Combination Skyline Queries

Xi Guo[1], Chuan Xiao[2], and Yoshiharu Ishikawa[2,1,3]

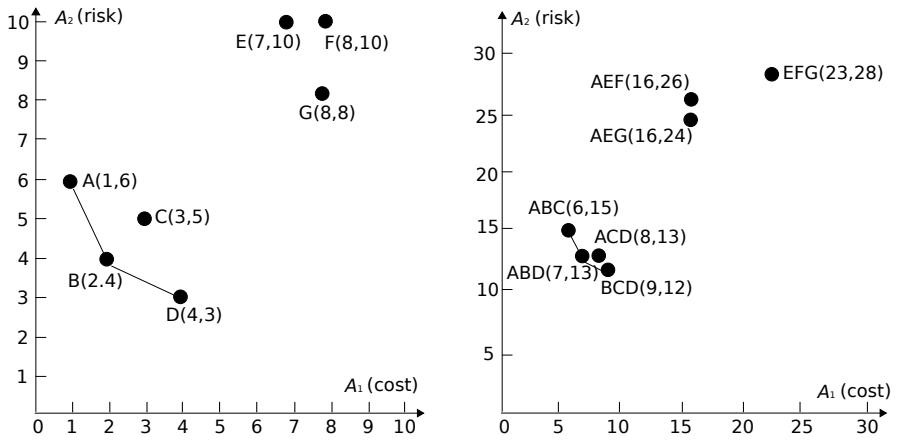
[1] Graduate School of Information Science, Nagoya University, Japan
[2] Information Technology Center, Nagoya University, Japan
[3] National Institute of Informatics, Japan

Abstract. Given a collection of data objects, the skyline problem is to select the objects which are not dominated by any others. In this paper, we propose a new variation of the skyline problem, called the combination skyline problem. The goal is to find the fixed-size combinations of objects which are skyline among all possible combinations. Our problem is technically challenging as traditional skyline approaches are inapplicable to handle a huge number of possible combinations. By indexing objects with an R-tree, our solution is based on object-selecting patterns that indicate the number of objects to be selected for each MBR. We develop two major pruning conditions to avoid unnecessary expansions and enumerations, as well as a technique to reduce space consumption on storing the skyline for each rule in the object-selecting pattern. The efficiency of the proposed algorithm is demonstrated by extensive experiments on both real and synthetic datasets.

Keywords: Skyline queries, combinations, dominance relationships, R-trees.

1 Introduction

Given a set of objects $\mathcal{O}$ where each $o_i \in \mathcal{O}$ has m-dimensional attributes $\mathcal{A} = \{A_1, \ldots, A_m\}$, a *skyline query* [2] returns the objects that are not dominated by any other objects. An object *dominates* another object if it is not worse than the other in every attribute and strictly better than the other in at least one attribute. Skyline problems exist in various practical applications where trade-off decisions are made in order to optimize several important objectives. Consider an example in the financial field: An investor tends to buy the stocks that can minimize the commission costs and predicted risks. Therefore, the goal can be modeled as finding the skyline with minimum costs and minimum risks. Fig. 1(a) shows seven stock records with their costs (A_1-axis) and risks (A_2-axis). A, B, and D are the stocks that are not dominated by any others and hence constitute the skyline. Skyline computation has received considerable attention from the database community [4, 7, 14] after the seminal paper [2], yet only a few studies explored the scenario where users are interested in combinations of objects instead of individuals. For the stock market example, assume that each portfolio consists of five stocks and its cost (risk) is the sum of costs (risks) of its



(a) A Skyline Problem (b) A Combination Skyline Problem

Fig. 1: Skyline and Combination Skyline

components. Users may want to choose the portfolios which are not dominated by any others in order to minimize the total costs and the total risks.

In this paper we investigate the *combination skyline query problem*. Its goal is to find combinations that are not dominated by any other combinations. We focus on the combinations consisting of a fixed number of individual objects, and their attribute values are the aggregations of those from its members.

Example 1 *Fig. 1(b) shows some three-item combinations consisting of stock records. Assume that their attribute values are the sums of their components' values, and the combination skyline problem is to find combinations that have minimal values in attributes A_1 (cost) and A_2 (risk). Combinations $\{ABC, ABD, BCD\}$ cannot be dominated by any others and thus they are the answers for the combination skyline query.*

There have been a few studies on the combination skyline problem. [18] proposed a solution to find the top-k optimal combinations according to a user-defined preference order of attributes. However, it is difficult to define a user preference beforehand for some complicated decision making tasks. [17] tries to find the skyline combinations that are on the convex hull enclosing all the combinations, yet it will miss other many combinations on the skyline which provide meaningful results. In this paper, we present an efficient solution that constructs the whole combination skyline, within which the user may select a smaller subset of his interest [12, 16, 19].

For the combination skyline query problem, the number of combinations is $\binom{|\mathcal{O}|}{k}$ for a database containing $|\mathcal{O}|$ objects when we select combinations of size k. This poses serious algorithmic challenges compared with the traditional skyline problem. As Example 1 shows, $\binom{7}{3} = 35$ possible combinations are generated

from only seven objects. Even for a small database with thousands of entries, the number of combinations of objects is prohibitively large.

A naïve way to answer constrained combination skyline query is to employ the existing skyline approaches [4, 7, 14] by regarding each enumerated combination as a single object. However, the huge number of enumerations renders them inapplicable for large datasets. In addition, some prevalent skyline approaches such as the BBS algorithm [14] uses index structures [9]; it means that we have to create a very large index for the combinations.

In this paper, we propose a *pattern-based pruning* (PBP) algorithm to solve the combination skyline problem by indexing individual objects rather than combinations in an R-tree. The PBP algorithm searches for skyline combinations with a set of object-selecting patterns organized in a tree that represent the number of objects to be selected in each MBR. We exploit the attribute value ranges in the MBRs as well as search order, and develop two pruning strategies so as to avoid generating a large number of unpromising combinations. We also elaborate how to avoid repeated computations on expanding the same object-selecting patterns to combinations. The efficiency of the PBP algorithm is then evaluated with experiments.

Our contributions can be summarized as follows.

- We propose the combination skyline problem, a new variation of the skyline problem that prevalently exists in daily applications and poses technical challenges.
- We devise a pattern-based pruning algorithm to tackle the major technical issue. The algorithm indexes only individual objects and make combinations with a set of object-selecting patterns. Several optimization strategies are developed to improve the efficiency of the algorithm.
- We discuss two variations of the combination skyline problem – incremental combination skyline and constrained combination skyline, which can be solved by extending the PBP algorithm.
- We conduct extensive experimental evaluations both on synthetic and real datasets to demonstrate the efficiency of the proposed algorithm.

The rest of the paper is organized as follows. Section 2 reviews the studies related to our problem. Section 3 defines the combination skyline problem. Section 4 introduces object selecting patterns and the basic framework of the PBP algorithm. Section 5 proposes two pruning strategies to reduce the search space and an optimization approach to avoid duplicate searches. Section 6 extends the combination skyline problem to two variations and discusses their solutions. Section 7 reports the experimental results and Section 8 concludes the paper.

2 Related Work

In the database field, the skyline problems have received considerable attentions since the seminal paper [2] appeared. A number of subsequent papers propose various of algorithms to improve query performance, like BBS [14], SFS [4] and

LESS [7]. In contrast to these papers, which focus on objects themselves, our problem focuses on combinations. Although we can treat a combination as a normal object, it is time-consuming to find answers using these existing algorithms due to an explosive number of combinations. Among them, BBS is based on an R-tree index on objects. In order to index all of the combinations, a large-sized index is inevitable and results in poor performance.

To the best of our knowledge, there is no literature directly targeting the combination skyline problem. Two closely related topics are "top-k combinatorial skyline queries" [18] and "convex skyline objectsets" [17]. [18] studied how to find top-k optimal combinations according to a given preference order in the attributes. Their solution is to retrieve non-dominated combinations incrementally with respect to the preference until the best k results have been found. This approach relies on the preference order of attributes and the limited number (top-k) of combinations queried. Both the preference order and the top-k limitation may largely reduce the exponential search space for combinations. However, in our problem there is no preference order nor the top-k limitation. Consequently, their approach cannot solve our problem easily and efficiently. Additionally, in practice it is difficult for the system or a user to decide a reasonable preference order. This fact will narrow down the applications of [18].

[17] studied the "convex skyline objectset" problem. It is known that the points on the lower (upper) convex hull, denoted as $\mathcal{CH}$, is a subset of the points on the skyline, denoted as $\mathcal{SKY}$. Every point in $\mathcal{CH}$ can minimize (maximize) a corresponding linear scoring function on attributes, while every point in $\mathcal{SKY}$ can minimize (maximize) a corresponding monotonic scoring function [2]. [17] aims at retrieving the combinations in $\mathcal{CH}$, however, we focuses on retrieving the combinations in $\mathcal{CH} \subseteq \mathcal{SKY}$. Since their approach relies on the properties of the convex hull, it cannot extend easily to solve our problem.

There are some other works [15, 20] focusing on the combination selection problem but related to our work weakly. [15] studied how to select "maximal combinations". A combination is "maximal" if it exceeds the specified constraint by adding any new object. Finally, the k most representative maximal combinations, which contain objects with high diversities, are presented to the user. In their problem, the objects only have one attribute, in contrast to our multiple attribute problem. The approach for single attribute optimization problem is different from the approach for multiple attributes optimization problem. Thus, our problem cannot be solved by simple extensions of their approach.

[20] studies the problem to construct k profitable products from a set of new products that are not dominated by the products in the existing market. They construct non-dominated products by assigning prices to the new products that are not given beforehand like the existing products. Our problem is very different from theirs in two aspects. First, they concern whether a single product is dominated or not, while we concern whether a combination of product is dominated or not. Second, there exist unfixed attribute values (prices) in their problem, while all the attribute values are fixed.

Outside of the database field, the skyline problem is related to the multi-objective optimization (MOO), which has been studied over five decades [5]. Among the variations of the MOO problem, the most relevant to our problem is the *multi-objective combinatorial optimization* (MOCO) problem [6]. The goal is to find subsets of objects aiming at optimizing multiple objective functions subject to a set of constraints. Like the solutions for the MOO problem, most approaches for the MOCO problem essentially convert the multiple objectives to one single objective and find one best answer numerically. Such numerical approaches are not good at handling large scale datasets in databases. Furthermore, our problem aims at retrieving optimal combinations without making a trade-off of multiple objectives by some score functions. For these reasons above, we cannot use the existing MOCO approaches to solve our problem in databases.

This paper is an extended version of [8]. Compared with [8], this paper has the following substantial differences:

- We made modifications to the two pruning techniques, and developed a technique to avoid repeated pattern expansions (Section 5).
- The incremental combination skyline problem, which searches for $(k + \Delta k)$-item skyline combination based on the original k-item skyline combination, is discussed (Section 6.1).
- Attribute constraints, which are contained in the definition of a combination skyline query in [8], are made optional and discussed (Section 6.2).
- The experiments with the optimized PBP algorithm on both real and synthetic datasets were performed (Section 7).

3 Preliminaries

3.1 Problem Definition

Given a set of objects $\mathcal{O}$ with m attributes in the attribute set $\mathcal{A}$, a k-item combination c is made up of k objects selected from $\mathcal{O}$, denoted $c = \{o_1, \ldots, o_k\}$. Each attribute value of c is given by the formula below

$$c.A_j = f_j(o_1.A_j, \ldots, o_k.A_j), \tag{1}$$

where f_j is a monotonic aggregate function that takes k parameters and returns a single value. For the sake of simplicity, in this paper we consider that the monotonic scoring function returns the sum of these values; i.e.,

$$c.A_j = \sum_{i=1}^{k} o_i.A_j, \tag{2}$$

though our algorithms can be applied on any monotonic aggregate function.

Definition 1 (Dominance Relationship) *A combination c dominates another combination c', denoted $c \prec c'$, if c is not larger than c' in all the attributes and is smaller than c' in at least one attribute; formally, $c.A_j \leq c'.A_j$ $(\forall A_j \in \mathcal{A})$ and $c.A_t < c'.A_t$ $(\exists A_t \in \mathcal{A})$.*

Problem 1 (Combination Skyline Problem) *Given a dataset $\mathcal{O}$ and an item number k, the combination skyline problem $CSKY(\mathcal{O}, k)$ is to find the k-item combinations that are not dominated by any other combinations.*

Non-dominated combinations are also called *skyline combinations*. The combination skyline query in Example 1 can be formalized as $CSKY(\{A, \ldots, G\}, 3)$ and the result set is $\{ABC, ABD, BCD\}$. We use the term "cardinality" to denote the item number k if there is no ambiguity. In this paper, we consider the case where $k \geq 2$ because the case where $k = 1$ reduces to the original skyline query [2].

3.2 Baseline Algorithm

In order to solve the combination skyline problem, a naïve approach is to regard the combinations as "objects" and select the optimal ones using existing skyline algorithms. However, these algorithms retrieve optimal objects based on either presorting or indexing objects beforehand. It means that before using such an algorithm we have to enumerate all possible combinations. Due to the explosive number of combinations generated, the naïve approach is inapplicable for large data sets. We choose the BBS algorithm [14] as the baseline algorithm for comparison, and our experiment shows that even for a set of 200 objects and a cardinality of 3, it requires an index nearly one gigabyte and spends thousands of seconds on computing the skyline.

4 Object-Selecting Pattern and Basic PBP Algorithm

Unlike the baseline approach, we propose a *pattern-based pruning* (PBP) algorithm based on an index on single objects rather than an index on combinations. We choose to index objects with an R-tree [9] as it is proven efficient for organizing multi-dimensional data. In order to make combinations, we use a set of *object-selecting patterns* to indicate the number of objects to be selected within each MBR in the R-tree. The object selecting patterns are organized in a *pattern tree*. We search for skyline combinations in the order arranged by a *pattern tree* that corresponds to the R-tree.

4.1 Object-Selecting Pattern

An R-tree is a data structure that hierarchically groups nearby multi-dimensional objects and encloses them by minimum bounding rectangles (MBRs). Our idea is to create combinations by selecting objects from the MBRs. The way is to select k_i objects from each MBR $r_i \in R$ and to make the total number of selected objects equal k. Each k_i is limited in the range of $[0, \min(k, |obj(r_i)|)]$, where $obj(r_i)$ denotes the set of objects enclosed by r_i. An *object-selecting pattern* is defined formally below.

Definition 2 (Object-Selecting Pattern) *Given a cardinality k and a set of MBRs R, an object-selecting pattern p is* $\{ (r_i, k_i) | r_i \in R, k_i \in [0, \min(k, |obj(r_i)|)] \}$ *subject to* $\sum_{i=1}^{|R|} k_i = k$. *In addition, each MBR in R appears exactly once in the pattern p; i.e.,* $\forall r_i$ *and* $r_j, r_i \neq r_j$.

We call the pairs (r_i, k_i) constituting a pattern *rules*. By Definition 2, a *rule* (r_i, k_i) is to select k_i objects from the MBR r_i.

The attribute values of the combinations obtained from a pattern are within $[\sum_{i=1}^{|R|} r_i.A_j^{\perp} \cdot k_i, \sum_{i=1}^{|R|} r_i.A_j^{\top} \cdot k_i]$ $(A_j \in \mathcal{A})$, because we can infer attribute value ranges for the combinations formed using the rule (r_i, k_i) as $[r_i.A_j^{\perp} \cdot k, r_i.A_j^{\top} \cdot k]$ $(A_j \in \mathcal{A})$, where $r_i.A_j^{\perp}$ and $r_i.A_j^{\top}$ are the values of the bottom left and top right corners of r_i.

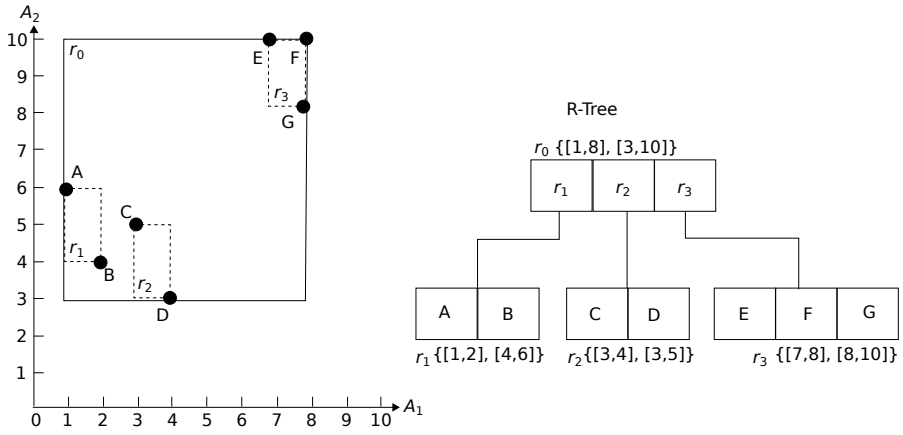


Fig. 2: Object Layout and R-Tree

Example 2 *Fig. 2 shows the R-tree that indexes the objects in Example 1. In order to make 3-item combinations, one of the patterns is* $\{ (r_1, 2), (r_2, 1), (r_3, 0) \}$, *consisting of three rules. Rule* $(r_1, 2)$ *means to select two objects from MBR* r_1, *and rule* $(r_2, 1)$ *means to select one object from MBR* r_2. *Thus, the pattern can generate the set of combinations* $\{ ABC, ABD \}$ *that contains two combinations in total. With the boundaries of the three MBRs, we can limit the attribute values of the generated combinations within* $[5, 8]$ *for* A_1 *and* $[11, 17]$ *for* A_2.

Consider a rule (r, k). If r is a leaf node of the R-tree, we can scan the objects contained and form combinations of size k. If r is an internal node, we need to expand it to child MBRs, and this will yield a group of patterns that select objects from r's child MBRs with the total number of objects summing up to k. We call such patterns the *child patterns* of the rule (r, k).

Definition 3 (Child Patterns of a Rule) *A child pattern of a rule* (r, k) *is a pattern that selects* k *objects from all of* r*'s child MBRs, formally* $cp = \{ (r_i, k_i) | r_i \in R, k_i \in [0, min(k, |obj(r_i)|)] \}$ *subject to* $\sum_{i=1}^{|R|} k_i = k$ *where* R *is the set of the child MBRs of* r*.*

Note that all the child patterns of rule (r, k) share the same set of MBRs, but differ in the quantities of selected objects k_i. In the R-tree shown in Fig. 2, the node r_0 has three child MBRs $\{r_1, r_2, r_3\}$. Thus, patterns $\{(r_1, 2), (r_2, 1), (r_3, 0)\}$, $\{(r_1, 2), (r_2, 0), (r_3, 1)\}$, and so on are the child patterns of the rule $(r_0, 3)$, which share the same set of r_i's but differ in k_i's.

Similarly, a pattern can be expanded to a set of child patterns. For each rule in the pattern, we expand the rule to its child patterns, and perform an n-ary Cartesian product on all these child patterns. Algorithm 1 presents the pseudo-code of the procedure.

Algorithm 1: ExpandPattern (p)

Input : A pattern p represented in a set of (r_i, k_i)'s.
Output : The set of child patterns of p.

```
1 P ← e;
  /* assume e is the identity element of Cartesian product          */
2 for each (r_i, k_i) ∈ p do
3 |   P' ← the child patterns of (r_i, k_i);
4 |   P ← P × P';
5 end for
6 return P
```

Starting with the root node r_0 in the R-tree and its corresponding root pattern $p_0 = \{ (r_0, k) \}$, if we traverse the R-tree with a breadth-first search, and expand each corresponding pattern using its child patterns, we can obtain all possible combinations at the leaf level. Accordingly, the patterns expanded constitute a *pattern tree*. Example 3 shows the procedure of constructing a pattern tree with respect to the R-tree in Fig. 2.

Example 3 *A pattern tree corresponding to the R-tree in Fig. 2 is shown in Fig. 3. The root pattern is* $p_0 = \{ (r_0, 3) \}$ *where* 3 *is the required cardinality. Since pattern* p_0 *only has a single rule* $(r_0, 3)$*, the eight child patterns of* $(r_0, 3)$*,* , $\{p_1, \ldots, p_8\}$*, are also the child patterns of* p_0*. Next, we expand the patterns at the second level of the pattern tree. Consider pattern* $p_1 = \{ (r_1, 2), (r_2, 1), (r_3, 0) \}$ *that contains three rules* $(r_1, 2)$*,* $(r_2, 1)$ *and* $(r_3, 0)$*. Rule* $(r_1, 2)$ *has one child pattern* $\{AB\}$ *and rule* $(r_2, 1)$ *has two child patterns* $\{C, D\}$ *and hence the child patterns of* p_1 *is* $\{AB\} \times \{C, D\} = \{ABC, ABD\}$*. Since these child patterns contain objects rather than MBRs, we also call them child combinations.*

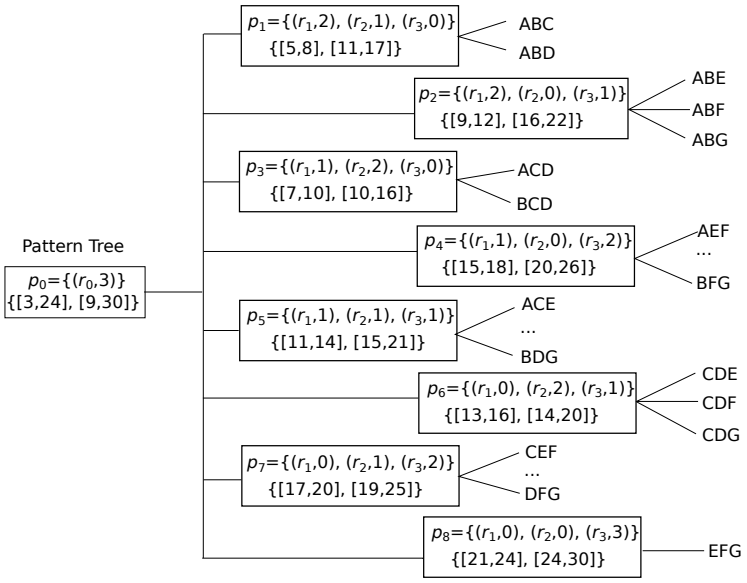


Fig. 3: Pattern Tree

4.2 Basic PBP Algorithm

Following the pattern tree, we design a basic PBP algorithm (Algorithm 2). It takes as an input the set of objects, and first builds an R-tree on the objects. Starting with the root node r_0 and the pattern $\{(r_0, k)\}$, we traverse the R-tree in a top-down fashion. Note that the pattern tree is not materialized in the algorithm. Instead, we use a queue Q to capture the patterns generated while traversing the pattern tree. Each pattern is expanded to its child patterns (Line 7) if the nodes in the pattern are internal nodes; otherwise leaf nodes are reached and hence we can make combinations in the MBRs according to the pattern (Line 11). The combinations are then checked for dominance relationship with the candidate skyline combination found so far and vice versa (Line 12). The candidates not dominated by any combinations are returned as the answer after processing all the expanded patterns.

Compared with the baseline algorithm, the basic PBP algorithm reduces the space consumption by building an R-tree on single objects. However, it suffers from the huge number of patterns. Even for a rule (r, k), the number of child patterns is $\binom{h+k-1}{h-1}$ if r has h child MBRs. We will discuss how to reduce this number and consider only promising child patterns in the following section.

5 Optimizations of PBP Algorithm

In a pattern tree, we can decide which patterns should be expanded and which patterns should not be expanded. For example, in the pattern tree shown in

Algorithm 2: BasicPBP (T, k)

Input : T is the R-tree built on $\mathcal{O}$; k is the cardinality.
Output : The skyline combination set $S = CSKY(\mathcal{O}, k)$.

```
1  S ← ∅;
2  r_0 ← the root node of T;
3  Q ← { (r_0, k) };
4  while Q ≠ ∅ do
5      p ← Q.pop();
6      if the MBRs in p are internal nodes then
7          P ← ExpandPattern(p);
8          for each p' ∈ P do
9              Q.push(p');
10     else
11         C ← generate combinations with p;
12         S ← Skyline(S ∪ C);
13 return S
```

Fig. 3, the combinations following pattern p_4 must be dominated by the combinations following pattern p_1. Thus, we can prune pattern p_4 without further expanding. Another intuition is that if the combinations from a pattern are guaranteed to be dominated by the current skyline combinations, the pattern can be pruned as well. We call these two scenarios *pattern-pattern pruning* and *pattern-combination pruning*. We also observe the existence of multiple expansions for same patterns in the pattern tree. In the rest of this section, we will study the two pruning techniques and how to avoid multiple expansions as well.

5.1 Pattern-Pattern Pruning

Patterns can be pruned safely without expanding if they will generate combinations that are guaranteed to be dominated by others. We first define the dominance relationship between patterns and capture the idea in Theorem 1.

Definition 4 (Pattern Dominance) *A pattern* p dominates *another pattern* p' *if* $p.A_j^{\top} \leq p'.A_j^{\perp}$ $(\forall A_j \in \mathcal{A})$ *and* $p.A_t^{\top} < p'.A_t^{\perp}$ $(\exists A_t \in \mathcal{A})$*, and is denoted as* $p \prec p'$.

Theorem 1 *A pattern* p' *cannot generate skyline combinations if it is dominated by another pattern* p.

Proof. Any combination c' following the pattern p' has values $c'.A_j \geq p'.A_j^{\perp}$ $(\forall A_j \in \mathcal{A})$. Any combination c following the pattern p has values $c.A_j \leq p.A_j^{\top}$ $(\forall A_j \in \mathcal{A})$. If $p \prec p'$, $c.A_j \leq c'.A_j$ $(\forall A_j \in \mathcal{A})$ and $c.A_t < c'.A_t$ $(\exists A_t \in \mathcal{A})$. Consequently, c' is not a skyline combination because $c \prec c'$.

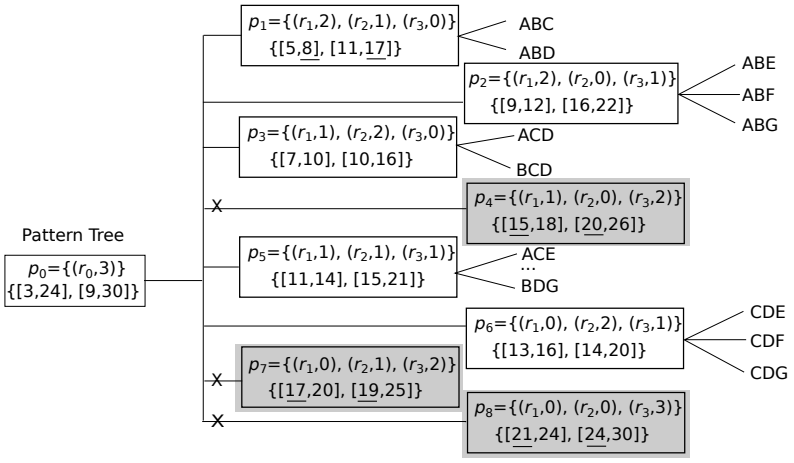


Fig. 4: Pattern-Pattern Pruning (Grey patterns are pruned using Theorem 1)

Example 4 *Consider the eight patterns* $\{p_1, \ldots, p_8\}$ *at the second level of the pattern tree shown in Fig. 4. Pattern* p_1 *with upper bounds* $(8, 17)$ *can dominate pattern* p_4 *with lower bounds* $(15, 20)$*,* p_7 *with lower bounds* $(17, 19)$*, and pattern* p_8 *with lower bounds* $(21, 24)$*. Thus, the three patterns* p_4*,* p_7 *and* p_8 *can be safely pruned according to Theorem 1.*

5.2 Pattern-Combination Pruning

Starting with the root pattern, we expand patterns to child patterns until obtaining combinations at the leaf level. Unlike BasicPBP in Algorithm 2 that traverses patterns in a breadth-first way, we can use a priority queue to implement the expansion process in a key-order way. Inspired by the BBS algorithm [14], the keys for the priority queue are the *mindists* of the patterns, and we process the patterns in the priority queue following the increasing order of their keys.

Definition 5 (Mindist of a Pattern) *The mindist* p*, denoted as* $p.mindist$*, is the sum of its lower bounds in all the attributes* $\mathcal{A}$*, namely,* $p.mindist = \sum_{j=1}^{|\mathcal{A}|} p.A_j^{\perp}$ $(A_j \in \mathcal{A})$.

Like BBS, we also insert the generated combinations to a priority queue. In the same way, the mindist of a combination can be defined as the sum of values in $\mathcal{A}$, namely, $b.mindist = \sum_{j=1}^{|\mathcal{A}|} b.A_j$ $(A_j \in \mathcal{A})$.

Theorem 2 *A combination* c *cannot be dominated by any combinations generated from a pattern* p' *if* $c.mindist < p'.mindist$*.*

Proof. Assume that the combination c can be dominated by c' which is generated from p'. According to Definition 1, $c'.A_j \leq c.A_j$ $(\forall A_j \in \mathcal{A})$ and $c'.A_t < c.A_t$ $(\exists A_t \in \mathcal{A})$. It means that $c'.mindist < c.mindist$ because $c'.mindist =$

$\sum_{j=1}^{|\mathcal{A}|} c'.A_j$ and $c.mindist = \sum_{i=1}^{|\mathcal{A}|} c.A_j$. On the other hand, $p'.mindist \leq c'.mindist$ because $\sum_{j=1}^{|\mathcal{A}|} p'.A_j^{\perp} \leq \sum_{i=1}^{|\mathcal{A}|} c'.A_j$. Consequently, the inequality $p'.mindist < c.mindist$ contradicts the condition $c.mindist < p'.mindist$, and thus Theorem 2 is proved.

The advantage of expanding patterns using a *mindist*-order priority queue is that when the top element is a combination, according to Theorem 2, it cannot be dominated by the combinations following the patterns behind it in the queue. It just needs comparisons with the skyline combinations already found in the result set $S = CSKY(\mathcal{O}, k)$. If it cannot be dominated by any combinations in S, it is a skyline combination and should be added into S. For the other case where the top element is a pattern, it should be discarded if it is dominated by any combinations in S; otherwise, it should be expanded and its child patterns are pushed into the queue. The above process begins with the root pattern pushed into the queue and ends when the queue is empty. The final S is returned as the answers. Example 5 illustrates the process.

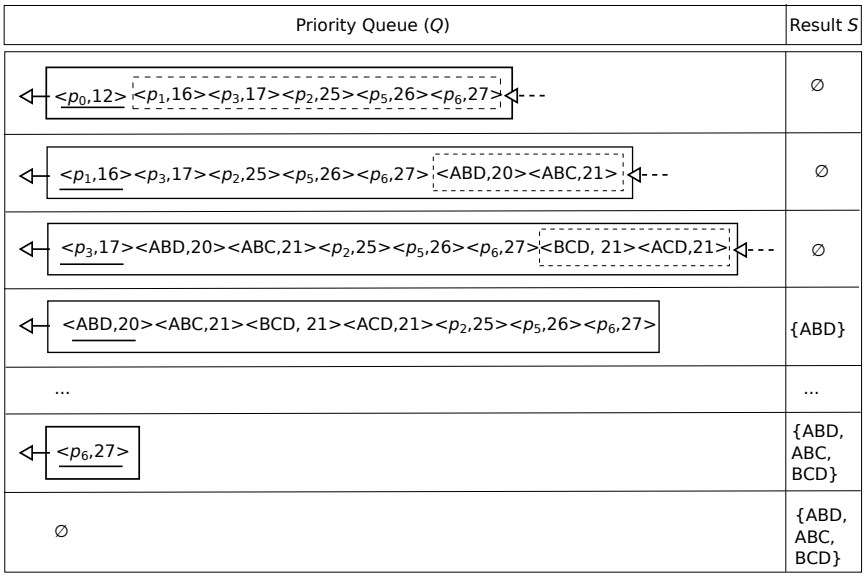


Fig. 5: Priority Queue and Query Result

Example 5 *Fig. 5 shows the process of the combination skyline queries. We initialize the priority queue Q as $\{\langle p_0, 12\rangle\}$ where p_0 is the root pattern and 12 ($p_0.mindist$) is the key. Next, p_0 is popped and its child patterns $\{p_1, p_3, p_2, p_5, p_6\}$ are pushed into Q. Note that other three patterns are pruned according to Theorem 1. We pop the top one p_1 and push its expansions $\{ABD, ABC\}$ into Q. For*

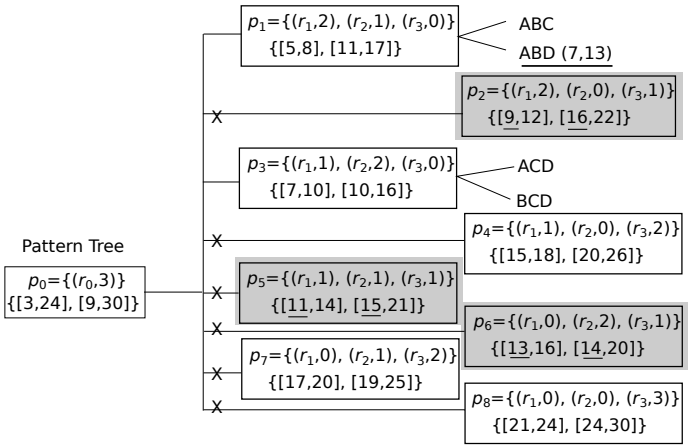


Fig. 6: Pattern-Combination Pruning (Grey patterns are pruned using Theorem 2 and the patterns beginning with $\times$ are pruned using both Theorem 1 and Theorem 2)

the next top element pattern p_3, we pop it and push its expansions $\{BCD, ACD\}$ into Q. Next, the top element is the combination ABD, which is popped and becomes the first result in $S = CSKY(\mathcal{O}, k)$. In the same way, we pop the top element and check whether it is dominated by the skyline combinations in S. If it is dominated, top element is discarded. Otherwise, its child patterns are pushed into Q. For example, when p_6 becomes the top element, it is dominated by $ABD \in S$. Thus, it is discarded. The process continues until the queue Q is empty and we obtain the final result set $\{ABD, ABC, BCD\}$. Fig. 6 shows the pattern tree after the pattern-combination pruning.

5.3 Pattern Expansion Reduction

Another problem with BasicPBP algorithm is that the same rules may appear in multiple patterns and thus may be expanded multiple times. In Fig. 3, among the child patterns expanded from the root pattern p_0, patterns p_1 and p_2 share the same rule $(r_1, 2)$, and it will be expanded twice into the same set of child patterns.

Duplicate expansion is even worse as the algorithm goes deeper in the R-tree. An immediate solution to address this problem is to perform a lazy expansion if a rule is encountered multiple times. The intuition is that once the descendant patterns of the first occurrence reach the object combination level, the generated combinations are kept, and all the multiple occurrences of the rule can be replaced by the combinations when a dominance check is invoked. In order to keep the combinations for each rule encountered, we use a matrix M with MBRs as rows and cardinalities columns.

As the search order shown in Fig. 5, pattern p_1 comes before pattern p_2 in the priority queue. p_2's child patterns will inherit the rule $(r_1, 2)$ from p_2, but not

expand the rule immediately. After all the descendant patterns of p_1 have been processed to create object combinations, the cells representing $(r_1, 2)$ and its descendants are filled with the combinations. When p_2 is expanded and reaches the object level, its component rule $(r_1, 2)$ is replaced by what we stored in the cell for dominance checking.[4]

The above solution ensures no duplicate expansion of a rule in the algorithm. However, it is not space-efficient to record the all the combinations for the rules encountered. Thanks to the following theorem, we are able to store only the *skyline* combinations instead for each cell in the matrix.

Theorem 3 *If a skyline combination $c \in CSKY(\mathcal{O}, k)$ contains k' objects in an MBR r', the combination consisting of the k' objects is a skyline combination of $obj(r')$ with cardinality k'.*

Proof. Consider a skyline combination $c \in CSKY(\mathcal{O}, k)$ that contains k' objects in an MBR r'. Assume the k' objects are $o_1, \ldots, o_{k'}$, and their combination is dominated by another combination $\{ o'_1, \ldots, o'_{k'} \}$ whose objects are also enclosed by r'. According to the monotonicity of the aggregate function,

$$c \backslash \{ o_1, \ldots, o_{k'} \} \cup \{ o'_1, \ldots, o'_{k'} \} \prec c.$$

It contradicts the assumption that c is a skyline combination of $\mathcal{O}$ with cardinality k, and hence the theorem is proved.

Therefore, we only need to keep $CSKY(obj(r_i), k_i)$ for each $M[r_i][k_i]$. For a rule with a leaf MBR, we compute it with the objects inside. For a rule with an internal MBR, we compute $M[r_i][k_i]$ once all of its child patterns have been expanded to the object level, and store the skyline over the results obtained from the child patterns. Note that this skyline computation is a byproduct of generating combinations of size k and checking dominance, and thus we do not need to compute them separately. Example 6 shows the process of filling in the matrix M.

Example 6 *According to the search order shown in Fig. 5, when expanding p_0 we fill $M[r_0][3]$ using the child patterns of $(r_0, 3)$. Next, we expand p_1 containing rules $(r_1, 2)$, $(r_2, 1)$, and $(r_3, 0)$. The corresponding cells $M[r_1][2]$ is filled with the combination $\{AB\}$ and $M[r_2][1]$ is filled with combinations $\{C, D\}$ that are not dominated each other. The Cartesian join products $\{AB\} \times \{C, D\} = \{ABC, ABD\}$ are the child patterns of p_1. The next pattern expanded is p_3 containing rules $(r_1, 1)$, $(r_2, 2)$, and $(r_3, 0)$. The corresponding cells $M[r_1][1] = \{A, B\}$ and $M[r_2][2] = \{CD\}$. The join products $\{A, B\} \times \{CD\} = \{ACD, BCD\}$ are the child patterns of p_3.*

[4] We assume the descendants of p_1 come before those of p_2 in the priority queue in this example. For the general case, once a descendant of $(r_1, 2)$ has produced object combinations, other patterns that contain the rule can avoid redundant computations.

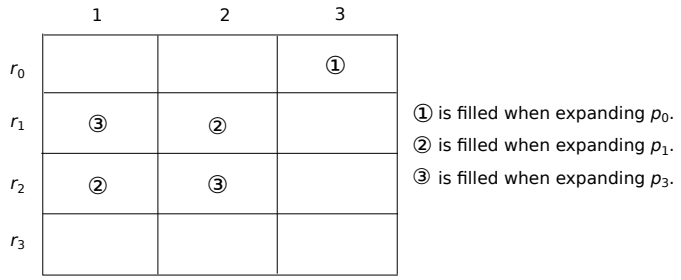


Fig. 7: Pattern Expansion Reduction Matrix

Since pattern-pattern pruning keeps unpromising patterns from the priority queue, not all the cells in the matrix need to be filled. Considering the sparsity of the matrix, we implement it with a hash table with an (MBR, cardinality) pair as the key for each entry, and store the value as

- a set of skyline combinations, if all its child patterns have been expanded to the object level; and
- a list of its child patterns, otherwise.

We design a new pattern expansion algorithm in Algorithm 3. It expands a rule under three different cases. If the rule is encountered for the first time, i.e., the cell in the matrix has not been initialized, we expand it to its child patterns, and fill the cell in the matrix with a list of the child patterns (Line 5 and 6). If the rule is encountered multiple times, but none of the patterns contains it have reached the object level so far, we expand the rule with the stored list of child patterns (Line 9). For the third case, as the object combinations for this rule have been seen before, we keep the rule intact until the patterns containing it reach the object level, and then it is replaced with the skyline combinations stored in the cell (Line 12).

5.4 PBP Algorithm

Applying the three optimization techniques, we summarize the complete pattern-based pruning (CompletePBP) algorithm in Algorithm 4. The algorithm iteratively pops the top element p in the priority queue Q (Line 5). The top element can be either a combination or a pattern. For a combination, we insert them into the final result set S after checking dominance with the current skyline combinations (Line 8). For a pattern, if p is dominated by any skyline combinations found so far, we discard it using with pattern-combination pruning (Line 10). Otherwise we expand it using the optimized pattern expansion algorithm (Line 13). If the MBRs involved in P are internal nodes of the R-tree, we push the non-dominated child patterns to the queue Q (Line 16), utilizing pattern-pattern pruning. Otherwise we generate combinations with the patterns in P, and update the matrix M to reduce pattern expansion (Line 19– 26). The algorithm terminates when the priority queue is empty.

Algorithm 3: ExpandPatternOpt (p)

```
   Input  : A pattern p represented in a set of (r_i, k_i)'s.
   Output : The set of child patterns of p.
 1 P ← e;
   /* assume e is the identity element of Cartesian product          */
 2 for each (r_i, k_i) ∈ p do
 3     switch M[r_i][k_i] do
 4         case has not been initialized do
 5             P' ← the child patterns of (r_i, k_i);
 6             M[r_i][k_i] ← P';
 7         end case
 8         case is a list of child patterns do
 9             P' ← M[r_i][k_i];
10         end case
11         case is a set of skyline combinations do
12             P' ← { (r_i, k_i) };
               /* (r_i, k_i) has been explored and replace with M[r_i][k_i] when
                  reaching the object level                            */
13         end case
14     endsw
15     P ← P × P';
16 end for
17 return P
```

6 Variations of Combination Skyline

In this section, we discuss two variations of the combination skyline problem and extend our PBP algorithm to solve the two variations.

6.1 Incremental Combination Skyline

We first discuss the incremental combination skyline problem, as a user may want to increase the cardinality of combinations as he has seen the result of $CSKY(\mathcal{O}, k)$. The problem is defined as follows:

Problem 2 (Incremental Combination Skyline Query) *An incremental combination skyline query $CSKY^+(\mathcal{O}, k + \Delta k)$ is to find $(k + \Delta k)$-item skyline combinations based on an original query $CSKY(\mathcal{O}, k)$ that has been answered already.*

The incremental query $CSKY^+$ searches for skyline combinations from the same dataset $\mathcal{O}$ as the original query $CSKY$, so we can use the same R-tree for the original skyline query. Starting with the root $(r_0, k + \Delta k)$, the patterns are processed using the PBP algorithm. As the matrix M for duplicate expansion reduction has been filled when processing the original query, if not all of its cells, the contents can be utilized. When the child patterns of rule (r_i, k_i) are

Algorithm 4: CompletePBP $(T,\ k)$

Input : T is the R-tree built on $\mathcal{O}$; k is the cardinality.
Output : The skyline combination set $S = CSKY(\mathcal{O}, k)$.

```
1  S ← ∅;
2  r_0 ← the root node of T;
3  Q ← { (r_0, k) }; M ← ∅;
4  while Q ≠ ∅ do
5      p ← Q.pop();
6      if p is a combination then
7          if ∄c ∈ S, c ≺ p then
8              S ← S ∪ { p };
9      else
10         if ∃c ∈ S, c ≺ p then
11             continue;
12         if the MBRs in p are internal nodes then
13             P ← ExpandPatternOpt (p);
14             for each p' ∈ P do
15                 if ∄p'' ∈ P, p'' ≺ p' then
16                     Q.push(⟨p', p'.mindist⟩);
17         else
18             C ← e;
               /* assume e is the identity element of Cartesian product
                  */
19             for each (r_i, k_i) ∈ p do
20                 if (r_i, k_i) is a set of skyline combinations then
21                     C' ← M[r_i][k_i];
22                 else
23                     C' ← CSKY(obj(r_i), k_i);
                       /* use only CSKY for the rule to generate
                          combinations                                  */
24                     C ← C × C';
25                     M[r_i][k_i] ← C';
26                     update (r_i, k_i)'s ancestor rules in M;
27             for each c ∈ C do
28                 Q.push(c, c.mindist);
29 return S
```

needed during expansion, we reuse the existing results in $M[r_i][k_i]$ if the cell was calculated already. In this way, the repeated calculations for the same cell can be avoided. Though Δk empty columns are appended to M at first, this is for free as M is implemented in a hash table.

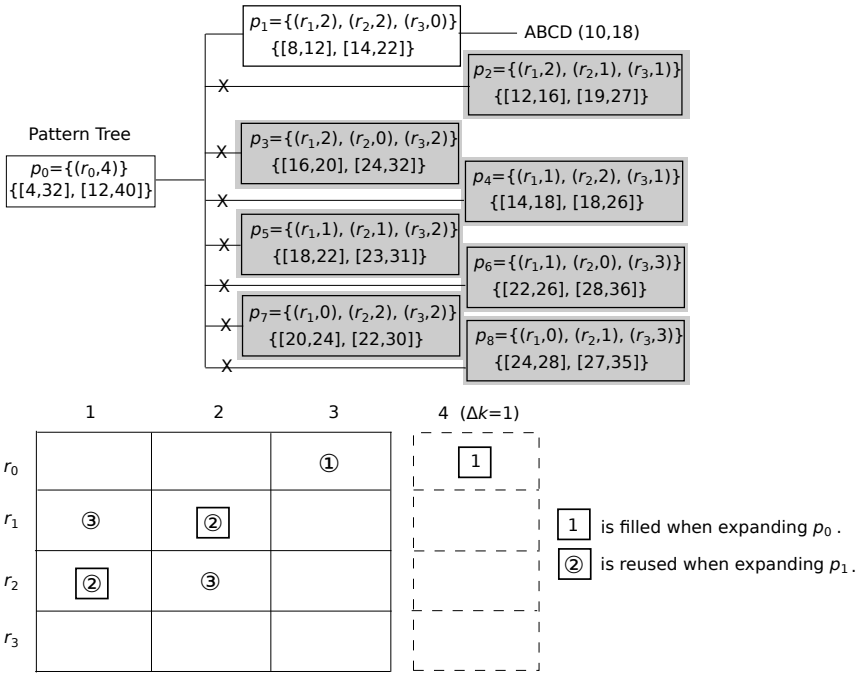


Fig. 8: Incremental Combination Skyline Query

Example 7 *Fig. 8 shows the pattern tree and the matrix M for the incremental query $CSKY^+(\mathcal{O}, 4)$ based on the original query $CSKY(\mathcal{O}, 3)$ with $\Delta k = 1$. The circled numbers in the matrix indicate what we have processed when processing $CSKY(\mathcal{O}, 3)$, and the quads indicate what we are going to fill for processing the incremental query. We start with expanding the root pattern $p_0 = \{ (r_0, 4) \}$, and an empty column is appended to M. Pattern p_0 has three child patterns that survive pattern-pattern pruning: p_1, p_2, and p_4, sorted by the increasing mindist order. Next we expand pattern p_1 consisting of three rules $(r_1, 2)$, $(r_2, 2)$, and $(r_3, 0)$. Both cells $M[r_1][2]$ and $M[r_2][2]$ were calculated already when answering the original query. By computing the Cartesian product, a combination $ABCD$ is obtained for p_1. Since $ABCD$ is the first combination found, it is a skyline combination and we put it into the result set. As the next top elements p_2 and p_4 are dominated by the combination $ABCD$, the process terminates when the queue is empty and the final result $CSKY^+ = \{ ABCD \}$ is returned.*

6.2 Constrained Combination Skyline

For a combination skyline query, we search for optimal combinations that have values as small as possible with respect to all the attributes $\forall A_j \in \mathcal{A}$. In practice, however, not all the attributes are being concerned and there are even some range constraints on the concerned attributes. We define a *constrained combination skyline query* that searches for optimal combinations with respect to a set of concerned attributes $\mathcal{A}^* \subseteq \mathcal{A}$ subject to range constraints V_j on attribute $A_j \in \mathcal{A}^*$.

Problem 3 (Constrained Combination Skyline Query) *A constrained combination skyline query $CSKY^*$ is defined as*

$$CSKY^* = \{\mathcal{O}, k, \langle A_1, V_1\rangle, \ldots, \langle A_{m^*}, V_{m^*}\rangle\}, \tag{3}$$

where $1 \leq m^* \leq m$ and $\{A_1, \ldots, A_{m^*}\} \subseteq \mathcal{A}$. We call $\mathcal{A}^* = \{A_1, \ldots, A_{m^*}\}$ constraint attributes. $V_j = [v_j^{\perp}, v_j^{\top}]$ is a range constraint for attribute A_j ($A_j \in \mathcal{A}^*$). If we do not specify a range constraint on attribute A_j, we set an infinite range $V_j = [-\infty, \infty]$.

The combination skyline query defined in Problem 1 is subsumed in the constrained combination skyline query $CSKY^*$ because $CSKY$ is a special case of $CSKY^*$ when $\mathcal{A}^* = \mathcal{A}$, and subject to $V_j = [-\infty, \infty]$ ($\forall A_j \in \mathcal{A}^*$).

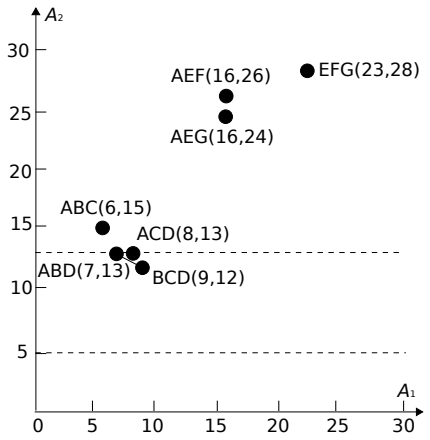


Fig. 9: Constrained Combination Skyline

Example 8 *Let us consider an example of a constrained combination skyline query, $CSKY^*(\{A, \ldots, G\}, 3, \langle A_1, [-\infty, \infty]\rangle, \langle A_2, [5, 13]\rangle)$. As Fig. 9 shows, since the combinations $\{ABD, ACD, BCD\}$ are within the range $[5, 13]$ on A_2, they are candidates for skyline combinations. Among the three candidates, combination ACD is dominated by combination ABD. Thus, the non-dominated combinations $\{ABD, BCD\}$ are the skyline combinations for query $CSKY^*$.*

Definition 6 (Feasible Combination) *A combination c is feasible if it has valid values in all the attributes $\forall A_j \in \mathcal{A}^*$, namely, $c.A_j \in [v_j^{\perp}, v_j^{\top}],\ \forall A_j \in \mathcal{A}^*$.*

The patterns can be discarded if they cannot generate feasible combinations.

Theorem 4 *A pattern p cannot generate feasible combinations if $[p.A_t^{\perp}, p.A_t^{\top}] \cap [v.A_t^{\perp}, v.A_t^{\top}] = \emptyset$ $(\exists A_t \in \mathcal{A}^*)$, where $[v.A_t^{\perp}, v.A_t^{\top}]$ is the valid range of values in attribute A_t.*

Proof. Any combination c following the pattern p has the value $c.A_t \in [p.A_t^{\perp}, p.A_t^{\top}]$ for attribute $A_t \in \mathcal{A}^*$. If $[p.A_t^{\perp}, p.A_t^{\top}] \cap [v.A_t^{\perp}, v.A_t^{\top}] = \emptyset$, $c.A_t \notin [v.A_t^{\perp}, v.A_t^{\top}]$. Consequently, combination c is not a feasible combination.

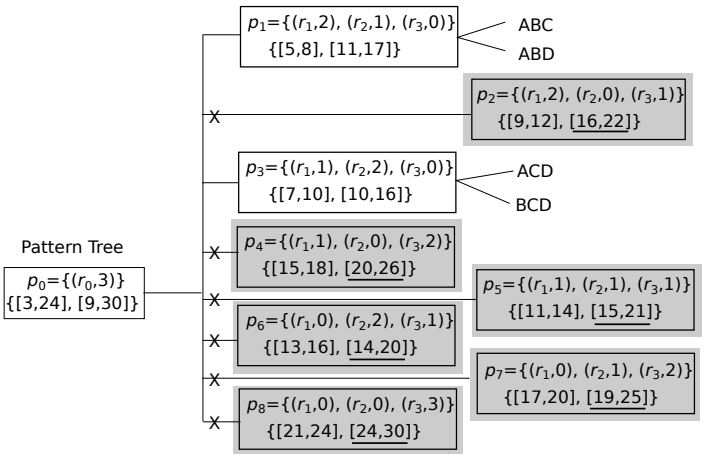


Fig. 10: Constraint-Based Pruning

Example 9 *Fig. 10 shows the pattern tree for the constrained combination skyline query $CSKY^*$. According to Theorem 4, patterns $\{p_2, p_4, p_5, p_6, p_7, p_8\}$ can be pruned with respect to the constraint $[5, 13]$ on attribute A_2 because their ranges on attribute A_2 are out of the range constraint $[5, 13]$. Thus, we only need to expand $\{p_1, p_3\}$ for query $CSKY^*$ and obtain the final result $S = \{ABD, BCD\}$.*

Given a pattern p, the sets of MBRs appearing in its child patterns are the same, and thus only the values of k_i's need to be assigned. We can avoid enumerating useless ones by employing the forward checking approach, which is a common solution for constraint satisfaction problems [1] and used for answering spatial database queries [13]. At first, the possible value of each variable k_i is in the range of $[0, \min(|obj(r_i)|, k)]$, and then we assign values from k_1. Once a k_i has been assigned, the ranges of the remaining variables may shrink due to the attribute constraints, and the new ranges can be determined using Theorem 4.

Example 9 (continued) *If k_1 is set as 1, we use forward checking to update the value ranges of k_2 and k_3. The range of k_2 will be $[0, 2]$, and the range of k_3 will be $[0, 1]$. For example, if $k_3 = 2$, then the value of the combination on A_2 is at least $4 + 8 + 8 = 20$, which violates the constraint $[5, 13]$.*

7 Experiments

In this section, we report experimental results and our analyses.

7.1 Experimental Setup

We used both synthetic and real datasets in our experiment. We generated synthetic dataset using the approach introduced in [2] with various correlation coefficients, and we used the uniform distribution as default unless otherwise stated. For real dataset, we used the NBA dataset[5] which contains the statistics about 16,739 players from 1991 to 2005. The NBA dataset roughly follows an anti-correlated distribution. The default cardinality and the number of dimensions are both two.

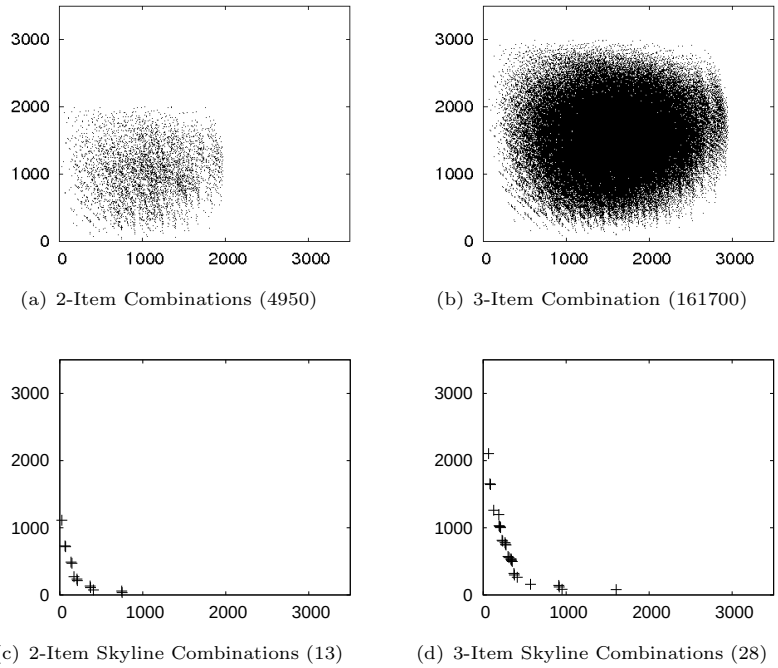
We compare our complete PBP algorithm with the baseline BBS algorithm. Since BBS cannot handle the explosive number of combinations when the dataset is large, we only compare PBP and BBS on small synthetic dataset. Both PBP and BBS were implemented in C++. The R-tree structure was provided by the spatial index library SaIL [10]. All the experiments were conducted on a Quad-Core AMD Opteron 8378 with 96 GB RAM. The operating system is Ubuntu 4.4.3. All the data structures and the algorithms were loaded into/run in main memory.

7.2 Experiments on Synthetic Datasets

Fig. 11(a) and 11(b) show the distributions of 2-item combinations and 3-item combinations, which are generated from a dataset containing 100 objects with two-dimensional attributes uniformly distributed in the range $[0, 1000] \times [0, 1000]$. In total, there are $\binom{100}{2} = 4950$ combinations and $\binom{100}{3} = 161700$ combinations shown as points in the two figures. The numbers of skyline combinations are much smaller; e.g., 13 from the 4950 2-item combinations and 28 from the 161700 combinations, as shown in the areas close to the horizontal axis and the vertical axis in Fig. 11(c) and 11(d).

Next, we compare our PBP algorithm with the BBS algorithm, and then study the efficiency of the PBP algorithm with respect to data distribution, cardinality, the number of attributes (dimensionality), and the fanout of R-tree.

[5] `http://www.nba.com/`



(a) 2-Item Combinations (4950)

(b) 3-Item Combination (161700)

(c) 2-Item Skyline Combinations (13)

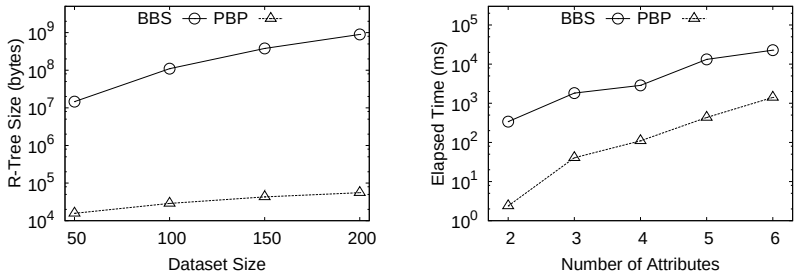
(d) 3-Item Skyline Combinations (28)

Fig. 11: Distribution of Combinations and Skyline Combinations

Comparison with the BBS algorithm Since BBS cannot find skyline combinations from large datasets in acceptable response time, we compare the performances of BBS and PBP on small datasets that contain 50, 100, 150, 200 objects. For every data size, we vary the number of attributes in the range of [2, 6]. The experimental query is to search for three-item skyline combinations.

Fig. 12(a) shows the size of R-trees used by BBS and PBP. For BBS, the R-tree sizes grows dramatically with the data size because R-trees have to index all the combinations that increases in an explosive way. As the figure shows, when the dataset contains 200 objects, the tree size is almost one gigabyte. Even worse, constructing such a huge R-tree consumes a lot of time, which means that BBS cannot work well in practice. In contrast, PBP uses the R-tree for indexing single objects rather than combinations, which makes the tree size grow relatively slow. This is also why PBP can handle large datasets as what will be shown in Section 7.2 to Section 7.3.

Fig. 12(b) shows the running time of BBS and PBP on the 100-object datasets, with the number of attributes varying from 2 to 6. For BBS, the time is the sum of the time for enumerating combinations and the time consumed by searching for skyline combinations. For PBP, the time is the time for searching for skyline combinations. The time for constructing R-trees is not included. As the figure shows, PBP outperforms BBS by at least one order of magnitude. One



(a) $k = 3,\ |\mathcal{A}| = 2$ (b) $k = 3$

Fig. 12: PBP versus BBS on Small Datasets

reason is that PBP executes queries on the R-tree that is far smaller than the R-tree used by BBS. Another reason is that the time for enumerating combinations is saved when running PBP.

The Effect of Data Distribution We evaluate PBP on 4K, 8K, 16K, 32K, 64K datasets with different correlation coefficients -0.9, -0.6, -0.3, 0.0, 0.3, 0.6 and 0.9. The datasets with correlation coefficients -0.9, -0.6 and -0.3 follow anti-correlated distributions. The datasets with correlation coefficients 0.9, 0.6 and 0.3 follow correlated distributions. The dataset with correlation coefficient 0.0 follows uniform distributions. Each dataset has objects with two attributes. The queries are to select five-item skyline combinations from these datasets.

Fig. 13(a) shows the number of skyline combinations and Fig. 13(b) shows the running time. As Fig. 13(a) shows, there are more skyline combinations for the anti-correlated datasets and fewer skyline combinations for the correlated datasets. In the anti-correlated datasets, some objects are good in one attribute but are bad in the other attribute. In the correlated datasets, a part of the objects are good in both attributes. It can be seen that the there are more skyline combinations generated from the anti-correlated datasets than from the correlated datasets. This is because the combinations exhibit distribution features as single objects since their attribute values are the sums of their component objects' attribute values.

Fig. 13(b) shows the running time of PBP. It spends much time when running PBP on the anti-correlated datasets than on the correlated datasets. The time depends on the size of the priority queue and the number of dominance checks. Fig. 13(c) and 13(d) show the maximum size of the priority queue and the number of dominance checks, respectively. Since the patterns also follow the same distributes, there are more patterns which cannot be pruned and have to be pushed into the priority queue for the anti-correlated datasets. Consequently, more dominance checks occur.

Another observation is that running time does not vary significantly with the sizes of datasets. The reason is that the performance of PBP is not sensitive

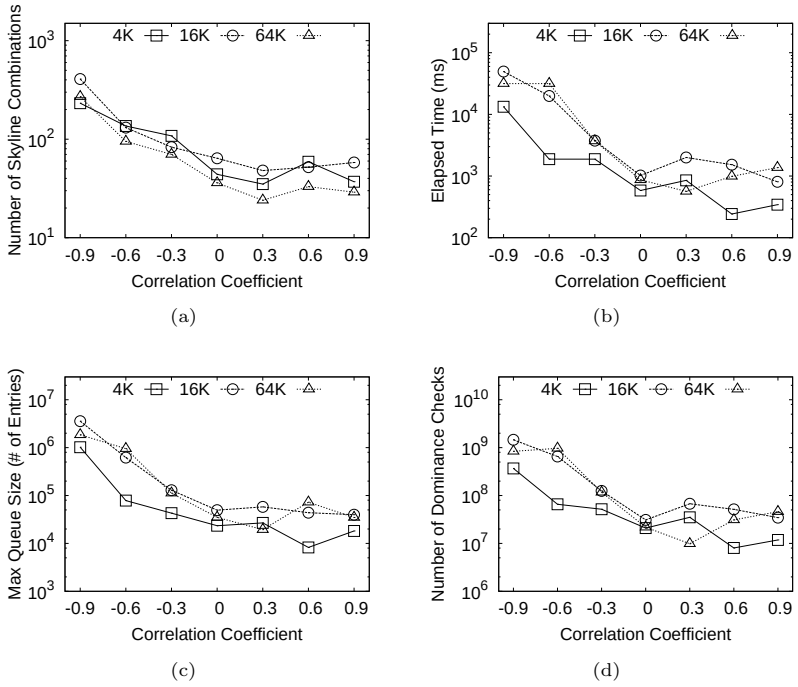


Fig. 13: PBP Performance for Different Distributions

to the data sizes for low dimension cases, as can be seen from the size of the priority queue and the number of dominance checks.

The Effect of Cardinality We run PBP on 8K, 16K, 32K, 64K, and 128K datasets to search for skyline combinations of cardinalities $k \in [3, 6]$. The objects in the datasets have two attributes.

Fig. 14(a) shows the number of skyline combinations. The number increases with the cardinality but not in an explosive way as does the explosive number of combinations. The reason is much more combinations can be dominated for larger cardinalities. As Fig. 11 shows, more combinations are dominated by the skyline combinations when the cardinality increase from two to three.

Fig. 14(b) shows the running time of PBP. The time increases with the cardinality. It depends on the maximum size of the queue and the number of dominance checks, which are shown in Fig. 14(c) and 14(d), respectively. When the cardinality enlarges, the number of patterns increases. Thus, more patterns are pushed into the queue and more dominance checks are needed. Another general trend is that the running time increases with dataset sizes, but the influence is not as significant as that of cardinality. Considering the number of combination $\binom{|\mathcal{O}|}{k}$, it grows faster with the increase of k than with that of $|\mathcal{O}|$.

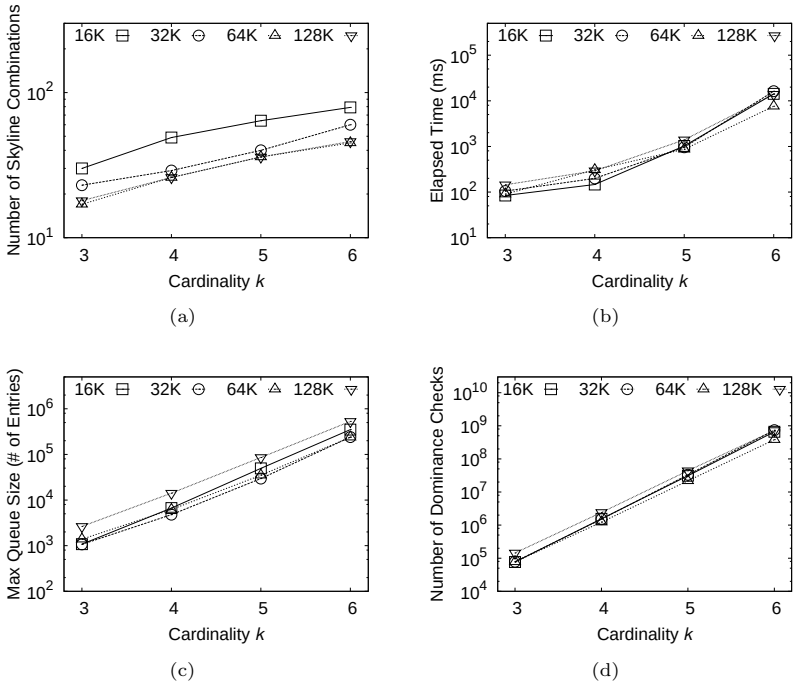


Fig. 14: PBP Performance for Different Cardinalities

The Effect of Dimensionality We evaluate the effect of dimensionality by varying the number of attributes in the range of [2, 6]. For each dimensionality, we run PBP on 1K, 2K, 4K, 8K, 16K datasets, respectively. The query is to search for 2-item combinations from these datasets.

Fig. 15(a) shows the number of skyline combinations. The number exhibits a rapid growth with the dimension. The reason is that when the dimension increases, it is more likely that two combinations are better than each other in different subsets of the dimensions. Thus, one cannot dominate another and the number of skyline combinations increases. It is also called the curse of dimensionality [3].

Fig. 15(b) shows the running time. The time increases with the number of attributes. It depends on the the maximum size of the priority queue and the total number of dominance checks, which are shown in Fig. 15(c) and 15(d), respectively. In Fig. 15(c) and 15(d), both the size of the queue and the number of dominance checks increase with the dimension. One reason is that when the dimension increases, the number of nodes in the R-tree grows and more overlap amongs MBRs is incurred. More patterns are hence generated, and the pruning power of PBP is reduced as well.

Observing Fig. 15(b), the time increases with the size of datasets, and the gap between two datasets with different sizes is more substantial for higher

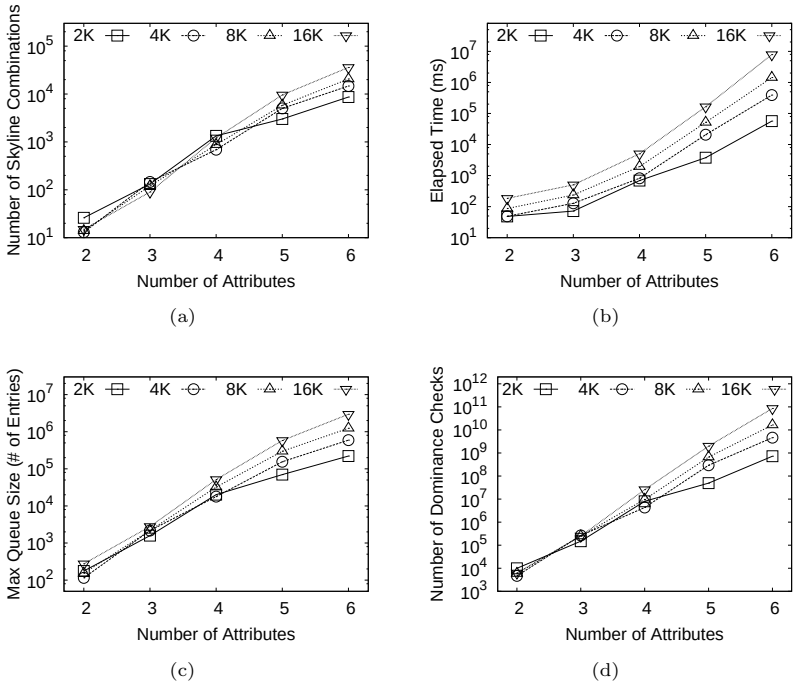


Fig. 15: PBP Performance for Different Number of Attributes

dimensionality. This is also due to the increase of nodes and more overlap in the R-tree.

The Effect of R-Tree Fanout The structure of R-tree may also impact the performance of PBP. Under the in-memory setting, the dominant factor of the algorithm's runtime performance is not I/O but the number of patterns processed. In addition, a large fanout, which is preferred in a disk-resident R-tree, is not necessary a good choice.[6] As shown below, a small fanout shows better performance in our problem setting. Consider an R-tree of order (m, M) where each node must have at most M child nodes and at least m child nodes. Note that m decides the fanout of the R-tree. There are at most $\lceil N/m^i \rceil$ nodes at the level i in the R-tree[7], and thus there are at most $\binom{\lceil N/m^i \rceil + k - 1}{k} \leq \frac{(\lceil N/m^i \rceil + k - 1)^k}{k!}$ patterns at the corresponding level i in the pattern tree. In the worst case, the

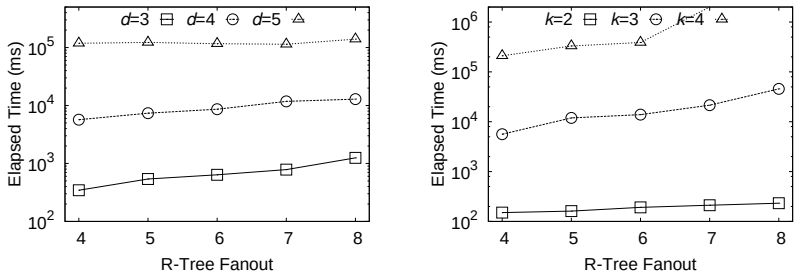
[6] For example, T-tree, an in-memory index for ordered keys has a binary index structure [11].

[7] Note that level-1 denotes the leaf level and level-$(i+1)$ denotes the parent level of level-i.

total number of patterns is

$$\sum_{i=1}^{\lceil \log_m N \rceil} \frac{(\lceil \frac{N}{m^i} \rceil + k - 1)^k}{k!} \tag{4}$$

where $\lceil \log_m N \rceil - 1$ is the maximum height of the R-tree. When m increases, the number of patterns decreases according to Equation 4, however, the pruning capabilities of Theorem 1 and 2 becomes weaker since the lower and upper bounds of a pattern become looser and less accurate.



(a) (b)

Fig. 16: PBP Performance for Different Fanouts of R-Tree

We run PBP on the datasets indexed by the different R-tree structures with the fanouts $m \in [4, 8]$. Fig. 16(a) shows the running time on three datasets with dimensions $d = 3$, $d = 4$, and $d = 5$. Each dataset has 1K objects and the algorithm searches for skyline combinations of cardinality $k = 3$. For the datasets with dimensions $d = 3$ and 4, PBP performs best when $m = 4$. In our experiments, when $m = 4$ we enumerate 341.3K patterns while when $m = 8$ we enumerate 690.5K patterns, which showcases the better pruning power of the proposed algorithm under small fanouts. For the dataset with dimension $d = 5$, PBP performs best when $m = 7$. The reason is that the increase of dimensionality causes more overlaps between MBRs and thus weaken the pruning power. We also found that a large fanout, which is preferred in a disk-resident R-tree, usually results in bad performance. When $k = 3$ and $d = 3$, the running time is 41.9s under a fanout $m = 32$, 121.5 times slower than using $m = 4$. Fig. 16(b) shows the running time on a four-dimensional dataset containing 1K objects. The algorithm searches for skyline combinations of cardinalities $k = 2$, $k = 3$, and $k = 4$ and performs best when $m = 4$. In general, we suggest users choose a small fanout, e.g., $m = 4$, for tasks with low dimensionality, and a moderately larger fanout, e.g., $m = 7$, for high-dimensional tasks.

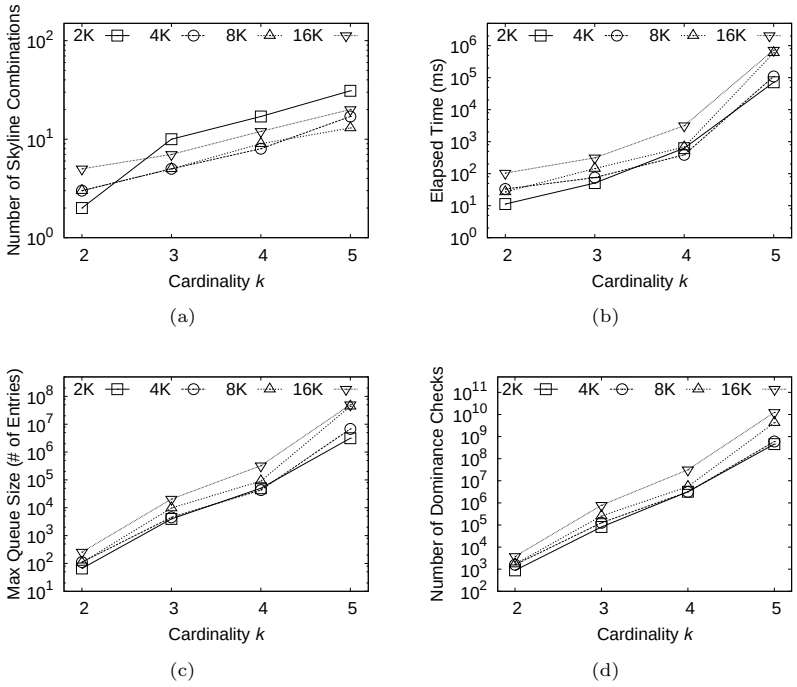


Fig. 17: PBP Performance for Different Cardinalities on Real Datasets

7.3 Experiments on Real Datasets

We run PBP on the real datasets. The sizes of our datasets are 2K, 4K, 8K, 16K, and the number of attributes varies from 2 to 5. We conduct two groups of experiments: one is to verify the effect of cardinality k, and another is to verify the effect of dimensionality $|\mathcal{A}|$.

Fig. 17 shows the effect of cardinality on real datasets. Fig. 17(a) shows the number of skyline combinations grows with the cardinality. Fig. 17(b) shows the running time increases with the cardinality, which is consistent with the increase of the queue size and the increase of the dominance check number shown in Fig. 17(c) and 17(d), respectively. A similar trend is observed as we have seen for synthetic datasets, but has a more rapid growth of running time with the cardinality. This is because the real dataset follows anti-correlated distribution while the synthetic dataset follows uniform distribution, and hence fewer combinations are dominated for the former.

Fig. 18 shows the effect of dimensionality on real datasets. Fig. 18(a) shows the number of skyline combinations is larger for higher dimensional datasets. Fig. 18(b) shows the running time of PBP on the real datasets with different number of attributes. Since the time depends on the size of the queue and the total number of dominance checks, the shapes and trends of the curves in Fig. 18(c) and 18(d) are consistent with the appearances of curves in Fig. 18(b). When the

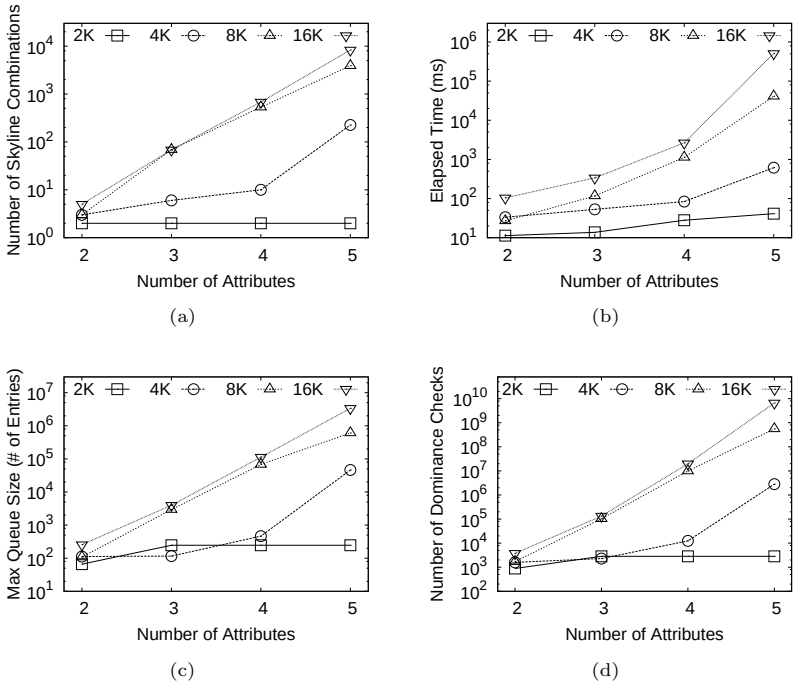


Fig. 18: PBP Performance for Different Number of Attributes on Real Datasets

number of attributes grows, the time increases and the gap between two datasets with different sizes is enlarged, and is more substantial than on synthetic data.

8 Conclusion and Future Work

In this paper, we have studied the combination skyline problem, a new variation of the skyline problem. The combination skyline problem is to find combinations consisting of k objects which are not dominated by others. We have proposed the PBP algorithm to answer combination skyline queries efficiently. With an R-tree index, the algorithm generates combinations with object-selecting patterns organized in a tree. In order to prune the search space and improve the efficiency, we have presented two pruning strategies and a technique to avoid duplicate pattern expansion. The efficiency of the proposed algorithm was evaluated by extensive experiments on synthetic and real datasets.

In the future, we would like to extend our work in the following interesting directions. We plan to extend the k-item combination skyline problem to a general version where the cardinality k varies. We also plan to solve the problem when the aggregation function is not monotonic. Additionally, we will implement a prototype system to support the combination skyline queries based on the proposed ideas.

Acknowledgments. This research was partly supported by the Funding Program for World-Leading Innovative R&D on Science and Technology (First Program).

References

1. K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
2. S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.
3. C. Y. Chan, H. V. Jagadish, K.-L. Tan, A. K. H. Tung, and Z. Zhang. On high dimensional skylines. In *EDBT*, pages 478–495, 2006.
4. J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *ICDE*, pages 717–719, 2003.
5. K. Deb and D. Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, 2001.
6. M. Ehrgott and X. Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR Spectrum*, 22(4):425–460, 2000.
7. P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB*, pages 229–240, 2005.
8. X. Guo and Y. Ishikawa. Multi-objective optimal combination queries. In *DEXA*, pages 47–61, 2011.
9. A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *SIGMOD*, pages 47–57. ACM Press, 1984.
10. M. Hadjieleftheriou, E. G. Hoel, and V. J. Tsotras. SaIL: A spatial index library for efficient application integration. *GeoInformatica*, 9(4):367–389, 2005.
11. T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *VLDB*, pages 294–303, 1986.
12. X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: The k most representative skyline operator. In *ICDE*, pages 86–95, 2007.
13. D. Papadias, N. Mamoulis, and V. Delis. Algorithms for querying by spatial structure. In *VLDB*, pages 546–557, 1998.
14. D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.
15. S. B. Roy, S. Amer-Yahia, A. Chawla, G. Das, and C. Yu. Constructing and exploring composite items. In *SIGMOD*, pages 843–854, 2010.
16. A. D. Sarma, A. Lall, D. Nanongkai, R. J. Lipton, and J. J. Xu. Representative skylines using threshold-based preference distributions. In *ICDE*, pages 387–398, 2011.
17. M. A. Siddique and Y. Morimoto. Algorithm for computing convex skyline objectsets on numerical databases. *IEICE*, 93-D(10):2709–2716, 2010.
18. I.-F. Su, Y.-C. Chung, and C. Lee. Top-k combinatorial skyline queries. In *DASFAA*, pages 79–93, 2010.
19. Y. Tao, L. Ding, X. Lin, and J. Pei. Distance-based representative skyline. In *ICDE*, pages 892–903, 2009.
20. Q. Wan, R. C.-W. Wong, and Y. Peng. Finding top-k profitable products. In *ICDE*, pages 1055–1066, 2011.