2012 IEEE 28th International Conference on Data Engineering

Upgrading Uncompetitive Products Economically

Hua Lu[†] Christian S. Jensen[‡]

[†]*Department of Computer Science, Aalborg University, Denmark*
luhua@cs.aau.dk
[‡]*Department of Computer Science, Aarhus University, Denmark*
csj@cs.au.dk

***Abstract*—The skyline of a multidimensional point set consists of the points that are not dominated by other points. In a scenario where product features are represented by multidimensional points, the skyline points may be viewed as representing competitive products. A product provider may wish to upgrade uncompetitive products to become competitive, but wants to take into account the upgrading cost. We study the top-k product upgrading problem. Given a set P of competitor products, a set T of products that are candidates for upgrade, and an upgrading cost function f that applies to T, the problem is to return the k products in T that can be upgraded to not be dominated by any products in P at the lowest cost. This problem is non-trivial due to not only the large data set sizes, but also to the many possibilities for upgrading a product. We identify and provide solutions for the different options for upgrading an uncompetitive product, and combine the solutions into a single solution. We also propose a spatial join-based solution that assumes P and T are indexed by an R-tree. Given a set of products in the same R-tree node, we derive three lower bounds on their upgrading costs. These bounds are employed by the join approach to prune upgrade candidates with uncompetitive upgrade costs. Empirical studies with synthetic and real data show that the join approach is efficient and scalable.**

I. Introduction

A. Overview

The skyline [3] of a d-dimensional point set S is those points that are not dominated by any other point in S. Point p_1 dominates point P_2 if p_1 is no worse than p_2 in all d dimensions and better than p_2 in at least one dimension. Skyline points are of interest because they are better in a meaningful sense than non-skyline points.

In a context where product features are represented by d-dimensional points, skyline points represent potentially competitive products. Refer to the cell phone example in Table I. Here, phones 1, 3, and 5 are skyline points, as no other phone is better in terms of all the three quality attribute: weight, standby time, and camera pixel size. Thus, phones 1, 3, and 5 are attractive to consumers because they offer better quality.

A range of approaches [2], [3], [5], [7]–[9], [12]–[14], [18] have been proposed to efficiently compute the skyline from a given, fixed data set. We consider a related, but different problem: Given a set P of competitor products, a set T of uncompetitive products to be considered for upgrading, and an upgrading cost function f defined for products in T, we identify the k products in T that have the lowest costs of being upgraded to not being dominated by products in P.

TABLE I
Cell Phone Set P

Phone	Weight	Standby Time	Camera Pixel
phone 1	140	200 hours	2.0 megapixels
phone 2	180	150 hours	3.0 megapixels
phone 3	100	160 hours	3.0 megapixels
phone 4	180	180 hours	3.0 megapixels
phone 5	120	180 hours	4.0 megapixels
phone 6	150	150 hours	3.0 megapixels

B. Motivating Example

Assume that we have a cell phone set T, shown in Table II, from a particular manufacturer. Each phone in T is dominated by a phone in Table I (P). In particular, phone A is dominated by phones 1, 3, 5, and 6, phone B is dominated by all phones in P, phone C is dominated by all phones in P save phone 1, and phone D is dominated by phones 1, 4, and 5.

TABLE II
Cell Phone Set T

Phone	Weight	Standby Time	Camera Pixel
phone A	150	120 hours	2.0 megapixels
phone B	180	130 hours	1.0 megapixels
phone C	180	120 hours	3.0 megapixels
phone D	220	180 hours	2.0 megapixels

For the manufacturer, it is of interest to provide competitive products instead of those in T. Instead of designing brand new products from scratch, it is considered more economical to upgrade existing products. This enables to reuse the legacy designs and product lines, and it shortens the learning curves for workers, to name but two potential benefits.

The question is then which existing product or products in T the manufacturer should choose to upgrade. An upgraded phone is expected to not be dominated by any phone in P. Also, it is expected that the cost of upgrading a phone is minimized. For example, phone D is currently heavy compared to the other phones. To decrease its weight, an upgrading cost is implied: the lighter the phone becomes, the higher its cost. Such an upgrading costs also applies to other quality attributes. For example, to equip phone C with a longer-lasting battery, the cost will go up.

In other words, the phone manufacturer needs to consider also how to upgrade an uncompetitive product, given a set of upgrading cost constraints on those quality attributes in consideration.

We call the problem described above the *product upgrading*

1084-4627/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDE.2012.92

problem. It is intuitively relevant and useful in applications other than manufacturing. For example, hotels can be described by multiple quality attributes such as their *star rating*, *room size*, *rental price*, etc. For a hotel chain to stay profitable, it may consider to upgrade its uncompetitive hotels. Again, it is desirable to upgrade the hotel or hotels that need the lowest cost to become competitive.

The product upgrading problem is non-trivial due to at least two reasons. First, both the uncompetitive product set T and the competitor set P can be very large, which renders naive, iterative approaches inefficient at finding suitable products from T to upgrade. Second, a product t in T may be dominated by many products in P, so how to upgrade a single t in terms of P and all quality attributes is non-trivial.

We proceed to cover preliminaries and to state the problem formally.

C. Preliminaries and Problem Statement

*Definition 1: (**Product Space**)* A *product space* is a c-dimensional space $\mathcal{D} = D_1 \times D_2 \times \ldots \times D_c$, where each dimension D_i $(1 \leq i \leq c)$ is the domain of a corresponding product attribute.

We use $\mathbb{D} = \{D_1, \ldots, D_c\}$ as the set of all dimensions.

*Definition 2: (**Product**)* A *product* is a c-dimensional point $p = (d_1, d_2, \ldots, d_c) \in \mathcal{D}$, where d_i's is p's i'th attribute and $d_i \in D_i$ $(1 \leq i \leq c)$.

*Definition 3: (**Dominance**)* Given two products p and p', p *dominates* p' if p is no worse than p' on all attributes and p is better than p' on at least one attribute. We use $p \prec p'$ to denote that p dominates p'.

Above "worse" can mean "smaller" or "larger," depending on the attribute semantics. For example, "worse" means "smaller" for the phone attributes standby time and camera pixel because consumers prefer longer standby times and higher camera resolutions. In contrast, "worse" means "larger" for phone weights because consumers prefer lighter phones.

*Definition 4: (**Attribute Cost Function**)* Given a product attribute domain D, an *attribute cost function* f_a maps an attribute value in D to a real value:

$$f_a : D \rightarrow \mathcal{R}$$

*Definition 5: (**Product Cost Function**)* A *product cost function* f_p maps a product in $\mathcal{D}$ to a real value:

$$f_p : \mathcal{D} \rightarrow \mathcal{R}$$

Given a product space $\mathcal{D}$, and c attribute cost functions $f_a^1, f_a^2, \ldots, f_a^c$, there may be different ways to define a product cost function that captures the overall cost of manufacturing the products in $\mathcal{D}$.

*Definition 6: (**Integration Function**)* An *integration function* $\mathcal{F}^{int} : (D \rightarrow \mathcal{R})^c \rightarrow \mathcal{R}$ integrates c attribute cost functions $f_a^1, f_a^2, \ldots, f_a^c$ to a product cost function:

$$f_p = \mathcal{F}^{int}(f_a^1, f_a^2, \ldots, f_a^c)$$

The integration function $\mathcal{F}^{int}$ can take different forms for different product types. For example, for some particular product types, the overall cost of a product p may be the sum of all attribute costs. Such cases are captured as the summation integration function $\mathcal{F}^{sum}$ as follows.

$$f_p^{sum}(p) = \mathcal{F}^{sum}(f_a^1(p.d_1), \ldots, f_a^c(p.d_c)) = \sum_{i=1}^{c} f_a^i(p.d_i) \quad (1)$$

Weights may be used for attributes in summation to differentiate the effects of attribute cost functions.

$$f_p^{wgt}(p) = \mathcal{F}^{wgt}(f_a^1(p.d_1), \ldots, f_a^c(p.d_c)) = \sum_{i=1}^{c} w_i \cdot f_a^i(p.d_i)$$

*Definition 7: (**Upgrading Cost**)* Given a competitor product set $P \subseteq \mathcal{D}$ and a product cost function f_p, the *upgrading cost* of a product t with respect to P and f_p is the cost to upgrade t to t' such that t' is not dominated by any product in P. We use $cost_{up}(t, P, f_p)$ to denote this upgrading cost, and $cost_{up}(t, P, f_p) = f_p(t') - f_p(t)$.

*Problem Statement: (**Top-k Product Upgrading Problem**)* Given product sets $P \subseteq \mathcal{D}$ and $T \subseteq \mathcal{D}$ and a corresponding product cost function f_p, return a k-subset T' of T such that $\forall t \in T'$ and $\forall t^* \in T \backslash T'$, $cost_{up}(t, P, f_p) \leq cost_{up}(t^*, P, f_p)$.

The top-k product upgrading problem finds the k products in T that can be upgraded in the most economical way to not be dominated by products in P. A given tuple t can be updated in multiple ways, each of which has its own upgrading cost. Given a cost function, we are interested in upgrading potential products with the minimum costs.

For simplicity, we assume that smaller values are preferred on each dimension in dominance comparison.[1] All techniques and algorithms in this paper are able to handle the general case.

We also assume that all cost functions in this paper are monotonic in the sense that $f_p(p_1) \geq f_p(p_2)$ if $p_1 \prec p_2$. This corresponds to the intuition that a dominating product usually costs more than a dominated product. We leave possible non-monotonic cost functions for future work.

No algorithm exists for the top-k product upgrading problem. In this paper, we propose two approaches to tackle the problem. The probing approach requires that competitor set P to be indexed by an R-tree and calculates the upgrading cost for each product in T iteratively. It finally returns the k products from T with the lowest upgrading costs. This approach serves as a baseline.

The join approach requires that both sets P and T are indexed by an R-tree. It joins nodes from either tree, estimates the lower bound upgrading cost for a set of T products that form an R-tree node, and it returns top products to upgrade on the fly. One of the desirable benefits of the join approach is its progressiveness: it can return top-k products incrementally without processing the entire T set; it can be stopped early once the user has a sufficient number k of products to upgrade.

[1]For a dimension on which large values are preferred, a simple negation conversion can be applied.

D. Contributions and Paper Organization

The paper's contributions are threefold. First, we formally define a new problem, the product upgrading problem. Solutions to this problem identify uncompetitive products that can be upgraded to be attractive at low costs. Second, we offer two solutions for the product upgrading problem. In particular, a probing approach works as a baseline, and a join-based approach exploits effective result pruning techniques. The pruning techniques are enabled by three lower bounds on the upgrading costs for a group of products. Third, we conduct an extensive empirical study to evaluate our proposals on both real and synthetic data sets. The results demonstrate that the join approach is efficient and scalable in a range of settings.

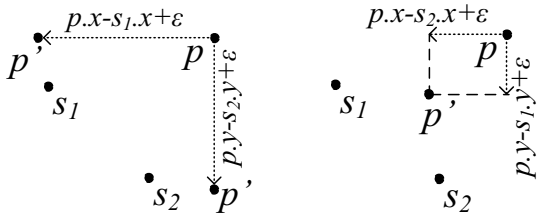
The remainder of this paper is organized as follows. Section II addresses how to upgrade a single given product in the presence of a set of competitors. Section III proposes the algorithms for choosing the top-k products that can be upgraded most economically. Section IV reports on an extensive empirical study with the algorithms. Section V reviews related work. Section VI concludes the paper and covers several research directions.

II. Upgrading A Single Product

A. Overview

Given a set S of skyline points and a dominated point p, there are two basic ways for p to become a non-dominated point p' that then belongs to S.

First, we can pick one dimension and give p the smallest value among all skyline points in that dimension. Figure 1(a) shows an example where point p is dominated by two skyline points s_1 and s_2. We decrease p's value on x axis by $p.x - s_1.x + \epsilon$ so that p' has the best x value when compared to s_1 and s_2, which makes p' a skyline point. Here, ϵ is a small positive value. Likewise, we can instead decrease p's value on y axis by $p.y - s_2.y + \epsilon$ and make p' enter the skyline.



(a) One Dimension (b) Multiple Dimensions

Fig. 1. Upgrading Costs

Upgrading a single dimension is simple, but it may incur high upgrading cost if the selected dimension is cost-inefficient and the difference between p and existing dominators on the dimension is large. Even on a dimension that is not cost-inefficient, being better than all skyline points can still be very expensive if this requires a substantial attribute value change for a product that is considered for upgrading.

Alternatively, we can tune p on several dimensions such that no point in S dominates it. In Figure 1(b), we can decrease p's x value by $p.x - s_2.x + \epsilon$ and its y value by $p.y - s_1.y + \epsilon$ such that neither s_1 nor s_2 dominates p'. Changing multiple attribute values may incur a lower upgrading cost than changing only one, but extra computation is needed to choose the most economical dimensions to upgrade.

We proceed to give an algorithm that upgrades a single product by taking into account both of the two possibilities described above.

B. Algorithm for Upgrading a Single Product

The pseudo code of upgrading a single product is shown in Algorithm 1. It takes as input an uncompetitive product p, a skyline set S, and a cost function f_p. It considers each dimension in turn. First, all skyline points are sorted ascendingly with respect to the current dimension (line 2–3). After that, it attempts to upgrade p by changing its attribute value on the current dimension, and records this upgrade it is less expensive than the currently known upgrading cost (lines 4–7). Note that $f_p.f_a^k$ is the attribute cost function on dimension D_k.

Algorithm 1 upgrade(Skyline point set S, product p, product cost function f_p)

```
1:  cost ← ∞;  p' ← p
2:  for each dimension D_k do
3:      sort S in ascending order of the D_k values
4:      s ← S[1]                ▷ get the one with the minimum D_k value
5:      if f_p.f_a^k(s.d_k − ε) − f_p.f_a^k(p.d_k) < cost then
6:          cost ← f_p.f_a^k(s.d_k − ε) − f_p.f_a^k(p.d_k)
7:          p'.d_k ← s.d_k − ε
8:      for each i from 1 to |S| − 1 do
9:          s_i ← S[i];  s_j ← S[i + 1];
10:         for each x from 1 to c do
11:             if x = k then p''.d_x ← s_j.d_x − ε
12:             else
13:                 p''.d_x ← s_i.d_x − ε
14:         if f_p(p'') − f_p(p) < cost then
15:             cost ← f_p(p'') − f_p(p)
16:             p' ← p''
17: return cost, p'
```

It continues to consider multiple dimensions on which p's old attribute values can be upgraded (lines 8–16). In particular, every pair of skyline points s_i and s_j that are consecutive in the current dimension are selected (lines 8–9), and new attribute values (after upgrading) are chosen with respect to s_i on each dimension (lines 10–13). If this yields a cheaper upgrading, the upgrading cost and the upgraded project are recorded (lines 14–16).

The correctness of the algorithm is guaranteed by Lemma 1.

Lemma 1: Algorithm 1 returns an upgraded product p' that is not dominated by any point in S.

Proof: If p' is decided by lines 4–7, i.e. $p'.d_k$ is equal to $s.d_k - \epsilon$, $\forall s \in S$ we have $p'.d_k < s.d_k$. Therefore, no point $s \in S$ can dominate p'.

Otherwise, p' is decided by lines 8–16. All points in S are sorted and divided into three parts by s_i: those ones before s_i, s_i, and those after s_i. We prove each part in turn.

For any point s that is after s_i on the sorting dimension D_k, we have $s.d_k \geq s_j.d_k \geq s_i.d_k$. Thus, $p'.d_k = p''.d_k = s_j.d_k - \epsilon < s_j.d_k \leq s.d_k$ (line 11). Therefore, no such point s can dominate p'.

For point s_i, on any dimension $D_x \neq D_k$ we have $p'.d_x = p''.d_x = s_i.d_x - \epsilon < s_i.d_x$ (line 13); therefore, s_i cannot dominate p'.

For any point s that is before s_i, we have $s.d_k \leq s_i.d_k$ on the sorting dimension D_k. If $s = s_i$ then s cannot dominate p' as proved above. If $s \neq s_i$, there must exist a dimension $D_x \neq D_k$ such that $s.d_x \geq s_i.d_x$ because otherwise, s would dominate s_i on all dimensions. Consequently, we have $p'.d_x = p''.d_x = s_i.d_x - \epsilon < s_i.d_x \leq s.d_x$ (line 13), and therefore s cannot dominate p'. ■

Algorithm 1 chooses the cheapest way to upgrade a product t among the alternatives it considers. It will be used in our algorithms for top-k product upgrading problem in Section III.

III. Algorithms for Top-k Product Upgrading Problem

We assume that the competitor set P is large and is indexed by an R-tree R_P. We propose two algorithms: one that does not require the uncompetitive product set T to be indexed and one that requires T to also be indexed by an R-tree.

A. Probing Algorithms

The pseudo code for a basic probing algorithm is shown in Algorithm 2. For each product t from T, the algorithm gets all the dominators from P by a range query (line 3) whose range is exactly t's anti-dominant region $ADR(t)$ [15], the hyper-rectangle determined with t as the maximum corner and the origin as the minimum corner. Subsequently, the skyline S of all dominators is obtained, and the upgrading cost is computed by calling Algorithm 1 (lines 4–5). During the iterations, only the product with the currently lowest upgrading cost is recorded, and the lowest cost and the corresponding original and upgraded products are finally returned (lines 6–8).

Note that it is straightforward to make minor changes to the algorithm such that the top-k products to upgrade are returned.

Algorithm 2 basicProbing(Competitor set P's R-tree R_P, product set T, product cost function f_p)

```
1: cost_min ← ∞;  t_min ← null;  t' ← null
2: for each product t ∈ T do
3:     dominators ← RangeQuery(R_P, ADR(t))
4:     get dominators's skyline S
5:     (cost, t'') ← upgrade(S, t, f_p)
6:     if cost < cost_min then
7:         cost_min ← cost;  t_min ← t;  t' ← t''
8: return cost_min, t_min, t'
```

The basic probing algorithm retrieves all competitors in t's anti-dominant region $ADR(t)$. This is actually unnecessary as only the skyline points in that region is needed. An improved probing algorithm is obtained by replacing lines 3–4 in Algorithm 2 with a call getDominatingSky(R_P, t) of Algorithm 3. This algorithm integrates a local skyline computation into a range query. The idea of the BBS algorithm [13] is adapted here, by considering only those R-tree nodes whose MBRs (minimum bounding rectangle) overlap with $ADR(t)$ (lines 3 and 12). Note that $e.min$ denotes the minimum corner of an entry e's MBR.

Algorithm 3 getDominatingSky(Competitor set P's R-tree R_P, product t)

```
1: S ← ∅
2: initialize a min-heap H
3: if R_P.root's MBR overlaps with ADR(t) then
4:     enheap(H, ⟨R_P.root, 0⟩)
5: else
6:     return S
7: while H is not empty do
8:     e ← deheap(H)
9:     if e's MBR is not dominated by S then
10:        if e is a non-leaf entry then
11:            for each child e' of e do
12:                if e''s MBR overlaps with ADR(t) and
                       e''s MBR is not dominated by S then
13:                    enheap(H, ⟨e', Σ_{i=1}^{c} e'.min.d_i⟩)
14:        else
15:            add e's point to S
16: return S
```

In the example in Figure 2, four R-tree nodes from R_P lie in $ADR(t)$. The basic probing algorithm retrieves all points that are in these four nodes through a range query. In contrast, by calling getDominatingSky in Algorithm 3, the improved probing algorithm prunes nodes indexed by entries e_2, e_3, and e_4 because all points in them are dominated by some point(s) from the node indexed by e_1. This example shows that the improved probing algorithm can save considerable computation.

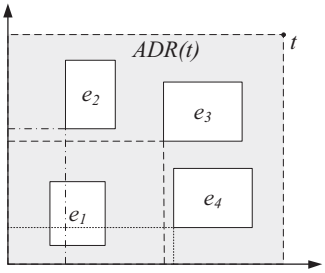


Fig. 2. Example of Improved Probing

B. Join-Based Approach

1) Motivation: The probing algorithms are not efficient as each product is processed in isolation. When the product set T is large, processing all products in T incurs a considerable computation cost. In such a case, it is beneficial if we can derive a lower bound on the upgrading cost for a single product in T or for a group of products in T. This way, once we find that one particular product can be upgraded with a lower cost than the lower bound for one or several products, we can stop processing these unpromising products. In addition, we can

prioritize the processing of all products or product groups, based on their corresponding lower bound upgrading costs.

Motivated by these observations, we index T by an R-tree. As a result, each R-tree node refers to a group of products from T. We then adapt a spatial join algorithm [4] to process products in batch, and we prune unpromising products effectively.

Section III-B2 gives an overview of the join approach. Section III-B3 derives the lower bound upgrading cost for a group of products in e_T and a group of competitors in e_P, where e_T and e_P are two R-tree node entries that are matched by the join. Section III-B4 derives the lower bound upgrading cost for a group of products in e_T and a join entry list $e_T.JL$ that contains all e_T's possible matches in the join. Section III-B5 presents the join algorithm.

2) Join-Based Approach Overview: Let the uncompetitive product set T be indexed by an R-tree R_T. Given a node entry e_T from R_T, we only need to access those R-tree nodes from R_P that may have competitor products that can dominate products in e_T. In other words, we ignore any R_P node whose MBR is outside of the reverse dominating region of $e_T.max$, i.e., the maximum corner of e_T's MBR. Formally, a node entry e_P from R_P can be ignored, if $\exists D_i \in \mathbb{D}$ such that $e_P.min.d_i > e_T.max.d_i$.

In the example in Figure 3(c), the R_P nodes indexed by entries e_{P1} and e_{P2} are discarded safely because they cannot contain any competitor products that dominate a product in e_T.

When we process an entry e_T from R-tree R_T, it may be joined with multiple entries from R-tree R_P. We use $e_T.JL$ to denote all such R_P entries and call it e_T's *join list*. For any two entries $e_{Pi} \neq e_{Pj}$ in $e_T.JL$, $e_{Pi}.max$ does not dominate $e_{Pj}.min$. Otherwise, no point in e_{Pj} can be a skyline point as it must be dominated by $e_{Pi}.max$, and thus must be dominated by at least one point in e_{Pi} according to the transitivity of the dominance relationship. This property can be used to reduce $e_T.JL$ when we process e_T.

We proceed to derive a lower bound on the cost of upgrading points in entry e_T. Given an entry e_T from R_T and an entry e_P from R_P, we use $LBC(e_T, e_P)$ to indicate the lower bound cost for upgrading any point $t \in e_T$ with respect to a single R_P entry e_P. Specifically, for any point $t \in e_T$ to become undominated by all points in e_P, we have to spend at least a cost of $LBC(e_T, e_P)$ to upgrade t.

3) The Lower Bound Upgrading Cost $LBC(e_T, e_P)$: We distinguish among three cases according to the topological relationship between entry e_T and an entry e_P from R_P. In each case, we consider the cost to upgrade the minimum corner of e_T, i.e., $e_T.min$, although there may be no point in the corner. The cost to upgrade such a (virtual) point must be the lowest among all costs of upgrading points in e_T since $e_T.min$ dominates all points in e_T.

When comparing the MBRs of e_T and e_P, we classify all e_T's dimensions in $\mathbb{D} = \{D_1, \ldots, D_c\}$ into three categories.

- Disadvantaged dimensions:
  $\mathbb{D}_D = \{D_i \in \mathbb{D} \mid e_P.max.d_i < e_T.min.d_i\}$
- Incomparable dimensions:
  $\mathbb{D}_I = \{D_i \in \mathbb{D} \mid e_P.min.d_i \leq e_T.min.d_i \leq e_P.max.d_i\}$
- Advantaged dimensions:
  $\mathbb{D}_A = \{D_i \in \mathbb{D} \mid e_T.min.d_i < e_P.min.d_i\}$

We use $Dims(\mathbb{D}, e_T, e_P)$ to indicate the classification with respect to e_T and e_P. For the sake of conciseness, we use $Dims(\mathbb{D}, e_T, e_P).\mathbb{D}_X$ to denote the corresponding dimension set $\mathbb{D}_X$ where $X \in \{D, I, A\}$.

When we upgrade e_T, it is apparent that we do not need to refine any of $e_T.min$'s dimensions if $e_T.min$ has an advantaged dimension. In fact, $e_T.min$'s value on that advantaged dimension is sufficient to render it not dominated by any point in e_P. On the other hand, as an optimistic estimate, we also maintain $e_T.min$'s incomparable dimension values when we upgrade $e_T.min$.

The lower bound $LBC(e_T, e_P)$ is derived according to e_T's dimension composition with respect to e_P.

- **[Case 1]** $\mathbb{D}_A \neq \emptyset$:
  $LBC(e_T, e_P) = 0$
- **[Case 2]** $\mathbb{D}_A = \emptyset$, $\mathbb{D}_D = \emptyset$, and $\mathbb{D}_I = \mathbb{D}$:
  $LBC(e_T, e_P) = 0$
- **[Case 3]** $\mathbb{D}_A = \emptyset$, $\mathbb{D}_I = \emptyset$ and $\mathbb{D}_D = \mathbb{D}$:
  $LBC(e_T, e_P) = f_p(e_P.max) - f_p(e_T.min)$
- **[Case 4]** $\mathbb{D}_A = \emptyset$, $\mathbb{D}_I \neq \emptyset$ and $\mathbb{D}_D \neq \emptyset$:
  $LBC(e_T, e_P) = f_p(t_v) - f_p(e_T.min)$, where
  $t_v.d_i = \begin{cases} e_P.max.d_i, & \text{if } D_i \in \mathbb{D}_D \\ e_T.min.d_i, & \text{if } D_i \in \mathbb{D}_I \end{cases}, i = 1, 2, \ldots, c.$

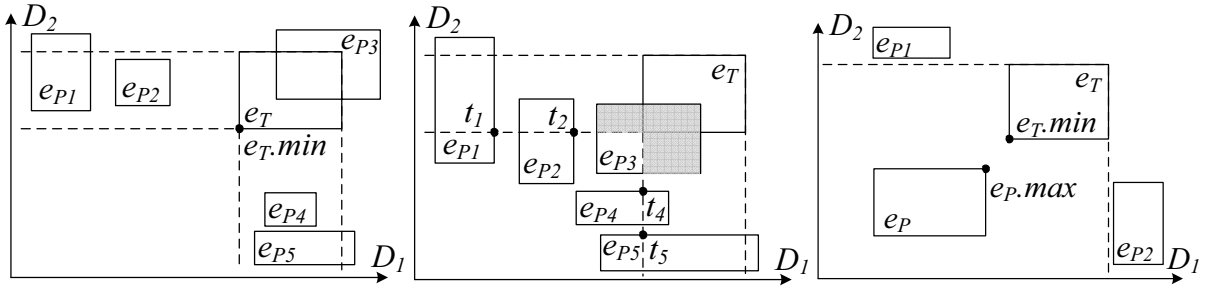
Refer to Figure 3(a). With respect to R_P entry e_{P3}, both dimensions of R_T entry e_T are in $\mathbb{D}_A$, i.e., $\mathbb{D}_A = \{D_1, D_2\}$. Therefore, $LBC(e_T, e_{P3}) = 0$. Further, $\mathbb{D}_A = \{D_2\}$ with respect to entries e_{P1} and e_{P2}, and $\mathbb{D}_A = \{D_1\}$ with respect to entries e_{P4} and e_{P5}. Therefore, $LBC(e_T, e_{Pi}) = 0$ where $i = 1, 2, 4, 5$. These zero lower bound values are consistent with $e_T.min$ not being dominated by any point in the five R_P entries.

Next, consider Figure 3(b). With respect to R_P entry e_{P3}, R_T both dimensions of entry e_T are in $\mathbb{D}_I$, i.e., $\mathbb{D}_I = \{D_1, D_2\}$. Therefore, $LBC(e_T, e_{P3}) = 0$. This is consistent with the possibility that e_{P3} contains points only in its shaded part as illustrated, where $e_T.min$ is not dominated by any point from e_{P3}.

Now refer to Figure 3(c). With respect to R_P entry e_P, both dimensions of R_T entry e_T are in $\mathbb{D}_D$, i.e., $\mathbb{D}_D = \{D_1, D_2\}$. Therefore, $LBC(e_T, e_P) = f_p(e_P.max) - f_p(e_T.min)$. This is because, in the best case, $e_T.min$ must be as good as $e_P.max$ such that $e_T.min$ is not dominated by any point from e_P.

Consider again Figure 3(b). With respect to R_P entry e_{P1} or e_{P2}, $\mathbb{D}_D = \{D_1\}$ and $\mathbb{D}_I = \{D_2\}$ for e_T; whereas with respect to e_{P4} or e_{P5}, $\mathbb{D}_D = \{D_2\}$ and $\mathbb{D}_I = \{D_1\}$ for e_T. As a result, $LBC(e_T, e_{Pi}) = f_p(t_i) - f_p(e_T.min)$, where $i = 1, 2, 4, 5$ and t_i are illustrated as dots. In the best case, $e_T.min$ must be as good as t_i such that $e_T.min$ is not dominated by any point from e_{Pi} $(i = 1, 2, 4, 5)$.

Next, we discuss how to derive a lower bound upgrading cost for an R_T entry e_T with respect to multiple R_P entries in



(a) Case 1 (b) Cases 2 and 4 (c) Case 3

Fig. 3. Lower Bound Cases

$e_T.JL$. We use $LBC(e_T, e_T.JL)$ to denote this overall lower bound upgrading cost.

4) The Lower Bound Upgrading Cost $LBC(e_T, e_T.JL)$: A naive method is to use the minimum lower bound across all R_P entries in $e_T.JL$, i.e.,

$$LBC_N(e_T, e_T.JL) = \min_{e \in e_T.JL}(LBC(e_T, e)) \quad (2)$$

Due to the property of lower bounds, this yields a correct, but also pessimistic, bound. We call it the *Naive Lower Bound*.

Observe that Cases 1 and 2 described in Section III-B3 can be ignored when Cases 3 or 4 are available. The following lemma states this formally.

Lemma 2: Given an entry e_T from R-tree R_T and its join list $e_T.JL$, $LBC(e_T, e_T.JL) > 0$ if $\exists e_P \in e_T.JL$ such that $LBC(e_T, e_P) > 0$.

Proof: Given an entry $e_P \in e_T.JL$ that satisfies $LBC(e_T, e_P) > 0$, for any product $t \in e_T$, its upgrading cost is no lower than $LBC(e_T, e_P)$. This is guaranteed by the definition of $LBC(e_T, e_P)$. Thus, any such product t has a non-zero upgrading cost. This implies $LBC(e_T, e_T.JL) > 0$. ■

The above lemma says that as long as there exists a join entry $e_P \in e_T.JL$ such that $LBC(e_T, e_P) > 0$, the overall $LBC(e_T, e_T.JL)$ must also be non-zero. As a result, we can obtain a tighter lower bound as the minimum of all non-zero lower bound upgrading costs with respect to individual entries in $e_T.JL$. We first define $e_T.JL'$ as follows.

$$e_T.JL' = \{e \in e_T.JL \mid LBC(e_T, e) > 0\}$$

We then define the following *Conservative Lower Bound*.

$$LBC_C(e_T, e_T.JL) = \min_{e \in e_T.JL'}(LBC(e_T, e)) \quad (3)$$

A more aggressive bound can be obtained from a careful study of the dimension composition case type of e_T with respect to each $e_P \in e_T.JL'$. As described, we can classify all dimensions in $\mathbb{D}$ into categories $\mathbb{D}_D$, $\mathbb{D}_I$, and $\mathbb{D}_A$. Based on this classification, we create a partitioning $\mathcal{E}_P$ of the set $e_T.JL'$ as follows.

- $\mathcal{E}_P = \{E_{P_1}, \ldots, E_{P_g}\}$, $1 \le g \le |e_T.JL'|$
- $\bigcup \mathcal{E}_P = e_T.JL'$
- $E_{P_i} \cap E_{P_j} = \emptyset$ if $E_{P_i} \in \mathcal{E}_P$, $E_{P_j} \in \mathcal{E}_P$, and $E_{P_i} \neq E_{P_j}$
- $\forall E_P \in \mathcal{E}_P$, $\forall e_{P_i}, e_{P_j} \in E_P$, we have $Dims(\mathbb{D}, e_T, e_{P_i}).\mathbb{D}_X = Dims(\mathbb{D}, e_T, e_{P_j}).\mathbb{D}_X$, where $X \in \{D, I, A\}$.

Two entries e_{P_i} and e_{P_j} in $e_T.JL'$ are in a same partition in $\mathcal{E}_P$ if they lead to the same $\mathbb{D}_D$ ($\mathbb{D}_I$, and $\mathbb{D}_A$). Otherwise, they are in different partitions. Using partitioning $\mathcal{E}_P$, we obtain a lower bound as follows.

$$LBC_A(e_T, e_T.JL) = \min_{E_P \in \mathcal{E}_P}(\max_{e \in E_P}(LBC(e_T, e))) \quad (4)$$

We call this bound *Aggressive Lower Bound*.

Lemma 3: The aggressive lower bound LBC_A gives a lower bound of the upgrading cost with respect to all entries in $e_T.JL$.

Proof: The correctness hinges on two aspects.

First, in each partition E_P in $\mathcal{E}_P$, $\max_{e \in E_P}(LBC(e_T, e))$ is the cost of upgrading any product t in e_T. This is due to the construction of $\mathcal{E}_P$. Specifically, in each E_P, all products in e_T are dominated by competitors on the exactly same set of dimensions. Therefore, in order to upgrade any product $t \in e_T$ to be non-dominated, we have to consider the most dominating competitor with respect to E_P. Such a competitor, with respect to E_P, is common to all products in e_T.

Second, all partitions E_Ps in $\mathcal{E}_P$ are incomparable in that they imply different sets of dimensions on which products in e_T are disadvantaged. Further, if a product t is upgraded to be non-dominated with respect to a particular partition $E_P \in \mathcal{E}_P$, it cannot be dominated by any other points with respect to any other partition in $\mathcal{E}_P$.

As a result, the cost of upgrading an arbitrary product $t \in e_T$ is at least $\min_{E_P \in \mathcal{E}_P}(\max_{e \in E_P}(LBC(e_T, e)))$. This proves the lemma. ■

5) Join Algorithm: The pseudo code of the join approach is shown in Algorithm 4. It takes as input R-trees R_P and R_T (for the competitor set P and product set T, respectively) and a cost function. A min-heap is used to order the access to R-tree entries in R_T (line 1). Initially, we join R_T's root with the singleton list of R_P's root (line 2). Given an entry e_T from R_T, the min-heap uses the lower bound cost $LBC(e_T, e_T.JL)$ as the priority, i.e., the entry e_T with the smallest LBC value is always processed first, as it is expected to contain uncompetitive products that can be upgraded with the lowest upgrading costs.

Algorithm 4 join(R-tree R_P of competitor set P, R-tree R_T of product set T, product cost function f_p)

```
1: initialize a min-heap H
2: enheap(H, ⟨{R_P.root}, R_T.root, null, ∞⟩)
3: while H is not empty do
4:     ⟨JL, e_T, t', cost⟩ ← deheap(H)
5:     if e_T is a leaf entry then
6:         if JL = ∅ then
7:             return cost, e_T's point, t'
8:         else
9:             get skyline S of all points indexed by entries in JL
10:            (cost, t') ← upgrade(S, e_T's point, f_p)
11:            enheap(H, ⟨∅, e_T, t', cost⟩)
12:    else
13:        if cost = 0 then
14:            for each entry e'_T in e_T do
15:                e'_T.JL ← ∅
16:                for each e ∈ JL do
17:                    if e.MBR ∩ ADR(e'_T.max) ≠ ∅ then
18:                        add e to e'_T.JL
19:                cost ← LBC(e'_T, e'_T.JL)
20:                enheap(H, ⟨e'_T.JL, e'_T, null, cost⟩)
21:        else
22:            remove the non-leaf entry e from JL according to
    Heuristic 2
23:            for each e's each subnode entry e' do
24:                if e'.MBR ∩ ADR(e_T.max) ≠ ∅ then
25:                    flag ← FALSE
26:                    for each e_P ∈ JL do
27:                        if e'.min is dominated by e_P.max then
28:                            flag ← TRUE; break
29:                        else if e_P.min is dominated by e'.max
    then
30:                            remove e_P from JL
31:                    if flag = FALSE then add e' to JL
32:            enheap(H, ⟨JL, e_T, null, LBC(e_T, JL)⟩)
```

The algorithm keeps processing the top node entry e_T from the heap and pushing relevant subentries onto the heap if necessary, until the top product is found (line 7). When processing the current entry e_T, the algorithm either expands e_T or picks an entry from $e_T.JL$. This decision is guided by the following heuristics.

Heuristic 1: If $LBC(e_T, e_T.JL) = 0$, we expand e_T.

Heuristic 2: If $LBC(e_T, e_T.JL) > 0$, we pick one entry e from $e_T.JL$ to expand.

Heuristic 2 is further explained by Heuristics 3 and 4.

Heuristic 3: If $LBC(e_T, e_T.JL) > 0$ and the naive lower bound LBC_N (Equation 2) or the conservative lower bound LBC_C (Equation 3) are used, we pick a non-leaf entry e with the minimum $LBC(e_T, e) > 0$.

Heuristic 4: If $LBC(e_T, e_T.JL) > 0$ and the aggressive lower bound LBC_A (Equation 4) is used, we pick a non-leaf entry e such that $LBC_A(e_T, e_T.JL) = LBC(e_T, e)$.

If the current entry e_T's LBC cost is zero, the algorithm expands e_T (lines 13–20). For each subentry e'_T of e_T, we find a relevant portion of $e_T.JL$ as e'_T's join list, i.e., those that overlap with $ADR(e'_T.max)$ and thus contain dominators of products in e'_T (lines 15–18). The lower bound upgrading cost is estimated for e'_T, which is then pushed onto the min-heap (lines 19–20).

If the current entry e_T's LBC cost is larger than zero, the algorithm expands a non-leaf node entry e from $e_T.JL$ (lines 22–32). Each e's subentry e' is considered only if e' contains dominators of products in e_T (lines 23–24). Further, a mutual dominance check is conducted between e' and $e_T.JL$ to remove those dominated entries (lines 25–30). If e' survives the dominance check, it is added to $e_T.JL$ (line 31). After all e's subentries are processed, e_T's LBC cost is calculated, and it is pushed back onto the min-heap (line 32).

The algorithm stops when the current e_T is a leaf entry (corresponding to a product), and its join list is empty (lines 5–7). This indicates that e_T is the uncompetitive product that is the cheapest to upgrade. For a leaf entry e_T with non-empty join list, the algorithm calculates its upgrading cost by calling Algorithm 1, sets its join list to empty, and pushes it back to the min-heap (lines 9–11).

As a remark, it is easy to change the algorithm slightly such that it returns top-k products to upgrade.

Moreover, the probing and join approaches basically yield the same upgrading results when there are no ties, i.e., they pick the same set of products to upgrade. This can be seen from the common use of algorithm **upgrade** (Algorithm 1) and the properties of the lower bound costs employed in the join approach. The two approaches may differ in choosing which product to include in the results when ties exist. This kind of uncertainty is seen in almost all top-k ranking problems.

IV. Empirical Study

This section reports on an empirical study of the basic probing algorithm (Algorithm 2, its improved version, and the join algorithm (Algorithm 4). It also studies the effectiveness of the three lower bound costs detailed in Section III-B3.

A. Overall Settings

All algorithms are implemented in Java and run on a Windows 7 enterprise desktop PC with a 2.93GHz CPU and 12GB RAM. As the performance metric, we measure the execution time of each algorithm. We investigate the effects of the cardinalities of the competitor and product sets P and T, the product dimensionality, and the number k of products to be returned. We use $k = 1$ as the default value in all experiments unless it is stated otherwise.

We use three data sets in the study. Section IV-B presents the results obtained on a real data set. Sections IV-C and IV-D present the results obtained on two synthetic data sets with different settings. Both data sets P and T are loaded into main memory before a top-k product upgrading algorithm is executed. The data loading time is excluded from the execution time measurements.

For the synthetic data sets, we use both anti-correlated and independent distributions according to the existing methods [3]. Each competitor set P is generated in the domain $[0, 1]^c$, and each product set T is generated in the domain $(1, 2]^c$, where c is the dimensionality of the data sets. We set

$|P|$ to be substantially larger than $|T|$ in most tests because we expect this to be occur frequently in practice. A manufacturer needs to compete in the market with all other manufacturers, who together have a large set P of competitor products.

The summation integration function (Equation 1) is used in the experiments. The attribute cost function used on each dimension is $f_a^i(p.d_i) = 1/(p.d_i + \epsilon)$ as smaller values are preferred in dominance.

B. Results on Real Data Set

We use the wine quality data set [6] from the UC Irvine Machine Learning Repository [1]. For use in the study, we adapt the white wine data set that contains 4,898 instances (tuples) as follows. From the original 12 attributes, we select three indicative attributes: chlorides, sulphates, and total sulfur dioxide. We chose these attributes because they are indicative of wine quality, as well as changeable to some degree by wine manufacturers during production. We then form the four attribute combinations shown in Table III that each have at least two attributes.

TABLE III
SELECTED WINE DATA SET ATTRIBUTES

Abbreviation	Wine Attributes
c, s	chlorides, sulphates
c, t	chlorides, total sulfur dioxide
s, t	sulphates, total sulfur dioxide
c, s, t	chlorides, sulphates, total sulfur dioxide

We then extract the data corresponding to each of these attribute combinations, obtaining four reduced data sets. In each reduced data set, we pick $1,000$ non-skyline tuples at random as the product data set T and let the remaining tuples be the competitor data set P. In other words, we have $|P| = 3,898$ and $|T| = 1,000$ in each combination. All data sets are normalized into the unit space $[0, 1]^c$.

The results of applying each algorithm to each of the four data sets are reported in Figure 4. As expected, the basic probing algorithm incurs the longest execution time because it is based on a brute-force approach. The improved probing algorithm cuts the execution time by approximately one third to one half. However, it is still significantly slower than the join algorithm. The three lower bounds yield only relatively modest performance differences. This is attributed to the small size of the real data sets, which offers limited room for optimizations to be effective.

We also evaluate the progressiveness of the join algorithm that is capable of returning the k results progressively during execution. We vary k, the number of products to be returned, from 1 to 20. We measure the execution time from the moment the join algorithm starts to the moment a particular number (1, 5, 10, 15, or 20) of products are available. The execution time results are reported in Figure 5. All lower bounds perform steadily as the k value increases. The conservative lower bound (CLB, Equation 3) performs the best because it avoids the blindness of the naive lower bound (NLB, Equation 2) and also avoids the complexity that the aggressive lower bound (ALB, Equation 4) incurs when estimating the upgrading costs. Note that the probing algorithms are not progressive: They cannot report top-k results incrementally before all products in T are processed.

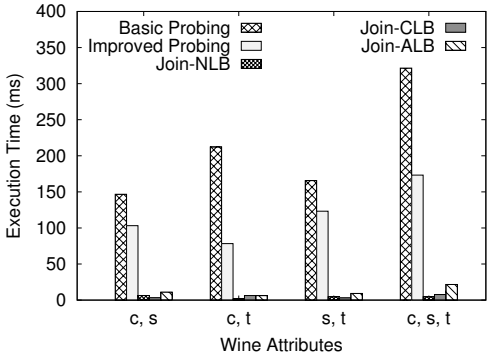


Fig. 4. Execution Time Costs on Different Wine Attribute Combinations

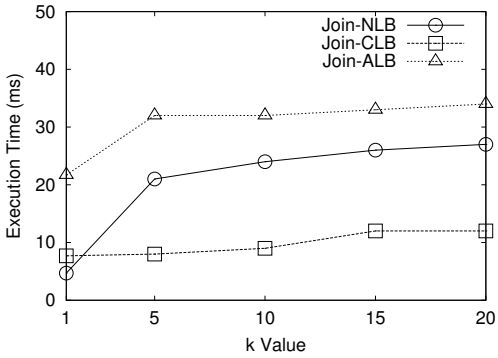


Fig. 5. Effect of k Value on Wine Data Set with c, s, t Attributes

We proceed to present the empirical results obtained with the two synthetic data sets. In the sequel, we omit the results of the basic probing algorithm because it executes significantly slower than the others.

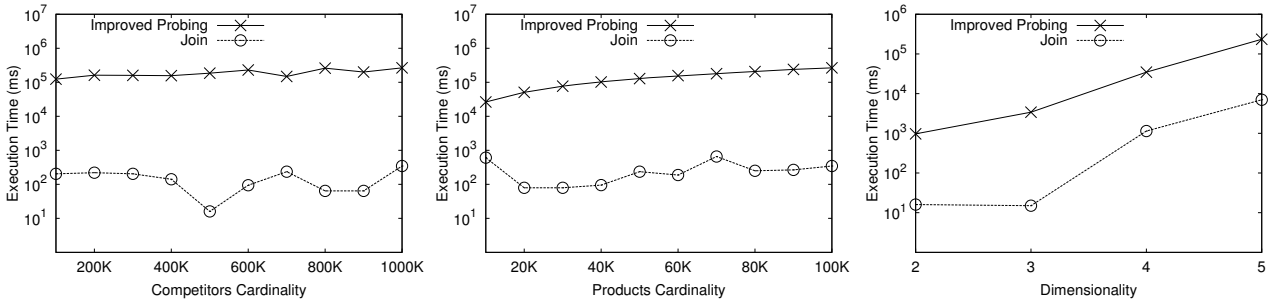
C. Results on Small Synthetic Data Sets

In the ensuring studies of the improved probing algorithm and the join algorithm, we use the data sets with the settings described in Table IV, where the default values are shown in bold. With these settings, the differences obtained for the join algorithm when using the three lower bounds are relatively small when compared with the differences from the improved probing algorithm. We thus use simply the NLB lower bound for the join algorithm.

TABLE IV
PARAMETER SETTINGS—SMALL SYNTHETIC DATA SETS

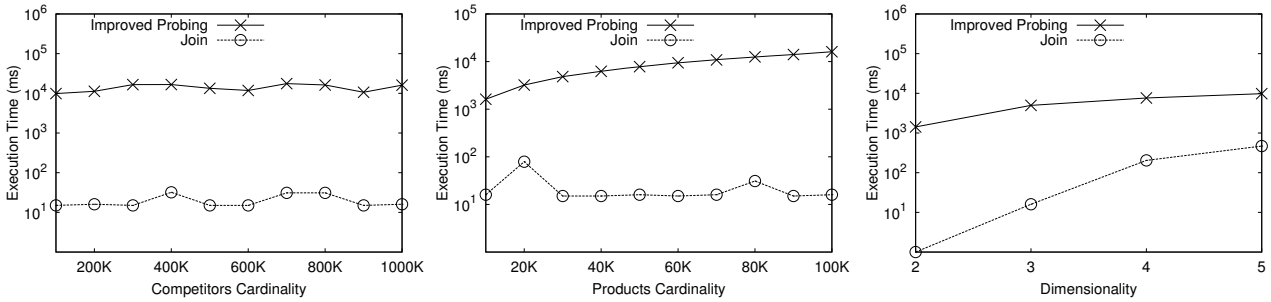
Parameter	Settings
Competitor Cardinality $\|P\|$	100K, 200K, . . ., **1000K**
Product Cardinality $\|T\|$	10K, 20K, . . ., **100K**
Dimensionality d	**2**, 3, 4, 5

We first fix the product set cardinality $|T|$ at 100K and the dimensionality d at 2, and we then vary the competitor



(a) Effect of Competitor Cardinality (b) Effect of Product Cardinality (c) Effect of Dimensionality

Fig. 6. Execution Time—Small Data Sets with Anti-Correlated Dimensions



(a) Effect of Competitor Cardinality (b) Effect of Product Cardinality (c) Effect of Dimensionality

Fig. 7. Execution Time—Small Data Sets with Independent Dimensions

set cardinality $|P|$ from 100K to 1,000K. The results are reported in Figure 6(a) and Figure 7(a) for anti-correlated and independent distributions, respectively. The results show that the join approach outperforms the improved probing approach by three orders of magnitude on both distributions. This indicates that the join approach is very efficient.

Comparing the results for the two distributions, both approaches incur longer execution times on anti-correlated dimensions. This is expected, as an anti-correlated data set usually has a large number of skyline points. Thus, the upgrading a product in anti-correlated dimensions likely needs to contend with relatively more dominating points, which yields higher computation costs.

Next, we fix the competitor set cardinality $|P|$ at 1,000K and the dimensionality d at 2, and we then vary the product cardinality $|T|$ from 10K to 100K. The results are shown in Figure 6(b) and Figure 7(b) for anti-correlated and independent distributions, respectively. Again, the join approach performs considerably better, by up to two orders of magnitude on anti-correlated data, than the improved probing approach. A more marked difference is seen on independent dimensions where the join approach executes faster by up to three orders of magnitude.

In addition, the results in Figure 6(b) and Figure 7(b) show that the improved probing degrades substantially (by approximately one order of magnitude) as the product cardinality increases. In contrast, the join-approach performance only fluctuates modestly under almost all settings. Thus, the study suggests that the join approach is scalable with respect to the product cardinality.

Finally, we investigate the effect of varying the dimensionality. We fix the competitor cardinality $|P|$ at 1,000K, the product cardinality $|T|$ at 100K, and we then vary the dimensionality d from 2 to 5. Figures 6(c) and Figure 7(c) show the results for the two distributions. As the dimensionality increases, more points are expected to be skyline points and to dominate existing products. This explains the execution time cost increase for both approaches. The join approach consistently outperforms the improved probing approach by one to three orders of magnitude.

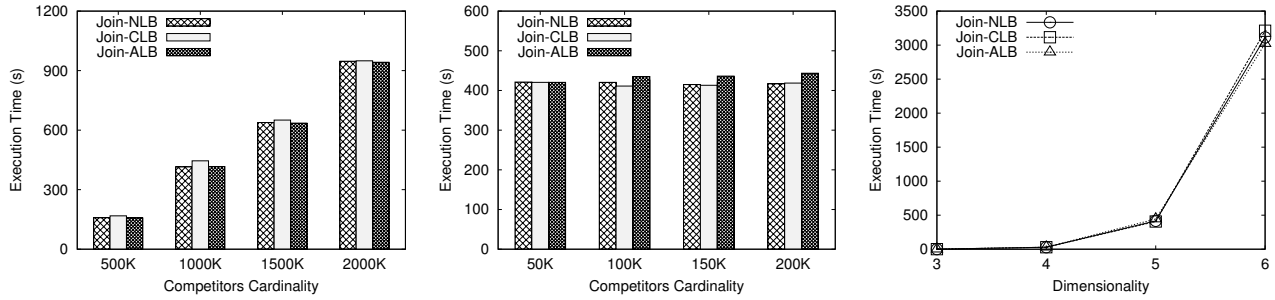
To summarize, the join approach executes much faster than the improved probing approach under all settings considered. In the sequel, we omit the improved probing approach and focus on evaluating the different ways of estimating the lower bound upgrading costs that are employed in the join approach.

D. Results on Large Synthetic Data Sets

We compare the three lower bound costs on large synthetic data sets. The relevant settings are presented in Table V, with default settings shown in bold. Note that the default dimensionality d is 5.

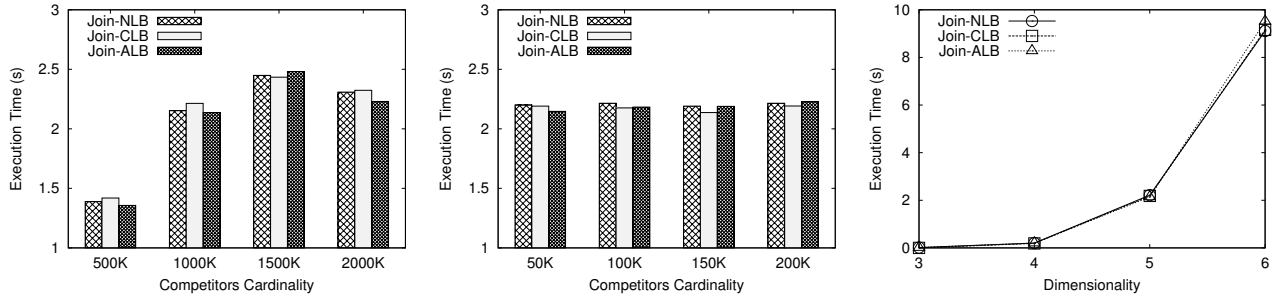
We first study the effect of varying the competitor set cardinality $|P|$ on the join algorithm when using different lower bound. We fix product cardinality $|T|$ at 100K and the dimensionality d at 5, and we then vary $|P|$ from 500K to 2,000K. The results are reported in Figure 8(a) and Figure 9(a).

With anti-correlated dimensions (Figure 8(a)) the execution time increases approximately linearly with increasing $|P|$



(a) Effect of Competitor Cardinality (b) Effect of Product Cardinality (c) Effect of Dimensionality

Fig. 8. Execution Time Costs—Large Data Sets with Anti-Correlated Dimensions



(a) Effect of Competitor Cardinality (b) Effect of Product Cardinality (c) Effect of Dimensionality

Fig. 9. Execution Time Costs — Large Data Sets with Independent Dimensions

TABLE V
PARAMETER SETTINGS—LARGE SYNTHETIC DATA SETS

Parameter	Settings
Competitor Cardinality $\|P\|$	500K, **1000K**, 1500K, 2000K
Product Cardinality $\|T\|$	50K, **100K**, 150K, 200K
Dimensionality d	3, 4, **5**, 6

regardless of which bound is used. Thus suggests scalability in $|P|$. The aggressive lower bound (ALB) outperforms its competitors slightly. This indicates that ALB is more efficient in estimating lower bound upgrading costs in the presence of higher numbers of dominating points.

ALB exhibits more variability for independent dimensions, as seen in the results in Figure 9(a). However, ALB also tends to incur shorter execution times than do NLB and CLB on independent dimensions. Fewer dominating points make the further optimization of ALB less effective compared to the cases on anti-correlated dimensions.

Next, we study the effect of the product set cardinality $|T|$. We fix the competitor set cardinality $|P|$ at 1,000K and the dimensionality d at 5, and we then vary $|T|$ from 50K to 200K. The results are presented in Figure 8(b) and Figure 9(b). For both distributions, all lower bounds are insensitive to the variation of the product set cardinality $|T|$. This indicates that our join approach is good at identifying, from a large set of products, the most economic products to upgrade.

Further, we study the effect of varying the dimensionality d from 3 to 6. The default cardinalities are used, i.e., $|P| = 1000$K and $|T| = 100$K. The results in Figure 8(c) and Figure 9(c) show that as the dimensionality increases, all lower bounds incur similar, longer execution times. A marked increase occurs when d goes from 5 to 6. A higher dimensionality leads to more skyline points as well as dominating points. Therefore, more computation is needed.

Figure 8(c) indicates that the aggressive lower bound (ALB) is slightly better than its alternatives on anti-correlated dimensions. This again indicates that ALB is more suitable in dealing with high number of dominating points when upgrading uncompetitive products. This advantage is not seen on independent dimensions, according to the results reported in Figure 9(c) where the three lower bounds perform almost the same.

We also evaluate the effect of k on the progressiveness of the join algorithm using the different lower bounds. We use the default settings and vary k from 1 to 20 to return different numbers of products that can be upgraded most economically. In each experiment, we measure the execution time from the moment the join algorithm starts to the moment a particular number of products is available.

The results on anti-correlated dimensions are reported in Figure 10. The naive lower bound (NLB) incurs considerably long execution times before it can return more than five products. The cost of NLB is significantly higher than those of its alternatives. This indicates that NLB is ineffective in estimating upgrading costs, especially when more uncompetitive products are expected. On the other hand, the conservative and aggressive lower bounds (CLB and ALB) exhibit a gradual

and more modest increase in execution time. Both are effective at making tight upgrading cost estimates and give priority to uncompetitive products with lower upgrading costs in the join.

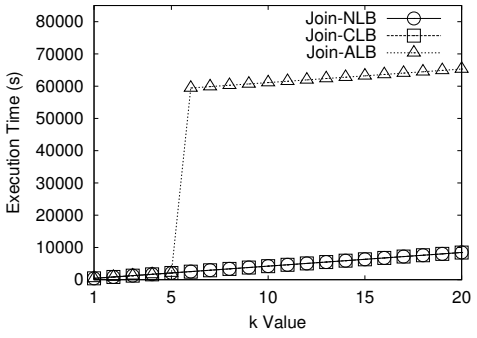


Fig. 10. Effect of Varying k on Anti-Correlated Distribution

The progressiveness results on independent dimensions are reported in Figure 11. Here, the three lower bounds differ only very slightly, which we attribute to the fact that independent dimensions result in fewer skyline points and dominating points, which offers less room for optimizations using lower bounds.

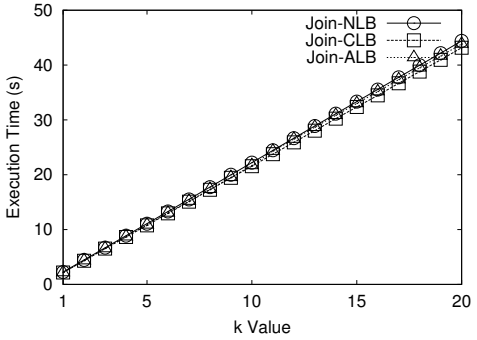


Fig. 11. Effect of Varying k on Independent Distribution

To summarize, the naive lower bound (NLB) is least desirable in terms of execution time and progressiveness. The aggressive lower bound (ALB) works well on anti-correlated dimensions in terms of both aspects, and the conservative lower bound (CLB) performs similarly to ALB, with some slight advantages in a few cases.

We also conducted the progressiveness test on small synthetic data sets, but found no drastic differences in the results. We thus omit the results due to the space limitation.

V. Related Work

A. Skyline Queries

Borzonyi et al. [3] introduce the skyline query to the database area. Given a multi-dimensional point set P, a skyline query returns all points from P that are not dominated by any other points in P. Subsequent skyline query processing algorithms fall into two categories: those that do not require indexes on the data set (e.g., [2], [5], [7], [18]) and those that require specific indexes (e.g., [8], [9], [12]–[14]).

Our top-k product upgrading problem adopts the dominance relationship that underlies skyline queries. However, rather than computing skylines, we consider the cost-effective modification of point attribute values to make them skyline points.

B. Dominance-Based Data Analysis

Li et al. [10] propose Dominant Relationship Analysis (DRA) for products and potential customers. Both products and customers are represented as multi-dimensional points; domination is applied to a product and a customer. For example, a product dominates a customer if the product satisfies (is no worse than) the customer's requirement. The authors propose a data cube-based structure, named *DADA*, to store the dominant relationships between products and consumers in a way that facilitates query processing. In particular, three types of queries are considered. First, given a space D, a plane L in D, and a set of C customers in D, a *Linear Optimization Query* $LOQ(L, C, D)$ finds a product p on L such that the number of customers dominated by p is maximized. Second, given a product p and a set of C customers in D, a *Subspace Analysis Query* (SAQ) finds a subspace $D' \subseteq D$ such that the difference between the number of customers dominated by p and that of customers dominating p is higher than a threshold. Third, given two sets A and B of products and a set C of customers in D, a *Comparative Dominant Query* (CDQ) computes the number of customers that are dominated by products in A and (or but not) by products in B.

Li et al. [11] combine the dominance relationship with spatial distance and define several location selection problems. Given a set H of spatial objects with both spatial locations and multiple comparable attributes, the authors define three types of queries. First, given an object q in H, a *Nearest Dominator Query* (*NDQ*) returns q's nearest dominator $ND(q)$ in H, i.e., $ND(q)$ dominates q on all non-spatial attributes and $ND(q)$ is the nearest to q among all q's dominators. Such a nearest distance is called $ndd(q)$. Second, given a hyperplane P in the non-spatial attribute space of H, a *Least Dominated, Profitable Points Query* (*LDPQ*) finds object(s) t from H such that t is dominated by some object(s) on P and $ndd(t)$ in H is maximized. Third, given a threshold δ and a hyperplane P as above, a *Minimal Loss and Least Dominated Points Query* (*ML2DQ*) finds object(s) t from H such that $ndd(t) \geq \delta$ and the distance to P on all non-spatial attributes is minimized.

This paper's proposals differ from the above two works as follows. First, our focus is on upgrading uncompetitive products rather than on finding existing ones that satisfy given conditions. Second, while the above works [10], [11] use linear planes as product constraints, we adopt more general constraints. Third, the non-spatial distance used by Li et al. [11] does not take into account the different implications of different attributes and the uniqueness of cost calculation on a particular attribute, which we do.

Zhang et al. [19] view manufacturers as game players and apply game theory to analyze the competitions among manufacturers who want to attract as many customers as possible by manufacturing products that dominate customer

expectations. Assuming a profit constraint hyperplane, the authors prove the existence of a Nash Equilibrium between manufacturers such that one cannot attract more customers by changing the quality of its own products. Algorithms for designing the optimal product are proposed for a manufacturer in the presence of competition from others.

In contrast, we do not model competitions among multiple manufacturers or measure the number of dominated customers. Such information is very hard to obtain and maintain precisely in an ever-changing market. Moreover, our work also assumes more general constraints than linear hyperplanes.

Wan et al. [16] address the problem of creating competitive products from existing "half-products." Specifically, all half-products are stored in k *source tables* $T_1, T_2, \ldots, T_k$, each having a set X_i of attributes; all existing products are stored in a table T_E. Given k tuples $t_1, t_2, \ldots, t_k$ where $t_i \in T_i$, a product function θ generates a new product $q = \theta(t_1, t_2, \ldots, t_k)$. All such new products form another product table T_Q. The problem of creating competitive products is to find all new products in T_Q that are not dominated by any product in $T_E \cup T_Q$, i.e., the new products in the skyline of $T_E \cup T_Q$.

Our proposals differ from this work in two important ways. First, we do not assume multiple half-product tables as sources for new products, but rather work on a single complete product set. Second, we take into account the manufacturing cost of products, which is not considered by Wang et al.

In recent work, Wan et al. [17] study the problem of finding top-k profitable products in a setting that is quite different from the one we assume. Wang et al. consider a set P of existing products and a set Q of potential new products. All products have multiple attributes including price. Each product in P has a known price, while the prices in Q are undecided. The top-k profitable products problem finds a k-subset Q' of Q and decides the price for each product in Q, such that Q' is contained in the skyline of $P \cup Q$ in terms of all attributes and so that all products in Q' produce maximized total profit with respect to their known, fixed costs.

The top-k profit products problem clearly differs from our top-k product upgrading problem. Our problem is concerned with the changing of a products' attribute values, not with deciding their prices, to make them non-dominated and thus attractive at low cost. We focus on quality attributes rather than on prices and profits. Furthermore, we consider variable costs that are expressed by cost functions.

VI. Conclusion and Research Directions

We formulate and study the top-k product upgrading problem: Given a set T of uncompetitive products and a set P of competitors, the problem is to find the k products in T that can be upgraded with the lowest costs such that no products in P dominate them.

We propose two types of algorithms for this problem. The probing algorithm process each product in T in isolation and does not require an index on T. The join algorithm requires both P and T to be indexed by an R-tree and processes products in T jointly. Three lower bounds on the upgrading cost for a group of products are developed and employed by the join algorithm to prune products with too high upgrading costs. All proposals are evaluated empirically using real and synthetic data. The results suggest that the join algorithm is efficient and scalable. The results also elicit the performance differences among the three lower bounds.

Several research directions exist. We assume that all dimensions are on numerical domains. It is relevant to explore how the definitions and proposed techniques can be extended to data with a mix of both numerical and non-numerical domains, e.g., categorical domains.

The cost functions we use are assumed to be monotonic. However, cost functions that are not strictly monotonic may be of interest. This renders it relevant to extend our techniques to contend with more general cost functions.

We keep the uncompetitive products and their competitors in two separate sets. However, they can also be stored in a single set, e.g., if a manufacturer alone owns a large number of products and wants to upgrade some of its uncompetitive products in the presence of advantaged ones. It is of interest to develop efficient algorithms for this setting.

Finally, while we prove that Algorithm 1 is correct, further studies of the optimality of the algorithm, in terms of the upgrade cost of the result, are in order.

References

[1] UC Irvine Machine Learning Repository. http://archive.ics.uci.edu/ml/.

[2] I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. *ACM TODS*, 33(4), 2008.

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

[4] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *Proc. SIGMOD*, pp. 237–246, 1993.

[5] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *Proc. ICDE*, pp. 717–719, 2003.

[6] P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis. Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, 47(4):547–553, 2009.

[7] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *Proc. VLDB*, pp. 229–240, 2005.

[8] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proc. VLDB*, pp. 275–286, 2002.

[9] K. C. K. Lee, B. Zheng, H. Li, and W.-C. Lee. Approaching the skyline in z order. In *Proc. VLDB*, pp. 279–290, 2007.

[10] C. Li, B. C. Ooi, A. K. H. Tung, and S. Wang. Dada: a data cube for dominant relationship analysis. In *Proc. SIGMOD*, pp. 659–670, 2006.

[11] C. Li, A. K. H. Tung, W. Jin, and M. Ester. On dominating your neighborhood profitably. In *Proc. VLDB*, pp. 818–829, 2007.

[12] B. Liu and C.-Y. Chan. Zinc: Efficient indexing for skyline computation. *PVLDB*, 4(3):197–207, 2010.

[13] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *Proc. SIGMOD*, pp. 467–478, 2003.

[14] K. L. Tan, P. K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *Proc. VLDB*, pp. 301–310, 2001.

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

[16] Q. Wan, R. C.-W. Wong, I. F. Ilyas, M. T. Özsu, and Y. Peng. Creating competitive products. *PVLDB*, 2(1):898–909, 2009.

[17] Q. Wan, R. C.-W. Wong, and Y. Peng. Finding top-k profitable products. In *Proc. ICDE*, pp. 1055–1066, 2011.

[18] S. Zhang, N. Mamoulis, and D. W. Cheung. Scalable skyline computation using object-based space partitioning. In *Proc. SIGMOD*, pp. 483–494, 2009.

[19] Z. Zhang, L. V. S. Lakshmanan, and A. K. H. Tung. On domination game analysis for microeconomic data mining. *TKDD*, 2(4), 2009.