Creating Competitive Products[*]

Qian Wan[1], Raymond Chi-Wing Wong[1], Ihab F. Ilyas[2], M. Tamer Özsu[2], Yu Peng[1]

[1]Hong Kong University of Science and Technology [2]University of Waterloo
{qwan,raywong,gracepy}@cse.ust.hk {ilyas,tozsu}@uwaterloo.ca

ABSTRACT

The importance of dominance and skyline analysis has been well recognized in multi-criteria decision making applications. Most previous works study how to help customers find a set of "best" possible products from a pool of given products. In this paper, we identify an interesting problem, creating competitive products, which has not been studied before. Given a set of products in the existing market, we want to study how to create a set of "best" possible products such that the newly created products are not dominated by the products in the existing market. We refer such products as competitive products. A straightforward solution is to generate a set of all possible products and check for dominance relationships. However, the whole set is quite large. In this paper, we propose a solution to generate a subset of this set effectively. An extensive performance study using both synthetic and real datasets is reported to verify its effectiveness and efficiency.

1. INTRODUCTION

Dominance analysis is important in many multi-criteria decision making applications.

EXAMPLE 1 (SKYLINE). Consider that a customer is looking for a vacation package, where each package typically contains a flight reservation and a hotel reservation, using some travel agencies like Expedia.com and Priceline.com. The customer uses four criteria for choosing a package, namely No-of-stops, Distance-to-beach, Hotel-class and Price. For two packages p and q, if p is better than q in at least one factor, and is not worse than q in the rest of remaining factors, then p is said to *dominate* q. Table 1 shows four packages: p_1, p_2, p_3 and p_4. In attribute Hotel-class, the numbers in braces can be ignored at this point and will be described later. For example, in the table, p_1 has attribute "Hotel-class" equal to 4. Assume that less stops, shorter distance to beach, higher hotel class and lower price are more preferable. Thus, p_1 dominates p_4 because p_1 has less stops, shorter distance to beach, higher hotel class and lower price. However, package p_1 does not dominate p_2 because p_2 has lower price. Similarly, package p_2 does not dominate p_1 because p_1 has less stops. □

In a table, a tuple that is not dominated by any other tuple is said to be a *skyline tuple* or it is in the *skyline*. Recently, skyline analysis [17, 12, 20, 10, 15, 23] has received a lot of interest in the literature. In Example 1, package p is in the skyline if it is not dominated by any other packages. The packages in the skyline are the best possible tradeoffs among the four factors in question. For example, p_1 is in the skyline because it is not dominated by p_2, p_3 and p_4. However, p_4 is not in the skyline because p_4 is dominated by p_1.

EXAMPLE 2 (CREATING COMPETITIVE PRODUCTS). A new travel agency wants to start or create some new packages to be formed from a pool of flights and a pool of hotels as shown in Table 2 and Table 3, respectively. One straightforward way of forming new packages is to generate all possible combinations of flights and hotels.

When generating a new package from a flight f and a hotel h, we set the price of the new package as a function of the cost of f and the cost of h. For example, we set the price of package q exactly to the sum of the cost of f and the cost of h. Here, a package q generated from f_3 and h_5 has price at least $80 + 140 = 220$. Thus, all attributes of q (No-of-stops, Distance-to-beach, Hotel-class, Price) are (2, 170, 4, 220). □

In Example 2, the set of all possible packages generated from flights and hotels as shown in Table 4 is $\{q_1 : (f_1, h_1), q_2 : (f_1, h_2), q_3 : (f_1, h_3), q_4 : (f_1, h_4), ..., q_{24} : (f_4, h_6)\}$.

Note that there are some *existing* packages in the market as shown in Table 1. Not all newly generated packages from flights and hotels will be chosen by customers because some of them are dominated by existing packages in the market. For example, $q_{24} : (f_4, h_6)$ has (No-of-stops, Distance-to-beach, Hotel-class, Price) = (2, 200, 3, 210). It is dominated by package p_2 in the existing market because the price of p_2 is lower than the price of q_{24} and other attributes of p_2 are not worse than those of q_{24}.

In addition to the existing packages in the market, some newly generated packages may also be dominated by other *newly* generated packages. For example, a newly generated package $q_{24} : (f_4, h_6)$ is dominated by another newly generated package $q_{13} : (f_3, h_1)$ where q_{13} has (No-of-stops, Distance-to-beach, Hotel-class, Price) = (2, 100, 3, 180).

[*]This research was supported by an RGC Earmarked Research Grant of HKSAR HKUST and by the Natural Sciences and Engineering Research Council (NSERC) of Canada. All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

Package	No-of-stops	Distance-to-beach	Hotel-class	Price
p_1	0	130	4 (2)	250
p_2	1	140	4 (2)	170
p_3	1	300	5 (1)	150
p_4	1	150	2 (4)	300

Table 1: Packages in the existing market

Flight	No-of-stops	Flight-cost
f_1	0	120
f_2	1	100
f_3	2	80
f_4	2	90

Table 2: A set F of flights from the new travel agency

Hotel	Distance-to-beach	Hotel-class	Hotel-cost
h_1	100	3 (3)	100
h_2	200	4 (2)	90
h_3	400	5 (1)	80
h_4	150	4 (2)	150
h_5	170	4 (2)	140
h_6	200	3 (3)	120

Table 3: A set H of hotels from the new travel agency

Package	No-of-stops	Distance-to-Beach	Hotel-class	Price
$q_1 : (f_1, h_1)$	0	100	3 (3)	220
$q_2 : (f_1, h_2)$	0	200	4 (2)	210
$q_3 : (f_1, h_3)$	0	400	5 (1)	200
...	...	...	...	...
$q_7 : (f_2, h_1)$	1	100	3 (3)	200
...	...	...	...	...
$q_{13} : (f_3, h_1)$	2	100	3 (3)	180
...	...	...	...	...
$q_{24} : (f_4, h_6)$	2	200	3 (3)	210

Table 4: All possible packages generated from F and H

The set of all possible newly generated packages that are not dominated by any packages in the existing market and any newly created packages corresponds to the "best" packages formed from flights and hotels. We call these packages *competitive packages*.

Hence, the problem in Example 2 is: Given a table T_E storing all packages in the existing market, a table storing flights and a table storing hotels, we want to find all competitive packages generated from the flights and the hotels. Specifically, they are in the skyline with respect to the final dataset that include packages in T_E and all possible packages formed from hotels and flights. In Table 4, only q_1, q_2, q_3, q_7 and q_{13} are competitive packages.

A naive way to obtain the set of competitive packages is to (1) generate all possible combinations of hotels and flights, (2) add these to the existing market packages and (3) compute the skyline of the whole dataset. This approach has several weaknesses. Firstly, the set of all possible combinations generated from flights and hotels can be extremely large. This motivates us to propose an algorithm which considers only a subset of the space of possible combinations and thus effectively reduces the search space while computing the full set of competitive packages. Secondly, since a newly generated product possibly dominates another newly generated product, there is a need to check the dominance relationship among each pair of newly generated packages, which can be prohibitively expensive.

In this paper, we formulate this problem and introduce efficient algorithms that avoid fully materializing the space of all possible packages and naively applying the skyline algorithm on the whole space. We call this problem *creating competitive products* where a package in our example refers to a product.

Forming competitive products is common in real life applications. Other applications for creating competitive products include assembling new laptops which involve CPU, memory and screen where laptops correspond to products and CPU, memory and screen are used to form products; a laptop company can order the components from different vendors and there are a lot of existing laptops in the market. Another interesting application is to create a delivery service which involves different transportation carriers such as flights and trucks. A cargo delivery company can use different transportation carriers for the delivery. In this application, delivery services are products which are generated from different transportation carriers.

Our contributions are summarized as follows. (1) To the best of our knowledge, we are the first to study how to create competitive products. Creating competitive products can help the effort of companies to generate new packages, which cannot be addressed by existing methods. (2) We also propose a solution which can reduce the size of the space of possible combinations effectively by grouping "similar" products in the same groups and processing them as a whole. (3) We present a systematic performance study using both real and synthetic datasets to verify the effectiveness and the efficiency of our method. The experimental results show that creating competitive products is interesting.

The rest of the paper is organized as follows. We first give a background and some notations of this problem in Section 2. In Section 3, we formally define our problem. Our proposed method is developed in Section 4. In Section 5, we give some discussions of the proposed method. A systematic performance study is reported in Section 6. In Section 7, we describe some related work. The paper is concluded in Section 8.

2. BACKGROUND AND NOTATIONS

We first describe the background about skyline in Section 2.1. Then, we give some notations used in this paper in Section 2.2.

2.1 Background: Skyline

A skyline analysis involves multiple attributes. The values in each attribute can be modeled by a partial order on the attribute. A *partial order* $\preceq$ is a reflexive, asymmetric and transitive relation. A partial order is also a total order if, for any two values u and v in the domain, either $u \preceq v$ or $v \preceq u$. We write $u \prec v$ if $u \preceq v$ and $u \neq v$.

By default, we consider tuples in an w-dimensional[1] space $\mathbb{S} = x_1 \times \cdots \times x_w$. For each dimension x_i, we assume that there is a partial or total order. For a tuple p, $p.x_i$ is the projection on dimension x_i. For dimension x_i, if $p.x_i \preceq q.x_i$, we also simply write $p \preceq_{x_i} q$. We can omit x_i if it is clear from the context.

For tuples p and q, p *dominates* q with respect to $\mathbb{S}$, denoted by $p \prec q$, if, for any dimension $x_i \in \mathbb{S}$, $p \preceq_{x_i} q$, and there exists a dimension $x_{i_0} \in \mathbb{S}$ such that $p \prec_{x_{i_0}} q$. If p dominates q, then p is more preferable than q.

DEFINITION 1 (SKYLINE). *Given a dataset $\mathcal{D}$ containing tuples in space $\mathbb{S}$, a tuple $p \in \mathcal{D}$ is in the skyline of $\mathcal{D}$ (i.e., a skyline*

[1]In this paper, we use the terms "*attribute*" and "*dimension*" interchangeably.

tuple in $\mathcal{D}$) if p is not dominated by any tuples in $\mathcal{D}$. The skyline of $\mathcal{D}$, denoted by $SKY(\mathcal{D})$, is the set of skyline tuples in $\mathcal{D}$.

For example, in Table 1 where $\mathcal{D} = \{p_1, p_2, p_3, p_4\}$, since p_1, p_2 and p_3 are not dominated by any tuples in $\mathcal{D}$, $SKY(\mathcal{D})$ is equal to $\{p_1, p_2, p_3\}$.

2.2 Notations

Given k *source tables*, namely $T_1, T_2, ..., T_k$, each source table T_i has a set X_i of attributes. The domain of each attribute in X_i is $\mathbb{R}$. For any two sets of attributes X_i and X_j, $X_i \cap X_j = \emptyset$. Let $\mathcal{X}$ denote the set of all attributes of source tables. That is, $\mathcal{X} = \cup_{i=1}^{k} X_i$.

The table T_1 storing the flights (Table 2) and the table T_2 storing the hotels (Table 3) are examples of source tables. X_1 and X_2 are {"No-of-stops", "Flight-cost"} and {"Distance-to-beach", "Hotel-class", "Hotel-cost"}, respectively. $\mathcal{X}$ = {"No-of-stops", "Flight-cost", "Distance-to-beach", "Hotel-class", "Hotel-cost"}. Let x_1, x_2, x_3, x_4 and x_5 be "No-of-stops", "Flight-cost", "Distance-to-beach", "Hotel-class" and "Hotel-cost", respectively.

The source tables are used to generate the *product table*. The product table T_P has a set Y of attributes. The domain of each attribute in Y is $\mathbb{R}$. Each attribute $y_j \in Y$ of the product table can be computed from the attribute set $\mathcal{X}$ of the source tables according to the following definition:

DEFINITION 2 (MERGING FUNCTION g_j). *For each attribute $y_j \in Y$, we define a function g_j called* merging function *over attribute set $\mathcal{X}$ such that $y_j = g_j(\mathcal{X})$.*

In this paper, for the sake of illustration, we study the merging function g_j with the linear form over $\mathcal{X}$ as follows.

$$g_j(\mathcal{X}) = \sum_{x \in \mathcal{X}} w(x) \cdot x \tag{1}$$

where $w(x)$ is the weight of attribute x and is a real number. The weights of all attributes in $\mathcal{X}$ for function g_j are denoted by a vector v_j. If the weight $w(x)$ of an attribute $x \in \mathcal{X}$ is equal to 0, we say that $y_j \in Y$ is *independent* of attribute x. Otherwise, we say that y_j is *dependent* on attribute x. The weight vector of each function g_j is given by the user. Note that the technique presented in this paper can handle any other specific monotonic merging function.

The above linear form (Equation 1) can express how the attribute y_j of the product table can be derived from the attributes of source tables in many real applications. We distinguish between two kinds of attributes in the product table, namely *direct attribute* and *indirect attribute*.

A direct attribute of the product table is an attribute which is exactly equal to one of the attributes of a source table. For example, in our running example, attribute "No-of-stops" of Table 4, says y_j, is exactly equal to attribute "No-of-stops" of Table 2, says x. In this case, the vector for the merging function of attribute y_j contains only one entry $w(x)$ equal to 1 and other entries $w(x')$ equal to 0. An indirect attribute of the product table is the attribute that is equal to the weighted sum of multiple attributes of multiple source tables. For instance, the product table has attribute "Price" (y_4) which is equal to the sum of attribute "Flight-cost" of Table 2 (x_2) and attribute "Hotel-cost" of Table 3 (x_5). In this case, the vector v_4 contains two entries, namely $w(x_2)$ and $w(x_5)$, both equal to 1, and other entries $w(x')$ equal to 0. The formulation of the summation of attributes appears naturally in many applications.

DEFINITION 3 (DEPENDENT ATTRIBUTE). *Given an attribute $y_j \in Y$ and an attribute $x \in X_i$ where $i = 1, 2, ..., k$, x is said to be a* dependent attribute *of y_j if the weight of x in function g_j (i.e., $w(x)$) is equal to a non-zero value. We define $D(y_j)$ to denote a set of all dependent attributes of y_j.*

EXAMPLE 3 (DEPENDENT ATTRIBUTE). Table 4 is the product table. Y is equal to {"No-of-stops", "Distance-to-Beach", "Hotel-class", "Price"}. Let y_1 ="No-of-stops", y_2 ="Distance-to-Beach", y_3 ="Hotel-class" and y_4 ="Price".

Suppose the vector stores the weights of attributes in $\mathcal{X}$ in this order: x_1 :"No-of-stops", x_2 :"Flight-cost", x_3 :"Distance-to-beach", x_4 :"Hotel-class", x_5 :"Hotel-cost". Then, the vectors v_1, v_2, v_3 and v_4 are equal to (1, 0, 0, 0, 0), (0, 0, 1, 0, 0), (0, 0, 0, 1, 0) and (0, 1, 0, 0, 1), respectively. It is easy to verify that $D(y_1), D(y_2), D(y_3)$ and $D(y_4)$ are equal to $\{x_1\}$, $\{x_3\}$, $\{x_4\}$ and $\{x_2, x_5\}$, respectively. $\square$

In the above example, we observe that there is no overlapping among $D(y)$'s. In other words, each attribute x of a source table is involved in the computation of *exactly one* attribute y of a product table. In the following, to simplify the discussion, we assume that, for any two attributes y and y' in Y, $D(y) \cap D(y') = \emptyset$. If this assumption does not hold where an attribute x of a source table is involved in the computation of more than one attribute of a product table, we can duplicate attribute x such that the above assumption holds. With this assumption, we have the following definition.

DEFINITION 4 (TARGET ATTRIBUTE). *Suppose $y_j \in Y$ and $x \in X$. y_j is said to be the* target attribute *of x, denoted by $\alpha(x)$, if the weight of x in vector v_j (i.e., $w(x)$) is non-zero.*

Since each attribute x of a source table is involved in the computation of *exactly one* attribute y of a product table, each attribute x has its unique target attribute.

EXAMPLE 4 (TARGET ATTRIBUTE). Since $D(y_1)$, $D(y_2)$, $D(y_3)$ and $D(y_4)$ are equal to $\{x_1\}$, $\{x_3\}$, $\{x_4\}$ and $\{x_2, x_5\}$, we obtain that $\alpha(x_1), \alpha(x_2), \alpha(x_3), \alpha(x_4)$ and $\alpha(x_5)$ are equal to y_1, y_4, y_2, y_3 and y_4, respectively. $\square$

In our motivating application, we say that, in table flight, attribute Flight-cost is a *merging attribute* because its value is *merged* with the value of attribute Hotel-cost to form the value of attribute Price for a package. We say that attribute No-of-stops is a *non-merging attribute* since attribute No-of-stops is directly used in the attribute value for a package.

DEFINITION 5 (MERGING ATTRIBUTE). *Given an attribute $x \in X_i$, x is said to be a* merging attribute *if $|D(y)| > 1$ where $y = \alpha(x)$.*

EXAMPLE 5 (MERGING ATTRIBUTE). In table flight (Table 2), attribute "Flight-cost" (x_2) is a merging attribute. Let y_4 be attribute "Price". This is because $\alpha(x_2) = y_4$ and $D(y_4) = \{x_2, x_5\}$ where $|D(y_4)| > 1$. But, attribute "No-of-stops" (x_1) is not a merging attribute. In table hotel (Table 3), attribute "Hotel-cost" (x_5) is a merging attribute. But, attributes "Distance-to-beach" (x_3) and "Hotel-class" (x_4) are not. $\square$

3. PROBLEM DEFINITION

Each tuple in the product table is a *product* generated from one tuple of each source table T_i. Consider a tuple p in the product table generated from tuple t_1 in T_1, tuple t_2 in T_2, ..., tuple t_k in T_k. We define a function called *product function* over these tuples with merging function g_j as follows.

DEFINITION 6 (PRODUCT FUNCTION θ). *Consider k tuples, namely $t_1, t_2, ..., t_k$, where t_i is a tuple in T_i for $i = 1, 2, ..., k$. Suppose we generate the product q from these k tuples. We define a function θ called* product function *over the k tuples, namely $t_1, t_2, ..., t_k$, such that $q = \theta(t_1, t_2, ..., t_k)$.*

Let $\mathcal{X}_v$ be the set of all attribute values of tuples $t_1, t_2, ..., t_k$. Specifically, under function θ, for each attribute y_j of q, the value of y_j is equal to $g_j(\mathcal{X}_v)$.

A package q generated from f_3 of T_1 (flights) and h_5 of T_2 (hotels) is computed by $\theta(f_3, h_5)$. In Example 2, it has attributes (No-of-stops, Distance-to-beach, Hotel-class, Price) = (2, 170, 4, 220).

Let $U(T_1, T_2, ..., T_k)$ be the set of all possible products generated from source tables $T_1, T_2, ..., T_k$. Formally, $U(T_1, T_2, ..., T_k)$ is equal to

$$\{\theta(t_1, t_2, ..., t_k) | t_i \in T_i \text{ where } i \in [1, k])\}$$

In our running example, $U(T_1, T_2)$ is equal to $\{q_1 : (f_1, h_1), q_2 : (f_1, h_2), q_3 : (f_1, h_3), q_4 : (f_1, h_4), ..., q_{24} : (f_4, h_6)\}$. $U(T_1, T_2, ..., T_k)$ is represented by a product table denoted by T_Q for the ease of reference. Table 4 is an example of T_Q.

In addition to the possible products generated from source tables $T_1, T_2, ..., T_k$, there exist products in the *existing markets*. These existing products are stored in a product table denoted by T_E. Thus, the products in T_E are given but the products in T_Q are to be generated from source tables $T_1, T_2, ..., T_k$. Table 1 is an example of T_E.

We define *competitive products* as follows.

DEFINITION 7 (COMPETITIVE PRODUCT). *Given a product q in T_Q, q is said to be a* competitive product *if q is in the skyline with respect to $T_E \cup T_Q$.*

In our motivating example, described in Section 1, the newly created product q_{24} is not a competitive product because q_{24} is dominated by product p_2 in the existing market. However, the newly created product q_1 is a competitive product because there are no other products in T_E and T_Q dominating q_1.

In this paper, we address the problem of finding all competitive products in T_Q.

A straightforward solution involves two steps. (1) *Step 1 (Creating T_Q):* The first step is to generate T_Q from k source tables, namely $T_1, T_2, ..., T_k$. (2) *Step 2 (Finding Competitive Product):* The second step is to adopt one of the existing algorithms [1, 4, 13] to compute the skyline with respect to $T_E \cup T_Q$.

However, the computation is expensive. Suppose that each table T_i has γ tuples. The size of T_Q is equal to γ^k. For example, when $k = 3$ and $\gamma = 1,000,000$, then the size of T_Q is equal to 1×10^{18}, which is extremely large. Most of the known algorithms without indexing finding the skyline over a single table T are shown to have a worst-case complexity of $O(d|T|^2)$, where d is the number of dimensions and $|T|$ is the table size, and an average-case complexity at least linear in $|T|$ [9]. It is shown in [5] that the skyline problem requires at least $\lceil \log |T|! \rceil$ comparisons. Thus, since the second step of the straightforward approach processes the data $T(= T_E \cup T_Q)$, if $|T_E|$ is equal to 1,000,000, the size of the table T denoted by $|T|$ is equal to $1,000,000 + 1 \times 10^{18} \approx 1 \times 10^{18}$. With this large value of $|T|$, the complexity of the straightforward approach is quite high (i.e., $O(d \times (1 \times 10^{18})^2)$). This motivates us to propose an algorithm which considers only a subset of this set and thus effectively reduces the search space.

4. ALGORITHM

In the following, for the sake of illustration, we assume that, for each attribute, the smaller the value is, the better it is. In our motivating example, only attribute Hotel-class does not follow this assumption. We subtract each value in attribute Hotel-class from 5. In Table 1 and Table 3, the numbers in braces are the subtracted value. In the following, we use the subtracted value for attribute Hotel-class.

Our objective is to find all competitive products from T_Q *efficiently*. Note that all products in the answer are in the skyline with respect to $T_E \cup T_Q$. The computation cost of finding these products depends on two major components.

- *Intra-dominance Checking:* Intra-dominance checking refers to the dominance checking among all newly generated products in T_Q. For example, in Section 1, we observe that some newly generated packages, says q_{24}, may dominate another newly generated packages, says q_{13}. There are totally at most $|T_Q|^2$ dominance checks.
- *Inter-dominance Checking:* Inter-dominance checking refers to the dominance checking between the tuples in T_Q and the tuples in T_E. For example, as described in Section 1, the newly generated package q_{24} is dominated by an existing package p_2. There are totally at most $|T_P| \times |T_Q|$ dominance checks.

The total number of checks is at most $|T_Q|^2 + |T_P| \times |T_Q|$. In the following, we propose some techniques which reduce both the number of intra-dominance checks and the number of inter-dominance checks. At the same time, we do not want to materialize the entire T_Q.

This project is started with a travel agency which wants to create packages from flights and hotels in order to create competitive packages. This project has one important characteristic called the *at-most-one merging attribute characteristic*: for each source table T_i, there exists at most one merging attribute in X_i. This characteristic *avoids any intra-doiminace checking* among tuples in T'_Q. In the following, we assume that the application satisfies the at-most-one merging attribute characteristic. A general model which may not satisfy this characteristic is described in Section 5.

The at-most-one merging attribute characteristic comes naturally in a lot of applications in addition to creating packages. All applications for generating products based on attribute price are some examples. For example, when new laptops are formed, we consider attribute price of each component (e.g., CPU, memory and screen). Another example is the delivery service where each transportation carrier has attribute price.

We first describe the framework of our algorithm in Section 4.1 by avoiding intra-dominance checking steps. Based on this framework, we propose to group "similar" newly generated products together to reduce the number of inter-dominance checking steps.

4.1 Framework

In this section, we give a framework which is simple but effective to generate competitive products by avoiding the intra-dominance checking.

EXAMPLE 6 (FRAMEWORK). Consider a package $q : (f_4, h_6)$ and $q' : (f_3, h_2)$. From Table 2 and Table 3, it easy to verify that q has $(y_1, y_2, y_3, y_4) = (2, 200, 3, 210)$ and q' has $(y_1, y_2, y_3, y_4) = (2, 200, 2, 170)$. Specifically, q is dominated by q'. We call this dominance relationship as an intra-dominance relationship.

The reason why q is dominated by q' is that each of the tuples from the source tables which are used to generate q are dominated

Flight	No-of-stops	Cost
f_1	0	120
f_2	1	100
f_3	2	80

Table 5: A set F' of skyline tuples in F (i.e., $SKY(F)$)

by each of the correspondence tuples which are used to generate q'. Specifically, f_4 is dominated by f_3 (See Table 2) and h_6 is also dominated by h_2 (See Table 3). By this observation, we propose a framework which first removes all tuples in each source table dominated by other tuples. The remaining tuples of a source table T_i correspond to the skyline of T_i, which will be used to generate competitive products. $\square$

The framework is described as follows. For each source table T_i, we find the skyline over T_i, denoted by T'_i, where $T'_i = SKY(T_i)$. Let T_1 be table flight (Table 2) and T_2 be table hotel (Table 3). It is easy to verify that in T_1, only f_4 is dominated and thus T_1 becomes T'_1 as shown in Table 5. Besides, in T_2, only h_6 is dominated and thus T_2 becomes T'_2 as shown in Table 6. Let T'_Q be the set of all products generated from $T'_1, T'_2, ..., T'_k$. Note that $T'_Q = U(T'_1, T'_2, ..., T'_k)$ and $T'_Q \subseteq T_Q$. We have the following lemma.

LEMMA 1. *Suppose $q \in T_Q$. $q \in SKY(T_E \cup T_Q)$ if and only if $q \in SKY(T_E \cup T'_Q)$.*

The above lemma[2] claims that a newly generated product $q \in T_Q$ is in the skyline computed according to $T'_1, T'_2, ..., T'_k$ if and only if q is in the skyline computed according to $T_1, T_2, ..., T_k$. In other words, we can just focus on finding the skyline according to $T'_1, T'_2, ..., T'_k$ instead of $T_1, T_2, ..., T_k$. Since T'_i is much smaller than T_i in general, the total number of products generated from $T'_1, T'_2, ..., T'_k$ is much smaller than that generated from $T_1, T_2, ..., T_k$. Thus, the search space is significantly reduced.

After we obtain T'_Q, we have the following interesting property.

LEMMA 2 (NON-DOMINANCE RELATIONSHIP). *If the application satisfies the at-most-one merging attribute characteristic, for any two distinct tuples q and q' in T'_Q, there is no dominance relationship between q and q'. That is, $q \not\succ q'$ and $q' \not\succ q$.*

The above lemma guarantees no intra-dominance relationship among all products generated from the resulting source tables. It is a good feature since we do not need to perform any intra-dominance checking. We only need to check the inter-dominance relationship between tuples from T'_Q and tuples from T_E, as shown in Lemma 3.

LEMMA 3. *Suppose $q \in T'_Q$. $q \in SKY(T_E \cup T'_Q)$ if and only if $q \in SKY(T_E \cup \{q\})$.*

Lemma 3 is a key to the efficiency of the algorithm to be proposed. Since, here, we can save the computation of checking the intra-dominance relationship among tuples $q \in T'_Q$, the proposed step can reduce the search space effectively.

Algorithm 1 shows the algorithm for creating competitive products.

[2]Note that, according to the above lemma, $q \in T_Q/T'_Q$ must not be in the skyline $SKY(T_E \cup T_Q)$.

Hotel	Distance-to-beach	Hotel-class	Cost
h_1	100	3	100
h_2	200	2	90
h_3	400	1	80
h_4	150	2	150
h_5	170	2	140

Table 6: A set H' of skyline tuples in H (i.e., $SKY(H)$)

Algorithm 1 Algorithm for Creating Competitive Products

```
Input: a set $T_E$ of products in the existing market and a set $T'_Q$ of
    all possible products from $T'_1, T'_2, ..., T'_k$
Output: the set $O$ of competitive products
1: $O \leftarrow \emptyset$
2: for each $q \in T'_Q$ do
3:     // if $q \in SKY(T_E \cup \{q\})$
4:     if $q$ is not dominated by any tuple in $T_E$ then
5:         $O \leftarrow O \cup \{q\}$
6: return $O$
```

THEOREM 1. *Algorithm 1 returns $SKY(T_E \cup T_Q)$.*

With Algorithm 1, intra-dominance checking steps are removed if the scenario has the at-most-one merging attribute characteristic. Thus, $|T_Q|^2$ checks are avoided. Since we focus on processing T'_Q instead of T_Q (where $T'_Q \subseteq T_Q$), the total number of inter-dominance checking steps is reduced from $|T_E| \times |T_Q|$ to $|T_E| \times |T'_Q|$. The total number of checks in this algorithm is at most $|T_E| \times |T'_Q|$. Similarly, we can find $T'_E = SKY(T_E)$ so that the number can be reduced to $|T'_E| \times |T'_Q|$. In the following, when we write T_E, we mean T'_E.

Although Algorithm 1 helps us to derive an efficient algorithm, a naive implementation still *materializes* all possible products generated from $T'_1, T'_2, ..., T'_k$ and obtains a set T'_Q, which is computationally expensive. As we described before, if γ is the size of each table T'_i, the total number of tuples in T'_Q is γ^k. In the following, we propose techniques to avoid materializing T'_Q.

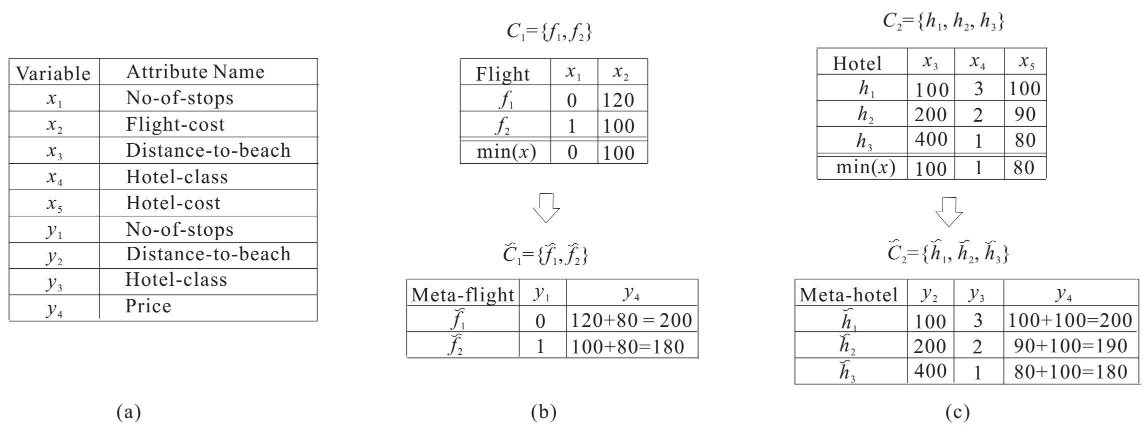
4.2 Group Partitioning

In the previous section, although we avoid the intra-dominance checking, in order to make the algorithm much more efficient, we have to reduce the number of inter-dominance checking steps. In this section, we propose a technique called *group partitioning* to further reduce the number of inter-dominance checking steps. The main idea of the group partitioning technique is to group "similar" tuples in T'_Q into a single group G, create a *best representative* for this group G, denoted by $b(G)$, and compare the tuples in T_E with this representative. With this technique, we propose two kinds of pruning, namely *full pruning*, in which we try to prune the whole groups, and *partial pruning*, in which we try to prune some members of some groups. Full pruning is described in this section while partial pruning is described in Section 4.3.

Intuitively, the best representative is a tuple which is the best among these "similar" tuples according to the following definition.

DEFINITION 8 (BEST REPRESENTATIVE). *Given a group G, a tuple t is said to be a* best representative *if t dominates all tuples in G which have some different attribute values from t.*

Using the best representative has the advantage of reducing the number of inter-dominance checking steps. For example, consider



Variable	Attribute Name
x_1	No-of-stops
x_2	Flight-cost
x_3	Distance-to-beach
x_4	Hotel-class
x_5	Hotel-cost
y_1	No-of-stops
y_2	Distance-to-beach
y_3	Hotel-class
y_4	Price

(a)

$C_1 = \{f_1, f_2\}$

Flight	x_1	x_2
f_1	0	120
f_2	1	100
min(x)	0	100

$\widehat{C}_1 = \{\widehat{f}_1, \widehat{f}_2\}$

Meta-flight	y_1	y_4
$\widehat{f}_1$	0	120+80 = 200
$\widehat{f}_2$	1	100+80=180

(b)

$C_2 = \{h_1, h_2, h_3\}$

Hotel	x_3	x_4	x_5
h_1	100	3	100
h_2	200	2	90
h_3	400	1	80
min(x)	100	1	80

$\widehat{C}_2 = \{\widehat{h}_1, \widehat{h}_2, \widehat{h}_3\}$

Meta-hotel	y_2	y_3	y_4
$\widehat{h}_1$	100	3	100+100=200
$\widehat{h}_2$	200	2	90+100=190
$\widehat{h}_3$	400	1	80+100=180

(c)

Figure 1: An example to illustrate the meta-transformation

N tuples, $q_1, q_2, ..., q_N$, in T'_Q forming a group G. Consider a particular tuple p in T_E. Without the best representative, in order to determine whether q_i is dominated by p, we have to perform N times of inter-dominance checks. However, with the best representative, we can just perform a *single* step of the dominance checking between the best representative $b(G)$ and tuple p. Thus, the number of inter-dominance checking steps may reduce from N to 1.

In our implementation, given a group G, the best representative of G is obtained by setting each attribute value of the representative to be the minimum possible attribute value among all tuples in G. It is easy to verify that the best representative t found with this method dominates all tuples in G which have different attribute values from t.

EXAMPLE 7 (BEST REPRESENTATIVE). Consider a group G containing only two products: $q : (f_2, h_4)$ and $q' : (f_2, h_5)$. Their attribute values of (No-of-stops, Distance-to-beach, Hotel-class, Price) are (1, 150, 4, 250) and (1, 170, 4, 240), respectively. We create a *best representative* for this group as (1, 150, 4, 240) by taking the minimum possible value among all tuples in G on each attribute. Note that $(1, 150, 4, 240)$ dominates both (1, 150, 4, 250) and (1, 170, 4, 240). Besides, since this best representative is dominated by an existing product p_2, we conclude that all members in the group are also dominated by p_2 and are not in the answer. □

LEMMA 4 (FULL PRUNING). *If $b(G)$ is dominated by a tuple p in T_E, all tuples in G are also dominated by p.*

The next question is how we find "similar" tuples in T'_Q to form a group G and then find the best representative $b(G)$. A straightforward solution is to perform clustering over all possible tuples in T'_Q to form groups and then find the best representative in each group. This solution has a requirement that we have to *materialize* all tuples in T'_Q by enumerating all possible products from $T'_1, T'_2, ..., T'_k$. As we mentioned before, materializing all possible tuples in T'_Q is time-consuming and there are a large number of possible tuples in T'_Q.

Instead, we leverage the way we generate products from source tables to perform clustering over the tuples in each *source table* T'_i instead of the materialized product table T'_Q. After we obtain the clusters for each source table, we (conceptually) generate a group G from one cluster of each source table. We do not materialize group G, which means that we do not enumerate all members generated from the corresponding cluster. We just keep the cluster IDs for a group G. Specifically, suppose G is formed from cluster C_1 (of table T'_1), cluster C_2 (of table T'_2), ..., cluster C_k (of table T'_k). We denote G in form of $(C_1, C_2, ..., C_k)$. Let L be the set of cluster IDs for $C_1, C_2, ..., C_k$. We just keep set L to denote group G in the implementation. Since each cluster from a source table contains "similar" tuples, the group G formed also contains "similar" tuples.

Any clustering techniques can be used in our algorithm. It opens the opportunity of leveraging the rich literature of clustering to optimize our algorithm. In our implementation, we adopt k-mean to cluster over each source table where Euclidean distance metric is used for the pairwise distance.

Although we do not enumerate all members in G, we can still create the best representative $b(G)$ by using the *best representative* of each corresponding cluster C_i of source table T'_i. The best representative of cluster C_i of a table T'_i is generated according to Definition 8 with its attributes set to X_i instead of Y. Specifically, for each cluster C_i, we create the best representative $b(C_i)$ of cluster C_i. Then, we find the best representative $b(G)$ of a group G by the following formula.

$$\theta(u_1, u_2, ..., u_k)$$

where $u_l = b(C_l)$ for $l \in [1, k]$.

EXAMPLE 8 (BEST REPRESENTATIVE). Suppose T'_1 is table flight and T'_2 is table Hotel. Consider $C_1 = \{f_2, f_3\}$ and $C_2 = \{h_4, h_5\}$. From Table 5 and Table 6, it is easy to obtain that $b(C_1)$ and $b(C_2)$ are equal to $(1, 80)$ and $(150, 2, 140)$, respectively. Suppose G is formed from C_1 and C_2. $b(G)$ is equal to $(1, 150, 2, 80 + 140) = (1, 150, 2, 220)$. Note that $b(G)$ is dominated by p_2 (See Table 1). Thus, the whole group can be pruned. □

In Algorithm 2, we adopt Algorithm 1 to include group partitioning. The major additional component of the algorithm is the introduction of full pruning (Line 9): if $b(G)$ is dominated by a tuple p in T_E, we can skip the inter-dominance checking between all tuples in G and tuples in T_E (Lemma 4).

The full pruning is used to prune the *entire* group. In Line 10, we introduce a function called *partialPrune* which is used to prune *some* tuples in G for the consideration of the inter-dominance checking. This pruning is called *partial pruning*. Details will be descrbied in Section 4.3. Function *partialPrune* removes a set W of tuples from G where each tuple in W must not be in $SKY(T_E \cup T'_Q)$.

Algorithm 2 Algorithm for Creating Competitive Products

```
Input: $T_E$ and $T'_1, T'_2, ..., T'_k$
Output: the set $O$ of competitive products
 1: $O \leftarrow \emptyset$
 2: perform clusterings over each source table $T'_i$
 3: generate a set $\mathcal{G}$ of disjoint groups conceptually according to
    the clustering results obtained in the previous step
 4: for each cluster $C_i$ of each source table $T'_i$ do
 5:     create the best representative $b(C_i)$
 6: for each group $G \in \mathcal{G}$ do
 7:     create the best representative $b(G)$ according to the best rep-
        resentatives of the correspondence clusters
 8: for each group $G \in \mathcal{G}$ do
 9:     if $b(G)$ is not dominated by any tuple in $T_E$ then
10:         $G' \leftarrow$partialPrune($G$)
11:         for each $q \in G'$ do
12:             // if $q \in SKY(T_E \cup \{q\})$
13:             if $q$ is not dominated by any tuple in $T_E$ then
14:                 $O \leftarrow O \cup \{q\}$
15: return $O$
```

Note that Algorithm 1 is a special case of Algorithm 2 if each cluster of each source table contains only 1 tuple.

4.3 Partial Pruning in Group Partitioning

Consider a group $G : (C_1, C_2, ..., C_k)$. In Algorithm 2, fucntion *partialPrune* is called if full pruning is unsuccessful (i.e., the best representative of group G, $b(G)$, is not dominated by any tuple in T_E). Note that the best representative cannot be used to remove *some* tuples in the group. This is because the best representative does not contain detailed information about the tuples in G. One way to remove some tuples in the group is a *full materialization* which enumerates all tuples in G so that we can obtain all *detailed* information. However, as described before, it is very computation-intensive.

Partial pruning is a tradeoff between the full materialization approach and the best representative approach. Specifically, we propose the following three steps for function *partialPrune*. Consider a group $G : (C_1, C_2, ..., C_k)$.

Step 1 (Meta-transformation): We do the following for each cluster C_i. Consider a cluster C_i from a source table. For each tuple t_i in C_i, we transform t_i to a tuple called a *meta-product* of t_i and then project the meta-product on some attributes to form a *meta-tuple* of t_i. The meta-tuples of all tuples in C_i form a new cluster $\widetilde{C}_i$.

A meta-product of a tuple t_i is defined as follows.

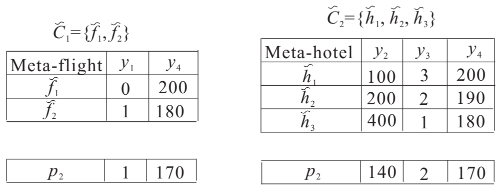
DEFINITION 9 (META-PRODUCT). *Suppose t_i is a tuple from cluster C_i. A* meta-product *of tuple t_i, denoted by $\beta(t_i)$, is equal to a product q where q is*

$$\theta(u_1, u_2, ..., u_k)$$

where $u_l = b(C_l)$ for $l \in [1, k]/\{i\}$ and $u_i = t_i$.

A meta-product of tuple t_i is similar to the best representative of $b(G)$. However, the difference is that a meta-product makes use of the real content of tuple t_i but the best representative utilizes the best possible information in C_i (instead of the real content of tuple t_i). Intuitively, a meta-product gives more detailed information compared with the best representative.

Next, we describe how we generate a meta-tuple of t_i from the meta-product by a projection operation.


$\widetilde{C}_1 = \{\widetilde{f}_1, \widetilde{f}_2\}$

Meta-flight	y_1	y_4
$\widetilde{f}_1$	0	200
$\widetilde{f}_2$	1	180

p_2	1	170

$\widetilde{C}_2 = \{\widetilde{h}_1, \widetilde{h}_2, \widetilde{h}_3\}$

Meta-hotel	y_2	y_3	y_4
$\widetilde{h}_1$	100	3	200
$\widetilde{h}_2$	200	2	190
$\widetilde{h}_3$	400	1	180

p_2	140	2	170

Figure 2: An example to illustrate how we use the projection for pruning

Note that each tuple in C_i is associated with an attribute set X_i since C_i comes from the source table T'_i. We define $\widetilde{Y}_i$ to be a set of target attributes of attributes in X_i. That is, $\widetilde{Y}_i$ is equal to $\{y|y = \alpha(x)$ where $x \in X_i\}$. Note that $\widetilde{Y}_i \subseteq Y$. With the attribute set $\widetilde{Y}_i$, we define the meta-tuple of t_i as follows.

DEFINITION 10 (META-TUPLE). *Suppose q is a meta-product of tuple t_i. A* meta-tuple *of tuple t_i, denoted by $\widetilde{t}_i$, is defined to be equal to $\prod_{\widetilde{Y}_i} q$ which is the projection of q on attribute set $\widetilde{Y}_i$.*

EXAMPLE 9 (META-TRANSFORMATION). Figure 1 illustrates the meta-transformation. Figure 1(a) shows the variables representing the attribute names in our motivating example where T'_1 is table Flight and T'_2 is table Hotel. Consider $C_1 = \{f_1, f_2\}$ and $C_2 = \{h_1, h_2, h_3\}$. Figure 1(b) shows the meta-transformation from C_1 to $\widetilde{C}_1$. Note that X_1 is equal to $\{x_1, x_2\}$. Since $\alpha(x_1) = y_1$ and $\alpha(x_2) = y_4$, we obtain $\widetilde{Y}_1 = \{y_1, y_4\}$. Note that $b(C_1)$ has $(x_1, x_2) = (0, 100)$ and $b(C_2)$ has $(x_3, x_4, x_5) = (100, 1, 80)$.

Consider how we generate the meta-tuple of f_1. Note that f_1 has $(x_1, x_2) = (0, 120)$. It is easy to obtain that the meta-product of f_i is equal to $\beta(f_1) = \theta(f_1, b(C_2))$ which is equal to $(0, 100, 1, 120 + 80) = (0, 100, 1, 200)$. Thus, $\widetilde{f}_1$ is equal to $\prod_{\widetilde{Y}_1} \beta(f_1) = (0, 200)$. In the same way, we obtain $\widetilde{f}_2 = (1, 180)$.

Similarly, from C_2, we also obtain $\widetilde{C}_2$ containing $\widetilde{h}_1, \widetilde{h}_2$ and $\widetilde{h}_3$ as shown in Figure 1(c). □

Step 2 (Dominance checking): After the transformation, for each transformed cluster $\widetilde{C}_i$ and each tuple $p \in T_E$, we determine a set of meta-tuples in $\widetilde{C}_i$ such that each of these meta-tuples is dominated by a tuple $p \in T_E$ with respect to $\widetilde{Y}_i$. We denote this set by $\gamma(C_i, p)$.

EXAMPLE 10 (DOMINANCE CHECKING). Figure 2 shows $\widetilde{C}_1$ and $\widetilde{C}_2$ from Example 9. Consider a tuple p_2 in T_E.

We can see that p_2 dominates $\widetilde{f}_2$ only in $\widetilde{C}_1$ with respect to $\widetilde{Y}_1$. It also dominates $\widetilde{h}_2$ only in $\widetilde{C}_2$ with respect to $\widetilde{Y}_2$. We have $\gamma(C_1, p_2) = \{f_2\}$ and $\gamma(C_2, p_2) = \{h_2\}$. □

Step 3 (Meta-pruning): According to the information obtained in Step 2, we can determine which tuples in G can be pruned for each $p \in T_E$.

Consider a tuple $p \in T_E$. We can use the content of $\gamma(C_i, p)$ for pruning some tuples in G by the following lemma. Let $W(p)$ be a set of possible combinations generated from $\gamma(C_1, p), \gamma(C_2, p), ..., \gamma(C_k, p)$. That is $W(p) = \{\theta(t_1, t_2, ..., t_k)|t_i \in \gamma(C_i, p)$ for $i \in [1, k]\}$. The following lemma suggests that we can prune any tuples in $W(p)$.

LEMMA 5 (PARTIAL PRUNING). *Let p be a tuple in T_E. Each tuple $q \in W(p)$ is not in $SKY(T_E \cup T'_Q)$.*

From Example 10, we know that $W(p_2) = \{\theta(f_2, h_2)\}$. Thus, we do not need to consider the product $\theta(f_2, h_2)$.

With Lemma 5 and Theorem 1, it is easy to verify the following.

THEOREM 2. *Algorithm 2 returns $SKY(T_E \cup T_Q)$.*

4.4 Implementation

We will describe how we can use some indexing techniques to speed up the inter-dominance checks. The major part of Algorithm 2 is to perform the inter-dominance checks between tuples in T_E and tuples in T'_Q. In our implementation, we build an R*-tree R_E over T_E. Suppose that Y contains $y_1, y_2, ..., y_u$. This tree can be used in full pruning and partial pruning. In full pruning, when we check whether a best representative b_o is dominated by an existing product in T_E, we perform a range query with range $(y_1 \leq b_o.y_1) \wedge (y_2 \leq b_o.y_2) \wedge ... \wedge (y_u \leq b_o.y_u)$. If the range query returns a set A of products which have some attribute values different from b_o, then b_o is dominated by $p \in A$. Otherwise, b_o is not dominated by any tuples in T_E.

In partial pruning, similarly, we can also use the R*-tree R_E as follows. We want to find $\gamma(C_i, p)$ for each $p \in T_E$ and each cluster C_i. Initially, we set $\gamma(C_i, p) = \emptyset$. For each meta-tuple $\widetilde{t_i}$ in $\widetilde{C_i}$, we find a set of existing tuples in T_E dominating $\widetilde{t_i}$ with respect to $\widetilde{Y_i}$ by a range query. Specifically, suppose that $\widetilde{Y_i}$ contains $y_1, y_2, ..., y_v$. Let the attributes in Y but not in $\widetilde{Y_i}$ be $y_{v+1}, y_{v+2}, ..., y_u$. We perform a range query $(y_1 \leq \widetilde{t_i}.y_1) \wedge (y_2 \leq \widetilde{t_i}.y_2) \wedge ... \wedge (y_v \leq \widetilde{t_i}.y_v) \wedge (y_{v+1} \leq \infty) \wedge (y_{v+2} \leq \infty) \wedge ... \wedge (y_u \leq \infty)$. Let R be the range query result containing products p which has some attribute values different from $\widetilde{t_i}$ with respect to $\widetilde{Y_i}$. For each $p \in R$, we insert $\widetilde{t_i}$ into $\gamma(C_i, p)$.

5. DISCUSSION

We first describe how the clustering quality affects our proposed method. Then, we discuss how our proposed algorithm can be extended to a general case.

Clustering Quality Issue: The clustering quality may affect the performance of full pruning. Let $G = (C_1, C_2, ..., C_k)$. If each cluster C_i contains many "similar" tuples, then G contains many "similar" tuples. Suppose each attribute of these tuples in G has a large value. It is very likely that the best representative of the group G is dominated by tuples in T_E. However, suppose that a cluster C_i contains some "distant" tuples such that a tuple in G have an attribute value which is much smaller compared with another tuple in G. It is less likely that the best representative of the group G, taking the smallest possible attribute value among all tuples in G, is dominated by tuples in T_E.

We want to emphasize that the clustering quality does not affect the correctness of the algorithm. If the cluster contains "distant" tuples, then the entire group cannot be pruned and thus has to be processed in the later steps of the algorithm.

General Model: In Section 4, we assume that the application satisfies the at-most-one merging attribute characteristic. With this characteristic, by Lemma 2, we can avoid the intra-dominance checking. If this characteristic is not satisfied, Lemma 2 does not hold and thus we have to perform the intra-dominance checking. In this case, after obtaining the answer O from Algorithm 2, we add a post-processing step which computes $SKY(O)$, which corresponds to $SKY(T_E \cup T_Q)$. We call this algorithm with the post-processing step the algorithm for creating competitive products (ACCP).

THEOREM 3. *Algorithm ACCP returns $SKY(T_E \cup T_Q)$.*

Parameter	Default value
No. of attributes in each source table (N)	4
No. of indirect attributes in a product table (I)	1
No. of source tables (k)	2
Size of T_E ($\lvert T_E \rvert$)	$5M$
Size of each source table ($\lvert T_i \rvert$)	$100k$

Table 7: Default values of parameters

6. EMPIRICAL STUDIES

We have conducted extensive experiments on a Pentium IV 2.4GHz PC with 4GB memory, on a Linux platform. The algorithms were implemented in C/C++. We conducted the experiments on both synthetic and real datasets.

The synthetic dataset is generated by a dataset generator. The dataset generator has five input parameters, namely (1) the number of attributes in each source table, N, (2) the number of indirect attributes in the product table, I, (3) the number of source tables, k, (4) the number of tuples in table T_E, $|T_E|$, and (5) the number of tuples in each source table, $|T_i|$. We generate the datasets as follows. Firstly, we create k source tables, namely $T_1, T_2, ..., T_k$. We adopted the data set generator released by the authors of [1]. For each source table T_i, as in [1], we generate the anti-correlated dataset containing $|T_i|$ tuples with N attributes each of which has a range from 0 to 1000. Details of the generation of this dataset can be found in [1]. Let $\mathcal{X}$ be the set of attributes of all source tables. Secondly, we generate table T_E as follows. We generate I indirect attributes. For each indirect attribute y in T_E, we randomly pick a value M from a distribution with mean 2 with standard derivation 1 to find the number of dependent attributes of y. Then, we randomly pick M attributes from $\mathcal{X}$ to be $D(y)$ and remove them from $\mathcal{X}$. For each of the remaining attributes x in $\mathcal{X}$, we create a direct attribute y in Y such that $D(y) = \{x\}$. We generate a set D of all possible combinations from $T_1, T_2, ..., T_k$. Then, we randomly select $|T_E|$ tuples from D and store them as V. For each tuple t in V, we modify each attribute of t by multiplying a number x which follows a normal distribution with mean 1.0 and variance 0.025. All modified tuples t form the final table T_E. If the parameters are not specified, we adopt the default values in Table 7.

The real datasets are obtained from two anonymous travel agencies, namely Agency A and Agency B. From each travel agency, we obtained all packages, all flights and all hotels for a round trip traveling from San Francisco to New York for a period from March 1, 2009 to March 7, 2009. In Agency A dataset, we have 296 packages, 1014 hotels and 4394 flights. In the Agency B dataset, we have 149 packages, 995 hotels and 866 flights. Hotels and flights form two source tables, and packages forms table T_E. Hotels have attributes, namely *quality-of-room,*, *customer-grading*, *hotel-class*, *hotel-price*, while flights have attributes, namely *class-of-flight*, *no-of-stops*, *duration-of-journey* and *flight-price*. Packages have four attributes, namely *quality-of-room*, *customer-grading*, *hotel-class,class-of-flight*, *no-of-stops*, *duration-of-journey* and *price*. Same as our motivating application, in T_E, attribute *price* is an indirect attribute where *price* is equal to the sum of attribute *hotel-price* and attribute *flight-price*, and others are direct attributes.

We denote our proposed algorithm as *ACCP*. This also involves two major steps. The first step is called *preprocessing step*, which finds $SKY(T_i)$ for each source table T_i and finds $SKY(T_E)$ for the table T_E. We also build an R*-tree R_E on T'_E where $T'_E = SKY(T_E)$. The second step is to use Algorithm 2 to find all competitive products. We adopt k-mean for clustering over each source table where k used in k-mean is equal to $|T_i|/1000$. Thus, the average cluster size is equal to 1000.

We also compared algorithm *ACCP* with two algorithms, namely *naive* and *baseline*. *Naive* is an algorithm which generates all possible combinations from $T_1, T_2, ..., T_k$ and stores them in T_Q. Then, it forms a dataset $\mathcal{D} = T_E \cup T_Q$ and use the existing skyline algorithm called SFS [4] to find the skyline in $\mathcal{D}$. *Baseline* is same as *ACCP* without full pruning and partial pruning.

We evaluated the algorithms in terms of seven measurements: (1) *Preprocessing:* We measured the time of the pre-processing step. (2) *Execution time:* The execution times of algorithms are measured. For *Baseline* and *ACCP*, in order to analyze the execution time of the framework of the algorithms, the time of the post-processing step is not reported. (3) $|SKY|/|T_Q|$: Let T_Q be the set of all possible combinations from the original source tables $T_1, T_2, ..., T_k$ (i.e., $T_Q = U(T_1, T_2, ..., T_k)$). Let $SKY = SKY(T_E \cup T_Q) \cap T_Q$. $|SKY|/|T_Q|$ corresponds to the proportion of skyline tuples among all tuples in T_Q. (4) $|SKY|/|T'_Q|$: $|SKY|/|T'_Q|$ corresponds to the proportion of skyline tuples among all tuples in T'_Q. (5) $|T_R|/|T_Q|$: Let T_R be the set of remaining products after full pruning and partial pruning. $|T_R|/|T_Q|$ corresponds to the proportion of remaining products after full pruning and partial pruning among all products in T_Q. (6) $|T_R|/|T'_Q|$: $|T_R|/|T'_Q|$ corresponds to the proportion of remaining products after full pruning and partial pruning among all products in T'_Q. (7) *Memory:* The memory usage of algorithm *ACCP* is the memory consumed by the R*-tree built on T'_E where $T'_E = SKY(T_E)$ and the temporary storage in the algorithm *ACCP* to store groups G' after full pruning and partial pruning.

6.1 Synthetic dataset

We first compare our algorithms, namely *Baseline* and *ACCP*, with *Naive* in Section 6.1.1 to show that *Naive* is not scalable to large datasets. In Section 6.1.2, we give a comprehensive experimental studies to study the scalability of our algorithms.

6.1.1 Comparison with Naive Algorithm

In the synthetic dataset where $N = 3, I = 5, k = 2, |T_E| = 10000$ and $|T_i| = 5000$, *Naive* took 1G memory and ran for hours. Both the memory usage and the execution time of *Naive* are several thousand times more than those of *baseline* and *ACCP*. Since *Naive* is not scalable to large datasets, in the following, we focus on the comparisons between algorithm *ACCP* and algorithm *Baseline*.

6.1.2 Scalability

In the following, we study the following factors: (a) the source table size, (b) the size of T_E, (c) the number of indirect attributes of each product table, (d) the number of attributes of the product table, (e) the number of source tables, and (f) the number of clusters in each source table.

Effect of the source table size: We change the size of source tables from 100k to 500k. Figure 3(a) shows that the preprocessing times and the execution times of both algorithms increase with the source table size. The execution time of algorithm *ACCP* is smaller than that of algorithm *Baseline* because algorithm *ACCP* performs full pruning and partial pruning, which speeds up the computation. In Figures 3(b) and (c), $|SKY|/|T_Q|$, $|SKY|/|T'_Q|$, $|T_R|/|T_Q|$, $|T_R|/|T'_Q|$ and $|T_R|/|SKY|$ remains nearly unchanged. In Figure 3(b), we observe that $|SKY|/|T'_Q|$ is larger than $|SKY|/|T_Q|$. This means that $|T'_Q|$ is smaller than $|T_Q|$, which shows the effectiveness of the step to produce T'_Q for the dataset. In Figure 3(c), $|T_R|/|T_Q|$ decreases by an order of magnitude when the source table size increases while $|T_R|/|T'_Q|$ remains relatively constant. A smaller value of $|T_R|/|T_Q|$ (or $|T_R|/|T'_Q|$) means that the search space is larger. Thus, the trend shows that creating T'_Q becomes more effective when the source table size increases. Figure 3(d) shows that the memory is more or less the same when the source table size changes.

Effect of the size of T_E: We also conducted experiments to study the effect of the size of T_E by varying from 2.5M to 10M. The results are similar to those for the effect of the source table size. Figure 4(a) shows that *ACCP* is also faster than *Baseline*. When the size of T_E is larger, the execution times of both algorithms decrease. This is because there are more products in T_E dominating tuples in T_Q. So, it is more likely that a tuple in T_Q is dominated by a tuple in T_E. Once a tuple q in T_Q is dominated by a tuple in T_E, the dominance checking between q and the remaining tuples in T_E can be skipped. Thus, the execution times are lower. Figure 4(b) shows that $|SKY|/|T_Q|$ and $|SKY|/|T'_Q|$ decreases when $|T_E|$ increases. In Figure 4(c), $|T_R|/|T_Q|$, $|T_R|/|T'_Q|$ and $|T_R|/|SKY|$ remains nearly unchanged when $|T_E|$ increases. Figure 4(d) shows that the memory consumptions of both algorithms increase slightly with $|T_E|$.

Effect of the number of indirect attributes of the product table: We conducted experiments to study the effect of the number of indirect attributes of the product table by changing from 1 to 7. We fix the number of attributes to be 7. The execution time of algorithm *ACCP* is within 3,000s. Figure 5(b) shows that when the number of indirect attributes increases, $|SKY|/|T_Q|$ remains nearly unchanged. However, $|SKY|/|T'_Q|$ decreases. This is because the size of T'_Q increases a lot. In Figure 5(c), as the number of indirect attributes is larger, $|T_R|/|T'_Q|$ is very large. This is because, when there are more indirect attributes in the product table, it is less likely that a tuple is dominated by another tuple. Thus, it is less likely that full pruning and partial pruning are successful.

Effect of the number of attributes of the product table: We studied the effect of the number of attributes of the product table where we fix the number of indirect attributes of the product table to be 1. The results are similar to Figure 5. For the sake of space, we omit the figure here.

Effect of the number of source tables: Figure 6 shows the results when we vary the number of source tables where $|T_E| = 100k, |T_i| = 1k, I = 5$ and $N = 3$. In the figure, the preprocessing time, the execution times of both algorithms, $|SKY|/|T_Q|$, $|SKY|/|T'_Q|$, $|T_R|/|T_Q|$, $|T_R|/|T'_Q|$, $|T_R|/|SKY|$ and the memory increases with the number of source tables. This is because with more source tables, $|T_Q|$ is larger. Thus, the execution time, the set of skyline tuples in the final dataset and the memory are larger.

Effect of the number of clusters: We conducted experiments to study the effect of the number of clusters over a source table. We varied the number of clusters from 6 to 30. The results are shown in Figure 7. Since *Baseline* is independent of the number of clusters, we do not include the results for *Baseline* in the figure. When the number of clusters increases, $|T_R|/|T_Q|$, $|T_R|/|T_Q|$ and $|T_R|/|SKY|$ decreases. This is because the cluster size decreases when there are more clusters. Thus, each group formed from one cluster of each source table is smaller. There are more groups which contain large attribute values. Thus, it is more likely that they are dominated by tuples in T_E. Thus, $|T_R|$ is smaller.

6.2 Real Dataset

In the real dataset, we conducted two sets of experiments, namely *Agency A Package Generation Set* and *Agency B Package Generation Set*. Let H_A (F_A) be the source tables of Agency A for Hotel (Flight). Let H_B (F_B) be the source tables of Agency B for

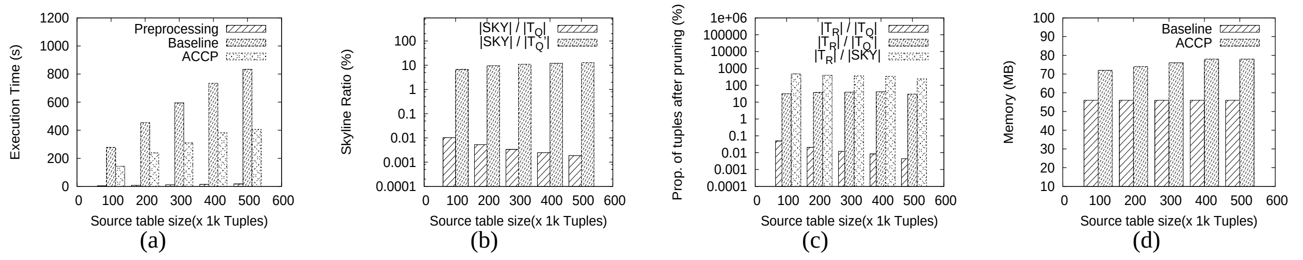


Figure 3: Effect of the source table size

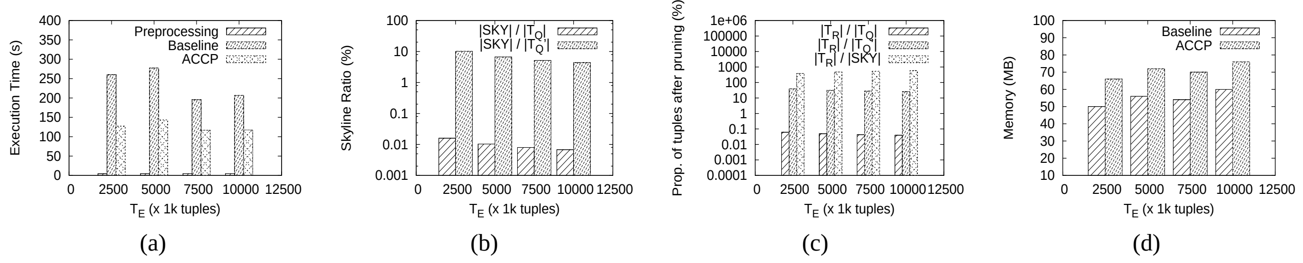


Figure 4: Effect of the size of T_E

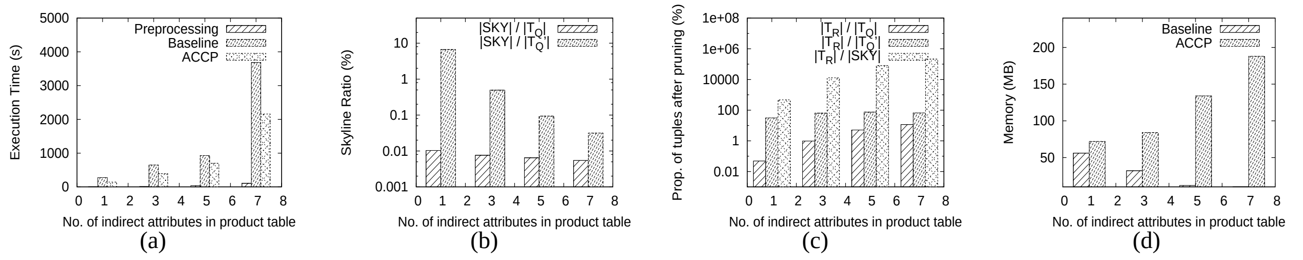


Figure 5: Effect of the number of indirect attributes of each product table

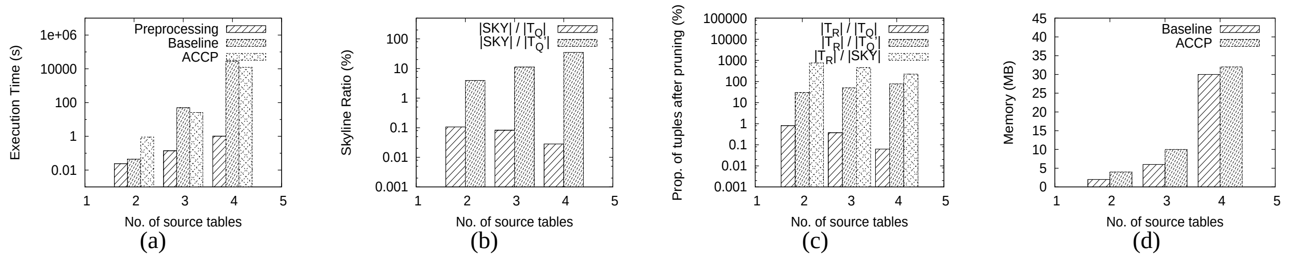


Figure 6: Effect of the number of source tables

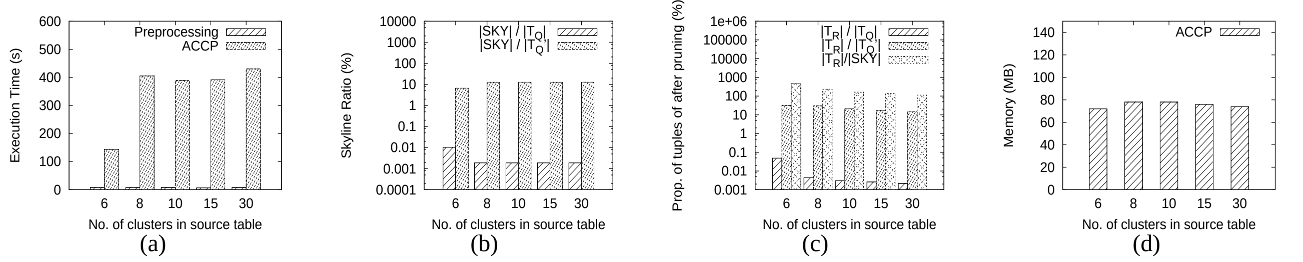


Figure 7: Effect of the number of clusters in each source table

Hotel (Flight). Suppose $T_{E,A}$ ($T_{E,B}$) is the product table storing the existing packages in agency A (agency B). In the Agency A Package Generation Set, we generate new packages from hotels and flights of Agency A and find which new packages are competitive in the existing market including new packages and the packages from Agency B That is, we want to find $SKY(T_Q \cup T_{E,B})$ where T_Q is the product table generated from H_A and F_A. The Agency B Package Generation Set is similar to Agency A Package Generation Set but the source tables come from Agency B and the existing packages come from Agency A. That is, we want to generate new packages from hotels and flights of Agency B and to find which new packages are competitive in the existing market including new packages and the packages from Agency A.

In Agency A Package Generation Set, the execution times of *ACCP* and *Baseline* are 44.74s and 84.47s, respectively. In Agency B Package Generation Set, the execution times of *ACCP* and *Baseline* are 10.43s and 27.14s, respectively.

The merging function of attribute Price is equal to the sum of attribute Flight-cost and attribute Hotel-price in our motivating example. In the following, we want to study the effect when the merging function is in another form. Consider that the merging function of attribute Price is equal to the sum of attribute Flight-cost and attribute Hotel-price *multiplied* by $(1 - r)$ where r is a discount rate. In the real travel agency sites, usually, when customers choose flights and hotels together, they will obtain a discount.

We conducted experiments for each set and measured the following: (1) $|SKY|/|T_Q|$: SKY is equal to $SKY(T_E \cup T_Q) \cap T_Q$. Thus, $|SKY|/|T_Q|$ is equal to the ratio of the tuples in T_Q which are in the skyline in dataset $T_E \cup T_Q$. and (2) $|DOM|/|T_E|$: DOM is equal to the number of tuples in T_E dominated by the newly generated packages in T_Q. Thus, $|DOM|/|T_E|$ is equal to the ratio of tuples in T_E dominated by some newly generated packages.

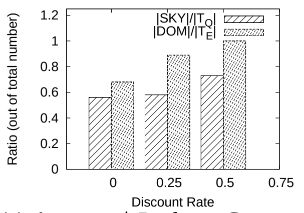
Figure 8(a) shows that $|DOM|/|T_E|$ increase with the discount rate r for the Agency A Package Generation Set. This is because when r increases, the price of the products in T_Q decreases. It is more likely that the products in T_Q dominates tuples in T_E. Thus, $|DOM|$ increases. $|SKY|/|T_Q|$ remains nearly unchanged when r increases. In the figure, $|SKY|/|T_Q|$ is greater than 0.5 for different values of r, which means most newly created packages are competitive. Surprisingly, when there is no discount (i.e., $r = 0$), $|DOM|/|T_E|$ is also greater than 0.5, which means that the newly created packages are "better" than half of the existing packages in the market. Thus, the newly created packages are quite competitive, which suggests that many existing packages may not be too "good" to customers. Figure 8(b) shows similar results for the Agency B Package Generation Set.

Conclusion: Algorithm Naive is not scalable to large datasets. Algorithms Baseline and ACCP perform thousand times faster than algorithm Naive. Algorithm ACCP (with full pruning and partial pruning) runs faster than algorithm Baseline (without full pruning and partial pruning).

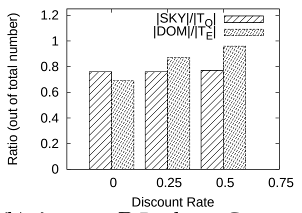
7. RELATED WORK

Skyline queries have been studied since 1960s in the theory field where skyline points are known as *Pareto sets* and *admissible points* [8] or *maximal vectors* [6]. However, earlier algorithms such as [6, 7] are inefficient when there are many data points in a high dimensional space. The problem of skyline queries was introduced in the database context in [1].

We can categorize the existing work into two major groups – *single-table skyline queries* and *multiple-table skyline queries*.



(a) Agency A Package Generation Set



(b) Agency B Package Generation Set

Figure 8: Results for real datasets

There are a lot of efficient methods proposed for single-table skyline queries where the tuples considered are based on a single table. Some representative methods include a bitmap method [17], a nearest neighbor (NN) algorithm [12], and branch and bound skylines (BBS) method [13]. Recently, skyline computation has been extended to subspace skyline queries [22, 14, 21] where the computation returns the skylines with respect to all possible subsets of attributes. Besides, the above skyline queries are based on numerical attributes. Recently, [3, 2, 19, 15] proposes some methods which can handle categorical attributes in addition to numeric attributes. However, *all of the above works are also based on a single table.*

Multiple-table skyline queries [11, 16] return the skyline based on *multiple* tables instead of a *single* table. [11, 16] study how to perform a *natural join* over multiple relational tables, generate one joined table and find the skyline in the joined table. The basic assumption of a natural join operation over multiple relational tables is that for each table T_1, one of its attributes, says x_1, is *associated* with an attribute x_2 of another table T_2 where x_1 and x_2 are a *primary key* of T_1 and a *foreign key* of T_2 (to T_1), respectively, or vice versa. However, they only consider how to join the tables where a *foreign key* of a table is a *primary key* of another table. Thus, their focus is to find how to *match* the value of a foreign key with the value of a primary key. Our work is fundamentally different from the works about natural joins [11, 16]. This is because their works are based on foreign keys but our work considers how to perform a *cartesian product* over multiple tables without any foreign key.

Creating products studied in this paper introduce challenges. This is because a tuple in a table can be combined with *any* tuple in another table such that the product is in the skyline. Thus, our focus is to find which potential tuples in some tables can be combined with a given tuple in a table such that the combined products are in the skyline.

8. CONCLUSION

In this paper, we identify and tackle the problem of creating competitive products, which has not been studied before. We propose a method to find competitive products efficiently. An extensive performance study using both synthetic and real datasets is reported to verify its effectiveness and efficiency. As future work, creating competitive products with dynamic data and creating the top-K interesting competitive products are interesting topics.

Acknowledgements: We are grateful to the anonymous reviewers for their constructive comments on this paper.

9. REFERENCES

[1] S. Borzsonyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.
[2] C. Chan, P.-K. Eng, and K.-L. Tan. Efficient processing of skyline queries with partially-ordered domains. In *ICDE*, 2005.
[3] C.-Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified computation of skylines with partially-ordered domains. In *SIGMOD*, 2005.

[4] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, 2003.

[5] H. K. et al. On finding the maxima of a set of vectors. In *Journal of ACM, 22(4)*, 1975.

[6] J. L. B. et al. On the average number of maxima in a set of vectors and applications. In *Journal of ACM, 25(4)*, 1978.

[7] J. L. B. et al. Fast linear expected-time algorithms for computing maxima and convex hulls. In *SODA*, 1990.

[8] O. B.-N. et al. On the distribution of the number of admissable points in a vector random sample. In *Theory of Probability and its Application, 11(2)*, 1966.

[9] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *International Conference on Very Large Data Bases (VLDB)*, 2005.

[10] B. Jiang, J. Pei, X. Lin, D. W.-L. Cheung, and J. Han. Mining preferences from superior and inferior examples. In *SIGKDD*, 2008.

[11] W. Jin, M. Ester, Z. Hu, and J. Han. The multi-relational skyline operator. In *ICDE*, 2007.

[12] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, 2002.

[13] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. In *ACM Transactions on Database Systems, Vol. 30, No. 1*, 2005.

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

[15] D. Sacharidis, S. Papadopoulos, and D. Papadias. Topologically-sorted skylines for partially-ordered domains. In *ICDE*, 2009.

[16] D. Sun, S. Wu, J. Li, and A. K. Tung. Skyline-join in distributed databases. In *ICDE Workshop*, 2008.

[17] K.-L. Tan, P. Eng, and B. Ooi. Efficient progressive skyline computation. In *VLDB*, 2001.

[18] Q. Wan, R. C.-W. Wong, I. F. Ilyas, M. T. Ozsu, and Y. Peng. Creating competitive products. In *http://www.cse.ust.hk/∼raywong/paper/createCompetitiveProduct-technical.pdf*, 2009.

[19] R. C.-W. Wong, A. W.-C. Fu, J. Pei, Y. S. Ho, T. Wong, and Y. Liu. Efficient skyline querying with variable user preferences on nominal attributes. In *VLDB*, 2008.

[20] R. C.-W. Wong, J. Pei, A. W.-C. Fu, and K. Wang. Mining favorable facets. In *SIGKDD*, 2007.

[21] T. Xia and D. Zhang. Refreshing the sky: The compressed skycube with efficient support for frequent updates. In *SIGMOD*, 2006.

[22] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *VLDB*, 2005.

[23] Z. Zhang, L. Lakshmanan, and A. K. Tung. On domination game analysis for microeconomic data mining. In *TKDD*, to appear.

Proof of Lemmas/Theorems:

For the sake of space, we only included the proof of Lemma 1 and the proof of Lemma 2. The proof of other lemmas and theorems can be found in [18].

Proof of Lemma 1: Suppose $q \in SKY(T_E \cup T_Q)$. There is no tuple $q' \in T_E \cup T_Q$ dominating q. Since $T'_Q \subseteq T_Q$, there is no tuple $q' \in T_E \cup T'_Q$ dominating q and thus $q \in SKY(T_E \cup T'_Q)$.

Suppose $q \in SKY(T_E \cup T'_Q)$. Thus, $q \in T'_Q$. Besides, there is no tuple $q' \in T_E \cup T'_Q$ dominating q. We want to prove that $q \in SKY(T_E \cup T_Q)$.

We prove by contradiction. Suppose there exists a tuple $q' \in T_Q/T'_Q$ dominating q. Suppose q is generated from $t_1, t_2, ..., t_k$ and q' is generated from $t'_1, t'_2, ..., t'_k$. where $t_i \in T'_i$ and $t'_i \in T_i$. Since $q' \in T_Q/T'_Q$, we know that there exist a tuple $t'_j \in T_j/T'_j$ among $t'_1, t'_2, ..., t'_k$. Let W be the set of all these tuples t'_j (among $t'_1, t'_2, ..., t'_k$). We deduce that $t'_j \in W$ is dominated by $t''_j \in T'_j$ (since $T'_j = SKY(T_j)$). Let V be the set of tuples used for generating q'. That is, $V = \{t'_1, t'_2, .., t'_k\}$. If we replace each tuple t'_j in V which also appears in W with the correspondence tuples t''_j, we obtain another set V'. Note that V' contains all tuples in T'_j instead of T_j/T'_j. Consider another tuple q'' generated from V'. We know that $q'' \in T'_Q$. We conclude that $q'' \in T'_Q$ dominates $q' \in T'_Q$. Since q' dominates q, we deduce that q'' dominates q. Note that both q'' and q are in T'_Q. Thus, $q \notin SKY(T_E \cup T'_Q)$. This leads to a contradiction that $q \in SKY(T_E \cup T'_Q)$. □

Proof of Lemma 2: We prove by contradiction. Suppose that there exists two distinct tuples q and q' in T'_Q such that there is a dominance relationship between q and q'. Without loss of generality, we assume that $q' \prec q$. In the at-most-one merging attribute characteristic, for each X_i, there exists at most one $x \in X_i$ such that x is involved in the merging function of an indirect attribute $y \in Y$. We call $x \in X_i$ is *merging* if it is involved in the merging function of an indirect attribute $y \in Y$. We call $x \in X_i$ is *non-merging* if it is involved in the merging function of a direct attribute $y \in Y$. In the at-most-one merging attribute characteristic, each source table has at most one merging attribute. Given an attribute x in a source table T_j, we define $s(x)$ to be j.

Suppose that q is generated from $t_1, t_2, ..., t_k$ such that $t_j \in T'_j$ for $j \in [1, k]$. That is, $q = \theta(t_1, t_2, ..., t_k)$. Also suppose that q' is generated from $t'_1, t'_2, ..., t'_k$ such that $t'_j \in T'_j$ for $j \in [1, k]$. That is, $q' = \theta(t'_1, t'_2, ..., t'_k)$. Since t_j and t'_j come from T'_j, we know that $t_j \not\prec t'_j$ and $t'_j \not\prec t_j$ for each $j \in [1, k]$. We can also deduce that, for all $x \in X_i$ where x is non-merging,

$$t'_j.x \preceq t_j.x \qquad (2)$$

where $j \in [1, k]$. Otherwise, q' does not dominate q.

Since q' dominates q in T'_Q, we know that there exists a dimension y in Y and all other dimensions y' in Y such that $q'.y \prec q.y$ and $q'.y' \preceq q.y'$. Consider two cases. *Case 1: y is an indirect attribute.* There exists $x \in D(y)$ such that

$$t'_j.x \prec t_j.x \qquad (3)$$

where $j = s(x)$. Since x is involved in the merging function of the indirect attribute y, x is merging. Since each source table has at most one merging attribute, from (2) and (3), we deduce that $t'_j.x \prec t_j.x$ for attribute x and $t'_j.x' \preceq t_j.x'$ for other attributes $x' \in X_j$. Thus, $t'_j \prec t_j$. This leads to a contradiction that $t'_j \not\prec t_j$.

Case 2: y is a direct attribute. $D(y)$ contains one attribute, says x. Since $q'.y \prec q.y$, we have

$$t'_j.x \prec t_j.x \qquad (4)$$

where $j = s(x)$. Note that x is non-merging. Consider two cases. *Case (a):* Table T_j does not contain any merging attribute. Thus, all attributes in X_j are non-merging. Similarly, from (2) and (4), we know that $t'_j.x \prec t_j.x$ for attribute x and $t'_j.x' \preceq t_j.x'$ for other attributes $x' \in X_j$. We conclude that $t'_j \prec t_j$. This leads to a contradiction that $t'_j \not\prec t_j$.

Case (b): Table T_j contains a merging attribute x'. We further consider two sub-cases. *Case (i):* $t'_j.x' \preceq t_j.x'$. From (2) and (4), We know that $t'_j.x \prec t_j.x$ for attribute x and $t'_j.x'' \preceq t_j.x''$ for other attributes $x'' \in X_j$. We conclude that $t'_j \prec t_j$, which leads to a contradiction that $t'_j \not\prec t_j$.

Case (ii): $t'_j.x' \succ t_j.x'$. Let $y' = \alpha(x')$. Since $q'.y' \preceq q.y'$, there exists $x'' \in D(y')$ such that $t'_l.x'' \prec t_l.x''$ where $l = s(x'')$. Note that x'' is merging. Consider two cases: *Case (A): T_l has at least one non-merging attribute.* Thus, from (2), we conclude that $t'_l.x'' \prec t_l.x''$ for attribute x'' and $t'_l.x \preceq t_l.x$ for other attributes $x \in X_l$. Thus, $t'_l \prec t_l$, which leads to a contradiction that $t'_l \not\prec t_l$. *Case (B): T_j has no non-merging attribute.* Since $t'_l.x'' \prec t_l.x''$ and there is only one merging attribute, we know that $t'_l \prec t_l$, which leads to a contradiction that $t'_l \not\prec t_l$. □