10(c), CPT reduces CPU time by over 30% compared to *Prune*).

In Figure 10(d) we plot the memory footprint of the methods. *Scan* maintains a score and a pointer (into the external file) for every tuple in $\mathcal{C}(\mathbf{q})$. *Thres* additionally keeps the $\mathcal{SL}_j$ lists, built on all candidates. *Prune* is enhanced with the space optimization described at the end of Section 5.1, and has the smallest footprint. CPT uses the same optimization and its extra overhead is due to the $\mathcal{SL}_j$ lists (built on the candidates remaining after pruning). The memory footprint of all methods is in the order of Kbytes.

In Figure 11 we repeat the same experiment on ST. Since the I/O cost follows the number of evaluated candidates, we omit the I/O charts. The most important difference from WSJ is that *Prune* is not effective here. The reason is that, unlike WSJ, candidate subsets $\mathcal{C}_j^0$ and $\mathcal{C}_j^H$ include too few tuples to begin with, so there is little room for improvement by pruning them. On the contrary, $\mathcal{C}_j^L$ accounts for the majority of candidates, as demonstrated in Figure 6(b), which allows *Thres* to shine. The best method is again CPT, owing primarily to its thresholding component in this case.

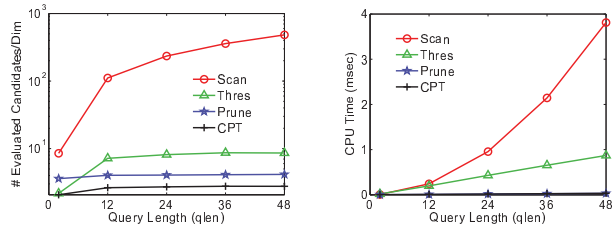
Figure 12 examines the effect of query length using KB. Here $qlen$ varies between 2 and 48 image features. As all three candidate subsets ($\mathcal{C}_j^0$, $\mathcal{C}_j^H$, and $\mathcal{C}_j^L$) are sizable, pruning and thresholding are both effective, especially when combined into CPT. Although not easily visible in Figure 12(b), the CPU cost in CPT is 37% to 40% smaller than in the runner-up, *Prune*.

Effect of Result Size: In Figure 13 we study the effect of k on performance, varying it from 10 to 80 while setting $qlen = 4$. We



(a) # Evaluated Candidates (b) CPU Cost

Figure 11: Synthetic Data, $k = 10$, varying $qlen$



(a) # Evaluated Candidates (b) CPU Cost

Figure 12: KB Dataset, $k = 10$, varying $qlen$

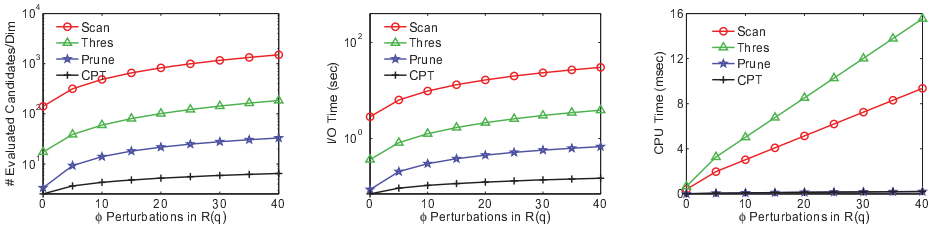
present results only for WSJ and ST as they represent two extremes; those for KB resemble WSJ more closely.

Figures 13(a) and 13(b) plot our measurements for WSJ. As expected, a larger k causes TA to reach deeper into the query lists, raising the size of $\mathcal{C}(\mathbf{q})$ and in turn *Scan*'s overheads. Interestingly, the other three methods improve with k. For *Prune* the reason is that in WSJ, the inverted lists have uneven lengths; popular terms have longer lists and rare ones shorter. For query dimensions j that correspond to rare terms, as k rises progressively to 80, there is a higher chance that all tuples with non-zero j-th coordinate are already in $\mathcal{R}(\mathbf{q})$, leaving $\mathcal{C}_j^H$ empty. This is why *Prune* processes fewer candidates with increasing k. For *Thres* the reason is different. As the result size increases, the interim immutable regions after Phase 1 get tighter (due to reorderings within the result), which allows the termination condition of *Thres* to be met earlier.

Figures 13(c) and 13(d) correspond to the synthetic dataset. Similar to Figure 11, *Prune* fails to disqualify almost any candidate (due to the correlation in data) and performs similarly to *Scan*; thus the increase in cost with k. Here CPT relies primarily on *Thres*, whose termination condition gets tighter with k.

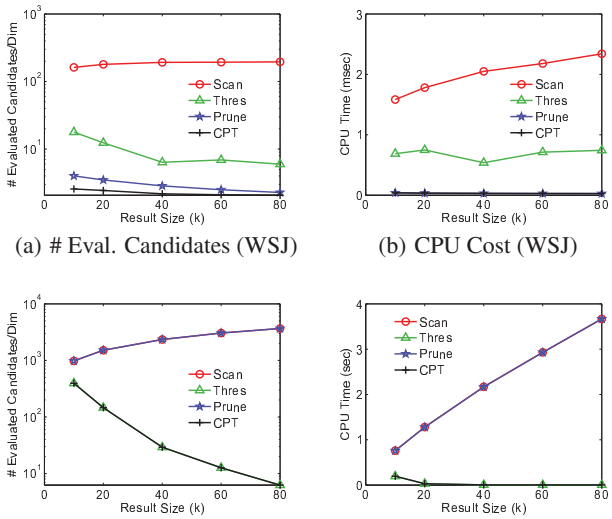
7.3 Experiments for $\phi > 0$

Next, we consider $\phi > 0$, i.e., when up to ϕ perturbations are tolerable in the result. Due to lack of space, we focus on WSJ data.



(a) # Evaluated Candidates (b) I/O Cost (c) CPU Cost

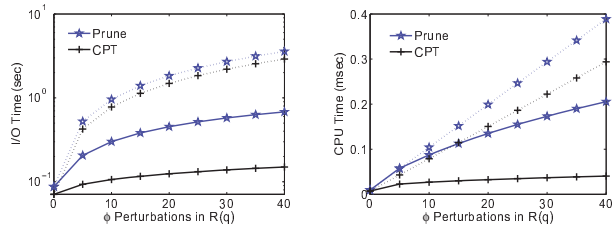
Figure 14: WSJ Corpus, $k = 10$, $qlen = 4$, varying ϕ



(a) # Eval. Candidates (WSJ) (b) CPU Cost (WSJ)

(c) # Eval. Candidates (ST) (d) CPU Cost (ST)

Figure 13: WSJ and ST Data, $qlen = 4$, varying k



(a) I/O Cost (b) CPU Cost

Figure 15: One-off versus Iterative Processing for $\phi > 0$

In Figure 14, we fix $k = 10$ and $qlen = 4$, and vary ϕ from 0 to 40. The costs of all methods increase with ϕ, and their relative performance is similar to Section 7.2 for $\phi = 0$. While a larger ϕ increases the number of evaluated candidates in *Scan* and *Thres*, *Prune* and CPT need only to cope with a slightly larger subset of $\mathcal{C}_j^0$ and $\mathcal{C}_j^H$ (each including $\phi + 1$ tuples). This is why *Scan* and *Thres* deteriorate much more rapidly than *Prune* and CPT in Figures 14(a) and 14(b). At $\phi = 0$, *Scan* and *Thres* evaluate 55.6 times and 6.8 times, respectively, more candidates than CPT. At $\phi = 40$, the gap widens to 228 times and 28 times. The experiment highlights the effectiveness of pruning for large ϕ settings.

Another key observation is that, although *Thres* examines fewer candidates (and incurs fewer I/Os) than *Scan*, it has a higher CPU cost. This is due to its overheads in forming sorted lists and repetitively checking the termination condition. Nevertheless, thresholding offers CPT a 25% to 80% reduction in CPU time compared to *Prune*; the difference between CPT and *Prune* in this experiment is clearer in Figure 15, described next.

Figure 15 compares our one-off computation approach for $\phi > 0$ against the straightforward, iterative re-evaluation of single region requests (i.e., repetitive calls of the $\phi = 0$ versions of the algorithms). We repeat the previous experiment for the two most efficient approaches, *Prune* and CPT. The dashed lines correspond to the iterative versions of the methods. The charts reveal that the techniques in Section 6 inherently share processing for neighboring immutable regions, i.e., avoid unnecessary repetition of operations.

7.4 Disregarding Reorderings within $\mathcal{R}(\mathbf{q})$

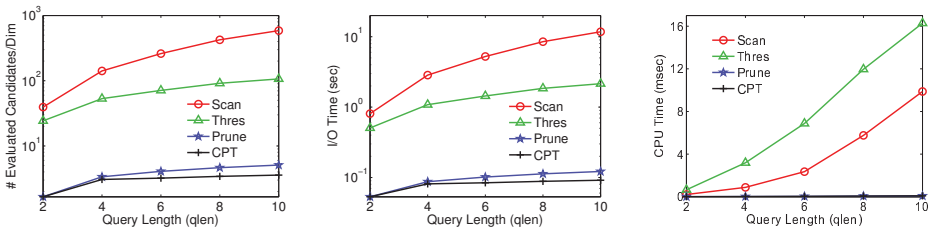
In previous experiments we considered that every change in the result, be it an update in the composition or a reordering of tuples in $\mathcal{R}(\mathbf{q})$, constitutes a valid perturbation. In this section, we evaluate the methods under the assumption that the user/application is only concerned about the composition of the top-k result, and thus reorderings in $\mathcal{R}(\mathbf{q})$ are ignored. In Figure 16 we present results on the WSJ corpus in this case, setting $\phi = 0$, $k = 10$ and varying $qlen$. Recall that processing by all algorithms is the same as previously, the difference being that Phase 1 is skipped and Phase 2 commences straightaway.

The performance here is similar to Figure 10. The most noticeable difference is that the effectiveness of *Thres* is reduced. The reason is that initialization with the widest possible $\mathcal{IR}_j$ makes the thresholding condition tougher to meet, causing *Thres* to perform additional iterations and examine more candidates. This in turn translates to higher I/O and CPU costs. Although *Thres* remains preferable to *Scan* in terms of I/O, its CPU time is longer, i.e., the reduction in candidates evaluated is outweighed by its overheads (mainly in list formation/probing and threshold checking). However, thresholding still enhances CPT, which achieves smaller I/O and CPU costs compared to *Prune*.

7.5 Summary of Experiment Results

The main conclusions drawn from the evaluation are:

1. Candidate pruning is most effective when there is no significant correlation among the query dimensions.
2. Candidate thresholding successfully reduces the number of evaluated candidates in all scenarios.
3. Candidate pruning and candidate thresholding complement each other, rendering CPT the best performer.
4. Although we used disk-resident data, the CPU charts indicate that our techniques achieve significant performance improvements in a memory-based setting as well.



(a) # Evaluated Candidates (b) I/O Cost (c) CPU Cost

Figure 16: WSJ Corpus, Disregarding Reorderings, $\phi = 0$, $k = 10$, varying $qlen$

8. CONCLUSION

This paper is the first study on immutable regions for top-k queries. The immutable regions define ranges of adjustment to the weights of the various decision variables, inside which the query result remains unchanged. We consider subspace top-k queries over high-dimensional data indexed by inverted lists. We observe that the costs incurred in computing immutable regions are determined primarily by the examination of non-result tuples (in order to ensure that after weight adjustment their scores do not exceed that of any result tuple). To reduce these costs, we introduce pruning and thresholding techniques that allow immutable regions to be derived correctly by processing just a small fraction of non-result tuples. The two techniques are combined into a robust algorithm, CPT, with superior performance for various data distributions. Experiments on real datasets from different domains, as well as on synthetic data, show that CPT incurs 2 to 500 times smaller I/O and CPU costs compared to a baseline approach.

9. REFERENCES

[1] R. Baeza-Yates and B. R. Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[2] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, 3rd ed. edition, 2008.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

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

[5] Y.-C. Chang, L. D. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: Indexing for linear optimization queries. In *SIGMOD*, pages 391–402, 2000.

[6] S. Chaudhuri, L. Gravano, and A. Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE Trans. Knowl. Data Eng.*, 16(8):992–1009, 2004.

[7] R. Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58(1):83–99, 1999.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. *JCSS*, 66(4):614–656, 2003.

[9] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *VLDB Journal*, 13(1):49–70, 2004.

[10] M. Hua, J. Pei, W. Zhang, and X. Lin. Ranking queries on uncertain data: a probabilistic threshold approach. In *SIGMOD*, pages 673–686, 2008.

[11] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB Journal*, 13(3):207–221, 2004.

[12] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[13] I. Kemelmacher and R. Basri. Indexing with Unknown Illumination and Pose. *CVPR*, 1:909–916, 2005.

[14] J. Li and A. Deshpande. Ranking continuous probabilistic datasets. *PVLDB*, 3(1):638–649, 2010.

[15] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD*, pages 635–646, 2006.

[16] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. The v*-diagram: a query-dependent approach to moving knn queries. *PVLDB*, 1(1):1095–1106, 2008.

[17] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of skyline: A semantic approach based on decisive subspaces. In *VLDB*, pages 253–264, 2005.

[18] M. Persin. Efficient implementation of text retrieval techniques. *Tech. rep. (thesis), RMIT, Australia*, 1996.

[19] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Trans. Computers*, 51(10):1124–1140, 2002.

[20] M. A. Soliman, I. F. Ilyas, D. Martinenghi, and M. Tagliasacchi. Ranking with uncertain scoring functions: semantics and sensitivity measures. In *SIGMOD*, pages 805–816, 2011.

[21] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD*, pages 79–96, 2001.

[22] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou. Branch-and-bound processing of ranked queries. *Inf. Syst.*, 32(3):424–445, 2007.

[23] Y. Tao, X. Xiao, and J. Pei. Subsky: Efficient computation of skylines in subspaces. In *ICDE*, page 65, 2006.

[24] Y. Tao, X. Xiao, and J. Pei. Efficient skyline and top-k retrieval in subspaces. *IEEE Trans. Knowl. Data Eng.*, 19(8):1072–1088, 2007.

[25] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. In *ICDE*, pages 277–288, 2003.

[26] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *ICDE*, pages 365–376, 2010.

[27] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *ICDE*, pages 189–200, 2003.

[28] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *SIGMOD*, pages 443–454, 2003.