Skyline Groups

Chengkai Li
Univ. of Texas at Arlington

Nan Zhang
George Washington Univ.

Naeemul Hassan
Univ. of Texas at Arlington

Sundaresan Rajasekaran
George Washington Univ.

Gautam Das
Univ. of Texas at Arlington

ABSTRACT

We define a novel problem of finding from an n-tuple dataset the *skyline k-tuple groups* - i.e., a group of k tuples which is not dominated by any other group of equal size, based on certain notion(s) of group dominance relationship. The concept of skyline groups finds applications in numerous domains ranging from online fantasy games to expert finding and project-team formations. We consider various group dominance relationships of interest in practice and summarize them into two categories: *pairwise-comparison based* and *aggregate based* relationships. We show that while finding skyline groups for the first category can be reduced to finding skyline tuples, finding skyline groups according to aggregate based relationships results in completely different challenges and requires novel techniques to be developed.

In particular, we find the major technical challenge to be identifying effective anti-monotonic properties for pruning the search space of skyline groups. To this end, we first show that the anti-monotonic property used in the well-known *Apriori* algorithm does not hold for skyline group pruning. Then, we identify two anti-monotonic properties with varying degrees of applicability: *order-specific property* which applies to SUM, MIN, and MAX as well as *weak candidate-generation property* which applies to MIN and MAX only. Experimental results on multiple real-world datasets verify that the proposed algorithms achieve orders of magnitude performance gain over the exhaustive method.

1. INTRODUCTION

1.1 Problem Motivation

The problem of computing all *skyline tuples* of a multidimensional dataset has been extensively investigated in recent years [5, 8, 10, 12, 14, 20, 24]. Consider a database table D of n tuples $\{t_1, \ldots, t_n\}$ and m numeric attributes $A_1, \ldots, A_m$, where the domain of each attribute has a preference order, with larger values being preferred over smaller values. A tuple t_1 *dominates* t_2 if and only if every attribute value of t_1 is either larger than or equal to the corresponding value of t_2 according to the preference order and t_1 has larger value on at least one attribute. The set of skyline tuples are those tuples that are not dominated by any other tuples in the dataset. In a sense, the skyline set represents the "extremal" tuples of a dataset. This concept has many useful applications. As a motivating example, consider a database consisting of the pool of available NBA basketball players. Each player is represented as a tuple consisting of several statistical categories: points per game, rebounds per game, assists per game, steals per game, blocks per game, etc. The skyline for this dataset represents all the elite players, i.e., the players that are not dominated by any other player in all aspects of their abilities. Determining such skyline tuples also has applications in top-k query processing (since the top tuples for any monotonic scoring function belong to the skyline set).

In this paper we propose and investigate a novel extension to the skyline problem that surprisingly does not appear to have been studied in prior work: the problem of computing the **skyline groups** of a dataset. The concept of skyline groups finds applications in various places, such as selecting a roaster of players to compete in an online fantasy sports game, choosing a panel of experts to evaluate a paper or a research proposal, forming a team to collaborate on a software development project, and so on. Consider again the dataset of available NBA players. Suppose the task now is to form a strong basketball team consisting of 5 players selected from this pool to, say, compete in an online fantasy basketball game. Since there can be $\binom{n}{5}$ different candidate teams, this naturally raises two question: (a) how do we extend the traditional notion of dominance between two individual tuples to the notion of dominance between two groups of 5 tuples each, and (b) using this notion of group dominance, how do we compute the *skyline groups* of 5 tuples each from all possible $\binom{n}{5}$ groups? The (significantly smaller) returned skyline groups can be useful to end users (such as fantasy game players), who can browse the skyline groups and narrow down to their specific requirements, or even use top-k algorithms to return the top teams according to application-specific ranking functions.

Our investigations of the first question revealed that the notion of dominance relationship between different groups may naturally take various forms/definitions. In this paper we consider several natural definitions of group dominance, and investigate how to efficiently compute the skyline groups for each such definition. Interestingly, in this paper we demonstrate that our skyline group problem is significantly different from the more traditional skyline tuples problem, to the extent that algorithms for the later (or any simple extensions) are quite inapplicable in solving the former. One of the main contributions of this paper is the development of novel algorithmic techniques, in particular *output compression*, *input pruning*, and *search space pruning* techniques, that are leveraged in our eventual algorithms for computing skyline groups.

1.2 Group Comparison Functions

While there can be many ways in which group dominance can be defined, in this paper we introduce two classes of group compari-

son functions: *pairwise-comparison based* and *aggregate based*, respectively. A group comparison function belongs to the pairwise-comparison category if and only if it is determined solely by the pairwise dominance relationships between individual tuples in the two groups (a simple example is where a group G may dominate another group G' if each tuple in G dominates all tuples in G'). Somewhat surprisingly, we show that for most real world databases, finding all skyline groups is extremely easy for *any* group comparison function which belongs to the pairwise-comparison category– the task is as simple as finding the (traditional) set of all skyline tuples and then listing all of its subsets. Therefore much of the paper is focused on the second class of aggregate based group comparison functions. A group comparison function belongs to the aggregate based category if and only if the two input groups G and G' are compared by replacing each group with a single aggregate tuple (i.e., whose attribute values are aggregated over the corresponding attribute values of the tuples of the group), and then comparing the two aggregate tuples using traditional tuple dominance functions. While many aggregate based functions can be considered for comparing two groups, in this paper we focus on three natural and distinct aggregate based functions– SUM, MIN and MAX.

1.3 Challenges: The Inapplicability of Traditional Skyline Algorithms

Given a specific aggregate based group comparison function, we focus next on the task of computing all skyline groups (say of size k). A simple solution to the problem is to first list every possible combination of $\binom{n}{k}$ tuples, compute the aggregate tuple for each combination, and then call any traditional skyline tuple algorithm to identify the skyline groups. The main problem with such an approach is the significant computational and storage overhead of having to create this huge intermediate input for the traditional skyline tuple algorithm (i.e., $O(\binom{n}{k})$ for an n-tuple input dataset). The skyline groups problem also has another idiosyncrasy that is not shared by the traditional skyline tuples problem. For certain aggregate functions, specifically MAX and MIN, even the output size– i.e., the number of skyline groups produced– while significantly smaller that $\binom{n}{k}$, may be nevertheless too large to explicitly compute and store. In this paper we develop novel techniques to address these two problems, namely *output compression*, *input pruning*, and *search space pruning*. We discuss these briefly next.

1.4 Algorithms: Output Compression, Input Pruning, and Search Space Pruning

For MAX and MIN aggregates, we observe that numerous groups may share the same aggregate tuple. Our approach to compressing the output is to list the distinct aggregate tuples, each representing possibly many skyline groups, but also providing enough additional information so that the actual skyline groups can be reconstructed if required. Interestingly, there is a difference between MIN and MAX in this regard: while the compression for MIN is relatively efficient, the compression for MAX requires the solution to the NP-Hard *Set Cover Problem* (which fortunately is not a real issue in practice, as we shall show in the paper).

Our approach to input pruning is to filter the input tuples to significantly reduce the input size to the search of skyline groups. Our main observation is that if a tuple t is dominated by k or more tuples in the original database, then we can safely exclude t from the input without influencing the distinct aggregate vectors found at the end. We also find that for MAX only, we can safely exclude any non-skyline-tuple from the input without influencing the results.

Our final ideas (perhaps, technically the most sophisticated of the paper) are on search space pruning– i.e., instead of enumerating each and every k-tuple combination, we quickly exclude a large number of combinations which are not on the skyline from consideration. To enable such candidate pruning, we identify and leverage a number of anti-monotonic properties similar to the one used in the well-known *Apriori* algorithm for frequent itemset mining [1]. However, it is important to emphasize here that the anti-monotonic property leveraged by the Apriori algorithm– i.e., every subset of a group "of interest" (e.g., a group of frequent items or a skyline group) must also be "of interest" itself– *does not hold* for skyline groups defined by SUM, MIN or MAX. Thus, a significant part of our technical contribution is the identification of alternate anti-monotonic properties which serve our algorithms. In particular, we identify two different anti-monotonic properties with varying degrees of applicability: (a) *Order-Specific Anti-Monotonic Property*, a generic anti-monotonic property that applies to SUM, MIN and MAX, and (b) *Weak Candidate-Generation Property* which applies to MIN and MAX but not SUM. Based on the two anti-monotonic properties, we develop algorithms to compute skyline groups. These algorithms iteratively generate larger candidate groups from smaller ones and prune candidate groups by the anti-monotonic properties. For each individual property, a different candidate generation and pruning algorithm is devised. In particular, we develop a dynamic programming based algorithm that leverages the order specific anti-monotonic property, and an iterative algorithm that leverages the weak candidate-generation property. For the details of these properties and their respective algorithms, we defer the reader to later sections in the paper.

1.5 Summary of Contributions

- We introduce and motivate the novel problem of computing skyline groups, and discuss the inapplicability of traditional skyline tuple algorithms in solving this problem.
- We develop several group comparison functions by natural extensions of the concept of tuple dominance, in particular based on group aggregates such as SUM, MIN and MAX.
- We develop novel algorithmic techniques for output compression, input pruning, and search space pruning. In particular, search space pruning requires the identification and leveraging of interesting anti-monotonic properties to filter out candidate groups from consideration.
- We run comprehensive experiments on several real datasets (NBA and stock data) to evaluate the effectiveness of our proposed algorithms.

2. SKYLINE GROUPS PROBLEM

2.1 Running Example

Table 1 depicts a 5-tuple, 2-attribute database table which we shall use as a running example throughout this section. Figure 1 depicts the five tuples on a 2-dimensional plane defined by the two attributes. The symbols corresponding to MIN and MAX skyline aggregate vectors shall be explained in the later part of the paper.

2.2 Problem Definition

Consider a database table D of n tuples $\{t_1, \ldots, t_n\}$ and m attributes $A_1, \ldots, A_m$. We refer to any subset of k tuples in the table, i.e., $G : \{t_{i1}, \ldots, t_{ik}\} \subseteq D$, as a *$k$-tuple group*. Our objective is to find, for a given k, all k-tuple groups "of interest" according to certain application-specific preferences. In particular, we capture such preferences as a combination of total orders for all attributes, where each total order is defined over (all possible values of) an attribute, with "larger" values always preferred over "smaller" values. For example, to select an elite group of 5 NBA players, the

	A_1	A_2
t_1	3	0
t_2	0	3
t_3	2	1
t_4	2	2
t_5	0	2

Table 1: Running Example

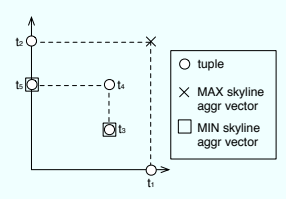


Figure 1: Running Example in 2-d Space

natural order of real numbers may be used for attribute field goal percentage, while the exact opposite order may be used for attribute turnovers per game. In the running example, we consider the natural order of real numbers as the preference order for all attributes. According to such total orders of attribute values, our goal is to find a *skyline* of k-tuple groups, defined in analogy to the traditional definition of skyline tuples [5].

In particular, similar to the case of skyline tuples, whether a k-tuple group belongs to the skyline or not is determined by the comparison, i.e., the "dominance relationship", between this group and other k-tuple groups in the database table. Such a *group comparison function*, when taking two groups G_1 and G_2 as input, produces (one of) three possible outputs: G_1 dominates G_2, G_2 dominates G_1, or neither dominates the other. According to the output of a group comparison function, a k-tuple group is a *skyline k-tuple group*, or *skyline group* in short (without causing ambiguity), if and only if it is *not dominated* by any other k-tuple group in D.

One can easily observe from the definition the analogy to skyline tuples: in particular, all existing work on skyline tuples used a uniform definition of "tuple comparison function" - i.e., tuple t_1 dominates t_2 if and only if every attribute value of t_1 is either larger than or equal to the corresponding value of t_2 according to the preference order (e.g., in the running example, t_2 dominates t_5 while neither t_2 nor t_3 dominates each other). In contrast, as we shall discuss in the next subsection, there is no single definition of group comparison function which stands out as in the skyline tuple case - instead, there are various definitions of group comparison functions which may fit different real-world applications. We describe a taxonomy of group comparison functions and discuss their corresponding applications in the following subsection.

2.3 Taxonomy of Group Comparison Functions

A group comparison function may be defined based on the comparison between individual tuples in the two groups, or by comparing the summary of each group, etc. In the following, we describe two types of group comparison functions, *pairwise-comparison* and *aggregate* based functions, respectively. For each type, we discuss its real-world applications and the challenges (or easiness) of finding all skyline groups defined accordingly.

2.3.1 Pairwise Comparison Based Functions

Definition and Applications: A group comparison function belongs to the pairwise-comparison based category if and only if it is determined solely by the pairwise comparisons between individual tuples in the two groups. Essentially, this means that the only input taken by the function is a directed bipartite graph, with tuples in each group forming the vertices of one side of the graph, and an edge exists from one tuple t_1 to another tuple t_2 (in the other group) if and only if t_1 dominates t_2 (according to the traditional definition of tuple dominance relationship). Figure 2 depicts an example of such a graph for two groups in the running example: $G : \{t_2, t_3, t_4\}$ and $G' : \{t_3, t_4, t_5\}$.

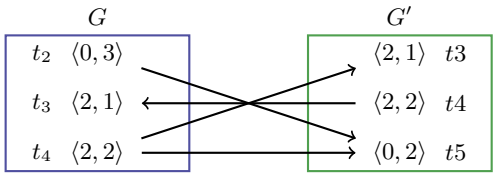


Figure 2: A directed bipartite graph between two groups

Based on the bipartite graph, the group comparison function can be defined in various ways. For example, one can say that group G dominates G' if and only if there exists at least one edge pointing from a tuple in G to G', but no edge from G' to G. According to this definition, the two groups in Figure 2 would be incomparable with each other. Another possible definition is that group G dominates G' if and only if the sum of outgoing degrees for all nodes in G is greater than that for G'. According to this definition, G_1 dominates G' in Figure 2.

One can see that pairwise comparison based functions best suit applications which require tuples in a selected (skyline) group to later "compete" with other tuples (in other groups). For example, when the task is to select tennis players for a team tournament like the Davis Cup, the main concern is how many pairwise matches the selected players can win by defeating players in other groups, making the pairwise comparison based function an appropriate model.

While a wide variety of group comparison functions may be defined in this category, we would like to point out two principles that any *reasonable* pairwise-comparison based function must follow: (1) if a group has no incoming edge in the bipartite graph, then it should not be dominated by the other group, (2) if a group has no outgoing edge to but only incoming edges from another group, then it should be dominated by that group. Intuitively, these two rules mean that the first definition listed above - i.e., group G dominates G' if and only if there exists an edge pointing from G to G', but not vice versa - is the *strongest* form of definition in the sense that if G dominates G' according to this definition, then the domination relationship should be upheld by any other definition in the category.

Finding All Skyline Groups: Somewhat surprisingly, for most real-world databases, finding all skyline groups is extremely easy for *any* group comparison function which belongs to the pairwise-comparison based category - as we shall show as follows, the task is as simple as finding the set of all skyline tuples (in the traditional sense) and then list all of its subsets. To understand why, it is important to note that the definition of a pairwise-comparison based function is closely intertwined with the traditional definition of tuple dominance relationship. In particular, we make the following two observations for skyline groups defined by a pairwise-comparison based function:

1. A group which consists solely of skyline tuples is always a skyline group because such a group has no incoming edge.
2. In a database D with more than k skyline tuples, any group G which contains a non-skyline tuple t is *not* a skyline group because it is always possible to find another group G' which consists solely of skyline tuples, and includes a tuple which dominates t. When G is compared with G', G has no outgoing edge but at least one incoming edge - i.e., G is dominated by G'.

According to these observations, when the number of skyline tuples is at least the size of skyline group k - which holds for almost all real-world databases given that k is usually a small value - the problem of finding all skyline groups can be solved by first finding the set of all skyline tuples, and then listing all size-k subsets of it as the all-and-only skyline k-tuple groups. As such, we do not further consider pairwise-comparison based functions in this paper.

2.3.2 Aggregate Based Function

Definition and Applications: A group comparison function belongs to the aggregate-based category if and only if it uses the following two steps to compare two input groups G and G':

1. For each input group, compute an *aggregate vector*, i.e., an m-dimensional vector with the i-th element being an aggregate value of A_i over all k tuples in the group, as the summary of the group.
2. Compare the aggregate vectors for the two groups according to the traditional definition of tuple dominance relationship, and output the result.

One can compute the aggregate vectors for input groups using many aggregate based functions. In this paper we focus on three aggregate functions: SUM, MIN, and MAX. Table 2 shows a sample case for each aggregate function according to the running example.

	Tuples	SUM	MAX	MIN
G	$t_2\langle 0,3\rangle$ $t_3\langle 2,1\rangle$ $t_4\langle 2,2\rangle$	$\langle 4,6\rangle$	$\langle 2,3\rangle$	$\langle 0,1\rangle$
G'	$t_3\langle 2,1\rangle$ $t_4\langle 2,2\rangle$ $t_5\langle 0,2\rangle$	$\langle 4,5\rangle$	$\langle 2,2\rangle$	$\langle 0,1\rangle$
Dominance Relationship		$G \succ G'$	$G \succ G'$	$G = G'$

Table 2: Examples of aggregate based functions

Finding All Skyline Groups: Unlike pairwise comparison based functions, it is non-trivial to find all skyline groups when the comparison function is aggregate based. In particular, we have the following two observations which stand in sharp contrast with those for pairwise comparison based functions:

1. A group solely consisting of skyline tuples may *not* be a skyline group. Consider group $G = \{t_1, t_2\}$ in the running example. Note that both t_1 and t_2 are skyline tuples. Nonetheless, with SUM-based group comparison function, G is dominated by $G' = \{t_3, t_4\}$, as SUM$(G) = \langle 3,3\rangle$ while SUM$(G') = \langle 4,3\rangle$. As such, G is not on the skyline.
2. A group containing non-skyline tuples could be a skyline group, even if there are skyline tuples which are not included in the group. Again consider the running example, this time with $G = \{t_4, t_5\}$ and MIN-based group comparison function. Note that t_5 is not on the skyline as it is dominated by t_2 and t_4. Nonetheless, G (with MIN$(G) = \langle 0,2\rangle$) is actually on the skyline, because the only other groups which can reach $A_2 \geq 2$ in the aggregate vector are $\{t_2, t_4\}$ and $\{t_2, t_5\}$, both of which yield an aggregate vector of $\langle 0,2\rangle$, the same as MIN(G). Thus, G is on the skyline despite containing a non-skyline tuple.

One can see from the two observations that, for aggregate based functions, the relationship between skyline groups and skyline tuples becomes much more complex (than that for pairwise comparison based functions) - indicating the difficulty of finding skyline groups for aggregate based functions.

3. MAIN IDEAS FOR FINDING SKYLINE GROUPS

In this section, we develop our main ideas for finding skyline k-tuple groups (hereafter simply referred to as skyline groups) defined by an aggregate-based group comparison function.

3.1 Challenges

We start by considering a brute-force approach which first enumerates each possible combination of k tuples in the input table, computes the aggregate vector for each combination, and then invokes a traditional skyline-tuple-search algorithm to find all skyline groups. This approach has two main problems. One is its significant computational overhead, as the input size to the final step - i.e., skyline tuple search - is $\binom{n}{k}$, which can be extremely large for real-world databases.

The other problem is actually is on the seemingly natural strategy of listing all skyline groups as the output. The problem here is that, for certain aggregate functions (e.g., MAX and MIN), even the output size - i.e., the number of skyline groups produced - may be too large for practical databases. Consider MAX as an example. If a tuple t dominates all other tuples in the input table, then every k-tuple combination which contains t is a MAX skyline group - leading to a total of $O(n^k)$ skyline groups. One can see that such a large output size makes the direct-skyline-usage case discussed in Section 1 hardly meaningful, because a human user cannot reasonably go through all skyline groups. Even for indirect-usage applications (e.g., pre-processing for top-k queries), the large number of skyline groups may lead to significant storage overhead.

We address both challenges in this section. We start with developing an output-compression technique that significantly reduces the output size when the number of skyline groups is large, thereby enabling both direct and indirect skyline-usage applications. Then, we consider how to efficiently find skyline groups. In particular, we shall describe two main ideas. One is input pruning - i.e., filtering the input tuples to significantly reduce the input size to the search of skyline groups. The other is search space pruning - i.e., instead of enumerating each and every k-tuple combination, we develop techniques to quickly exclude a large number of k-tuple combinations (which are not on the skyline) from consideration. Note that the two types of pruning techniques are transparent to each other and therefore can be readily integrated.

3.2 Output Compression for MIN and MAX

Main Idea: Note that out of the three aggregate functions we consider in the paper, i.e., SUM, MIN and MAX, the SUM function rarely, if ever, requires output compression. The intuitive reason is that, for any attribute, the SUM aggregate of a skyline group is sensitive to all tuples in the group, while MIN (resp. MAX) aggregate is in general only sensitive to tuples with minimum (resp. maximum) values on certain attributes, making it much more likely for two groups to share the same MIN (resp. MAX) vector. Thus, we focus on MIN and MAX output compression.

A key observation driving our design of output compression is that while the number of MIN and MAX skyline groups may be extremely large, many of these skyline groups share the same *aggregate vector*. Thus, our main idea for compressing skyline groups is to store not all skyline groups, but only the (much fewer) distinct aggregate vectors as well as one skyline group for each distinct vector. One can see that the "sample" skyline groups stored here are useful for many applications discussed in Section 1 (e.g., team building for fantasy games), as they only require one skyline group to be found among many which share the same score.

Reconstructing all Skyline Groups: While the distinct aggregate vectors and their accompanied (sample) skyline groups may suffice for many applications, there are also others which need to enumerate all skyline groups with the same aggregate vector. For example, when skyline group generation is used as a pre-processing step for top-k query processing, one may want a list of all groups which share the aggregate vector with the highest score. Thus, we now discuss how one can reconstruct all skyline groups from a given aggregate vector.

Consider MIN first. For a given MIN vector v, the search process is as simple as finding $\Omega(v)$, the set of all input tuples which dominate or are equal to v, because while every k-tuple subset of $\Omega(v)$ has an aggregate vector which either dominates or is equal to v (and therefore is a skyline group), any skyline group which con-

tains a tuple outside $\Omega(v)$ must have an aggregate vector dominated by v, and therefore cannot be on the skyline. One can see that the time complexity for finding $\Omega(v)$ is $O(n)$. Given $\Omega(v)$, the only additional step needed is to output all k-tuple subsets of $\Omega(v)$.

For MAX, the problem is, interestingly, much harder. To understand why, consider each tuple as a set consisting of all attributes for which the tuple reaches the same value as the MAX aggregate vector. One can see that the problem is now transformed to finding all combination of k tuples such that the union of their corresponding sets is the universal set of all attributes - i.e., finding all set covers of size k. The NP-hardness of this problem directly follows from the NP-completeness of SET-COVER, seemingly indicating that MAX skyline groups should not be compressed.

Fortunately, despite of the theoretical intractability, finding all skyline groups matching a given MAX aggregate vector v is usually efficient in practice, mainly because the number of tuples which "hit" the MAX attribute values in v - i.e., the input size - is in general extremely small for practical databases. As such, even a brute-force enumeration can be efficiently done, as demonstrated by the experimental results in Section 5.

Summary: In the rest part of the paper, we shall focus on the problem of finding all skyline k-tuple groups for SUM, and finding all distinct aggregate vectors and their accompanying (sample) skyline groups for MIN and MAX. Then, for MIN and MAX, we shall show through experimental results in Section 5 that the generation of all skyline groups from distinct aggregate vectors is an efficient process for practical databases. Before that, we use the term "skyline search" to refer to the process of finding all distinct aggregate vectors of k-tuple groups on the skyline.

Before starting the algorithmic discussions, we would like to make an important observation for the case of MAX when $k \geq m$, where k is the size of a skyline group, and m is the number of attributes. Since it takes at most m tuples to cover the MAX values of all attributes, there is only one distinct (skyline) aggregate vector in this case - which is the vector that takes the MAX value on every attribute. Thus, we only focus on the case of $k < m$ for MAX in the rest part of the paper.

3.3 Input Pruning

We now consider the pruning of input to skyline group searches - which is originally the set of all n tuples.

An important observation here is that if a tuple t is dominated by k or more tuples in the original database, then we can safely exclude t from the input without influencing the distinct aggregate vectors found at the end. To understand why, suppose that a skyline group G contains a tuple t which is dominated by h ($h \leq k$) tuples. One can see that there is always an input tuple t' which dominates t and is not in G. Since t' dominates t, the number of tuples which dominate t' must be smaller than h. Note that if t' is still dominated by k or more tuples, we can always repeat this process until finding $t' \notin G$ which is dominated by less than k tuples. Now consider the construction of another group G' by replacing t in G with t'. For SUM, one can see that G' always dominates G, contradicting our assumption that G is a skyline group. Thus, no skyline group under SUM can contain any tuple dominated by k or more tuples.

For MIN and MAX, it is also possible that the aggregate vector of G' is exactly the same as G. Even in this case, we can still safely exclude t from the input without influencing the distinct aggregate vectors found at the end. If there are other tuples in G which are dominated by k or more tuples, we can use the same process to remove them all and finally reach a group that (1) features the same aggregate vector as G, and (2) has no tuple dominated by k or more other tuples. Thus, we can safely remove from the input all tuples with at least k dominators for all aggregate functions - i.e., SUM, MIN and MAX.

Another observation for input pruning is that for MAX only, we can safely exclude any non-skyline-tuple t from the input without influencing the distinct aggregate vectors. The reason can be explained as follows. Suppose that a skyline group G contains a non-skyline-tuple t which is dominated by another skyline tuple t'. If $t' \notin G$, then we can replace t in G with t' to achieve the same (skyline) aggregate vector (because G is a skyline group) - i.e., t can be safely excluded from the input.

If $t' \in G$, then we consider the following two cases: (1) when G does not reach MAX on at least one attribute, say A_i. Let t_j be a tuple which reaches MAX on A_i. In this case, we can construct $G' = \{G \backslash t\} \cup \{t_j\}$ which is a k-tuple group and dominates G, contradicting our assumption of G being on the skyline. (2) when G reaches MAX on all attributes. One can see that there must be a group of size at most k which is formed solely by skyline tuples and reaches the same aggregate vector, because any tuple which reaches MAX on at least one attribute must be a skyline tuple[1]. Thus, we can safely exclude t from the input.

While the input size reduction is often significant (as we shall show in the experimental results), we cannot stop here because enumerating all k-tuple combinations of even the reduced input may still incur a significant overhead. Thus, we consider search space pruning for further improving the efficiency of skyline search in the next subsection.

3.4 Search Space Pruning: Anti-Monotonicity

Our principal idea for search space pruning is to find and leverage a number of *anti-monotonic properties* for skyline search, somewhat in analogy to the Apriori algorithm for frequent itemset mining [1]. Nonetheless, it is important to note that the original anti-monotonic property used by the Apriori algorithm - i.e., every subset of a group "of interest" (e.g., a group of frequent items or a skyline group) must also be "of interest" itself - does not hold for skyline search over any aggregate function (i.e., SUM, MIN or MAX)[2]. Thus, the key challenge here, and our focus in this subsection, is to find those anti-monotonic properties which hold for skyline search.

Before presenting the detailed properties, we would like to stress that the main contribution here is not about *proving* these properties, which is often straightforward, but rather about *finding* the right ones which can effectively prune the search space. For this very reason, our following discussions mainly focus on describing the anti-monotonic properties we found and discussing their effectiveness on improving the efficiency of skyline search.

3.4.1 Order-Specific Anti-Monotonic Property

Main Idea: Our first idea for finding an anti-monotonic property is to make a small revision to the classic property used by the apriori algorithm - specifically, by factoring in an order of all tuples in the database. To understand how, consider a skyline k-tuple group G_k which violates the apriori property - i.e., a $(k-1)$-tuple subset of it, $G_{k-1} \subseteq G_k$, is not be a skyline $(k-1)$-tuple group by itself. We note for this case that all $(k-1)$-tuple groups which dominate G_{k-1} must contain tuple $t_k = G_k \backslash G_{k-1}$. To understand why,

[1]Note that if there are fewer than k skyline tuples in the input, then we can immediately conclude that any skyline k-tuple group must reach MAX on all attributes.

[2]We have shown in Section 2.3.2 two examples which demonstrate the inapplicability for SUM and MIN, respectively. For MAX, the applicability can be easily observed from the fact that the set of all tuples is always a skyline n-tuple group, while many subsets of it are not on their corresponding skylines.

suppose that there exists a $(k-1)$-tuple group G' which dominates G_{k-1} but does not contain t_k. Then, $G' \cup \{t_k\}$ would always dominate $G_k = G_{k-1} \cup \{t_k\}$, contradicting the skyline assumption for G_k. One can see from this example that while a subset of a skyline group may not be on the skyline for the entire input table, it is always a skyline group over a subset of the input table - in particular, $D \backslash \{t_k\}$ in the above example. This observation leads to the following anti-monotonic property:

Definition 1: Order-Specific Property An aggregate function $\mathcal{F}$ satisfies the *order-specific anti-monotonic property* if and only if $\forall k$, if a k-tuple group G_k with aggregate vector v (i.e., $v = \mathcal{F}(G_k)$) is a skyline group, then for each tuple t in G_k, there must exist a set of $k-1$ tuples $G_{k-1} \subseteq D$ with $t \notin G_{k-1}$, such that (1) G_{k-1} is a skyline $(k-1)$-tuple group over an input table $D \backslash t$, and (2) $G_{k-1} \cup \{t\}$ is a skyline k-tuple group over the original input table D which satisfies $\mathcal{F}(G_{k-1} \cup \{t\}) = v$. ■

It may be puzzling from the definition where the "order" comes from - we note that it actually lies on the way search-space pruning can be done according to this anti-monotonic property: Consider an arbitrary order of all tuples in the input table, say $\langle t_1, \ldots, t_n \rangle$. For any $r < n$, if we know that an h-tuple group G_h ($h \leq r$) is *not* a skyline group over $\{t_1, \ldots, t_r\}$, then we can safely prune from the search space all k-tuple groups whose intersection with $\{t_1, \ldots, t_r\}$ is G_h - a reduction of the search space size by $O((n-r)^{k-h})$ - as Definition 1 clearly precludes such groups from being skyline k-tuple groups over the original input table. One can see that such a pruning technique considers all tuples in a specific order - hence the name of "order-specific" anti-monotonic property. More details of the pruning algorithm are discussed in Section 4.

Applicability of Order-Specific Property: The following theorem shows that the order-based property holds for all three aggregate functions we consider, i.e., SUM, MIN and MAX.

Theorem 1: (Positive Results) SUM, MIN and MAX satisfy the order-specific anti-monotonic property. ■

We do not include the proof as it follows directly from the definition. We do, however, want to note a limitation of the property. One can see from the above discussion that, to prune based on this order-specific property, one has to compute for every $h \in [k, n-k]$ the aggregate vectors of all skyline 1, 2, ..., $\min(k, h)$-tuple groups over the first h tuples (according to the order), because any of these groups may grow into a skyline k-tuple group for the database when latter tuples (again, according to the order) are brought into consideration. Given a large n (i.e., a long order), the order-specific pruning process may incur a significant overhead, as we shall show in Section 5. To address this problem, we consider order-free anti-monotonic properties as follows.

3.4.2 Weak Candidate-Generation Property: MIN and MAX only

We now describe an "order-free" anti-monotonic property which "loosens" the classic apriori property to one which holds for skyline search. The main idea is that, instead of requiring *every* $(k-1)$-tuple subset of a skyline k-tuple group to be a skyline $(k-1)$-tuple group (as in the Apriori property), we consider the following property which only requires *at least one* subset to be on the skyline.

Definition 2: (Weak Candidate-Generation Property) An aggregate function $\mathcal{F}$ satisfies the *weak candidate-generation property* if and only if, $\forall k$ and for any aggregate vector v_k of a skyline k-tuple group, there must exist an aggregate vector v_{k-1} for a skyline $(k-1)$-tuple group, such that for any $(k-1)$ tuple group G_{k-1} which reaches v_{k-1} (i.e., $\mathcal{F}(G_{k-1}) = v_{k-1}$), there must exist an input tuple $t \notin G_{k-1}$ which makes $G_{k-1} \cup \{t\}$ a skyline k-tuple group that reaches v (i.e., $\mathcal{F}(G_{k-1} \cup \{t\}) = v$). ■

An intuitive way to understand the definition is to consider the case where every skyline group has a distinct aggregate vector. In this case, the weak anti-monotonic property holds when every skyline k-tuple group has at least one $(k-1)$-tuple subset being a skyline $(k-1)$-tuple group - a property that is clearly "weaker" than the classic (Apriori) anti-monotonic property when being used for pruning, in the sense that it allows many more candidate sets to be generated than directly (and mistakenly) applying the classic property. We shall describe in Section 4 the detailed design of skyline-group generation algorithms which leverage the weak anti-monotonic property.

In general, this property avoids the pitfall of order-specific property by removing the requirement of enumerating all tuples in order and generating skyline groups for each subset of tuples along the way. Unfortunately, this weak candidate-generation property only holds for MIN and MAX, but not for SUM.

Theorem 2: (Positive Results: MIN and MAX) MIN and MAX satisfy the weak anti-monotonic property. ■

PROOF. We prove the case for MIN by contradiction. The case for MAX can be proved in analogy. Suppose that G_k is a skyline k-tuple group which satisfies $\mathcal{F}(G_k) = v_k$, but no v_{k-1} according to the definition exists. Consider an arbitrary $(k-1)$-tuple subset of G_k, denoted by G. Let t_1 be the other tuple not included in G - i.e., $t_1 = G_k \backslash G$. Since G is not a skyline $(k-1)$-tuple group, there must exist another $(k-1)$-tuple group G' which dominates G. We consider the following two cases respectively: (1) $t_1 \notin G'$, and (2) $t_1 \in G'$.

In Case 1, one can see that either $G' \cup \{t_1\}$ dominates G_k - which leads to contradiction because G_k is on the skyline - or $\mathcal{F}(G' \cup \{t_1\}) = \mathcal{F}(G_k)$ - which leads to contradiction as well because G' and t_1 would exactly satisfy the requirement of weak anti-monotonic property.

For Case 2, note that since G' and G are of equal size, there must at least one tuple in G which is not in G'. Let t_2 be such a tuple. Consider $G' \cup \{t_2\}$. Since $t_2 \in G$, every attribute value in $\mathcal{F}(G' \cup \{t_2\})$ is still greater than or equal to the corresponding value in $\mathcal{F}(G)$, which is in turn greater than or equal to that in $\mathcal{F}(G)$. Thus, we again reach the conclusion that either $G' \cup \{t_2\}$ dominates G_k - which leads to contradiction because G_k is on the skyline - or $\mathcal{F}(G' \cup \{t_2\}) = \mathcal{F}(G_k)$ - which leads to contradiction as well because G' and t_2 would exactly satisfy the requirement of weak anti-monotonic property.

Theorem 3: (Negative Result: SUM) SUM does not satisfy the weak candidate-generation property. ■

We would like to note that while the only proof needed here is one counter-example, our studies showed that finding such a counter-example is non-trivial. In particular, the weak candidate-generation property indeed holds when $k \leq 3$, but fails when $k \geq 4$. For $k = 4$, we constructed through MATLAB a 8-tuple, 69-attribute table as a counter-example. We do not include the example (which constitutes a proof) here due to space limitations.

4. ALGORITHMS

In this section, we develop skyline group search algorithms based on the anti-monotonic properties derived in Section 3.

4.1 Dynamic Programming Algorithm Based on Order-Specific Property

Consider an arbitrary[3] order of the n tuples in the input table, denoted by $t_1, \ldots, t_n$. Let T_r be the set of the first r according to this order - i.e., T_r=$\{t_1, \ldots, t_r\}$. Let Sky^r_k be set of all skyline k-tuple groups with regard to T_r - i.e., each group in Sky^r_k is not dominated by any other k-tuple group consisting solely of tuples in T_r. One can see that our original problem can be considered as finding Sky^n_k. We now develop a dynamic programming algorithm which finds Sky^n_k by recursively solving the "smaller" problems of finding Sky^{n-1}_k and Sky^{n-1}_{k-1}, etc.

The algorithm is based on the following idea - All skyline k-tuple groups in Sky^n_k can be partitioned into two disjoint sets S_1 and S_2 ($Sky^n_k \equiv S1 \cup S2$ and $S1 \cap S2 = \phi$) according to whether a group contains t_n or not. In particular, $S1 = \{G | G \in Sky^n_k, t_n \notin G\}$ and $S2 = \{G | G \in Sky^n_k, t_n \in G\}$. One can that there must be $S1 \subseteq Sky^{n-1}_k$. On the other hand, $S2$ is subsumed by a set of groups that can be expanded from Sky^{n-1}_{k-1}, the skyline (k-1)-tuple groups with regard to T_{n-1}. More specifically, given a skyline k-tuple group that contains t_n, if we remove t_n from it, then the resulting group belongs to Sky^{n-1}_{k-1}. These two properties are formally presented as follows. We omit the fairly simple proof. Note that Proposition 2 can be directly derived from Theorem 1.

Proposition 1: Given $G \in Sky^n_k$, if $t_n \notin G$, then $G \in Sky^{n-1}_k$. ■

Proposition 2: Given $G \in Sky^n_k$, if $t_n \in G$, then $G \backslash \{t_n\} \in Sky^{n-1}_{k-1}$. ■

Algorithm 1: $sky\_group(k, n)$: Dynamic programming algorithm based on order-specific property

Input: n: input tuples T_n=$\{t_1, \ldots, t_n\}$; k: group size; $k \leq n$
Output: Sky^n_k: skyline k-tuple groups among T_n

```
1  if Sky^n_k is computed then
2      return Sky^n_k;
3  if k == 1 then
4      S2^+ ← {{t_n}};
5  else
6      S2^+ ← φ;
7      Sky^{n-1}_{k-1} ← sky_group(k-1, n-1);
8      foreach group G ∈ Sky^{n-1}_{k-1} do
9          candidate_group ← G ∪ {t_n};
10         S2^+ ← S2^+ ∪ {candidate_group};
11 if k < n then
12     Sky^{n-1}_k (i.e., S1^+) ← sky_group(k, n-1);
13 else
14     S1^+ ← φ;
15 C^n_k ← S1^+ ∪ S2^+;
16 Sky^n_k ← skyline(C^n_k);
17 return Sky^n_k;
```

We further explain the dynamic programming algorithm by referring to the outline in Algorithm 1. The idea of Algorithm 1 is also intuitively illustrated in Figure 3. The function $sky\_group(k, n)$ is for finding Sky^n_k. It first computes Sky^{n-1}_{k-1} by calling function sky_group recursively (Line 7). By adding t_n into each group in Sky^{n-1}_{k-1} (Line 8-10), the algorithm obtains a superset of the aforementioned $S2$, according to Proposition 2. We denote this superset $S2^+$. By recursively calling the sky_group function (Line 12), it further computes Sky^{n-1}_k, which is a superset of the aforementioned $S1$, according to Proposition 1. We also denote Sky^{n-1}_k by $S1^+$. $S1^+$ and $S2^+$ thus contain all necessary candidate groups for Sky^n_k. Thus, the skyline over candidate groups (C^n_k=$S1^+ \cup S2^+$, Line 15) is guaranteed to be equal to Sky^n_k. Existing skyline query

[3]We consider a random order in the experimental studies of this paper and leave the problem of finding an optimal order (in terms of efficiency) to future work.

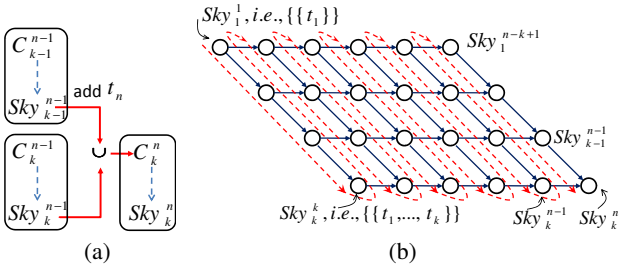


Figure 3: (a) Calculate Sky^n_k from Sky^{n-1}_{k-1} and Sky^{n-1}_k; (b) Overall dynamic programming algorithm for calculating Sky^n_k.

algorithms (e.g., [5, 10, 12]) can be applied for this purpose. Hence we use $skyline()$ to refer to such algorithms (Line 16). These algorithms involve comparing groups based on the dominance relationship between groups by their aggregated vectors. The number of candidate groups considered ($|S1^+ \cup S2^+|$) can potentially be much smaller than the number of all possible groups formed by all tuples, i.e., $\binom{n}{k}$.

Note that Sky^n_k is needed in calculating both Sky^{n+1}_k and Sky^{n+1}_{k+1}. The algorithm recursively calls $sky\_group(k, n)$ inside sky_group $(k, n+1)$, to compute and memoize Sky^n_k. Later it will call $sky\_group(k, n)$ again inside $sky\_group(k+1, n+1)$. This time Sky^n_k is not recomputed. Instead, the memorized result is directly used (Line 1). Hence it is a dynamic programming algorithm. The sequence of real calculation of Sky^1_1, Sky^2_2, ..., Sky^n_k is shown by the dashed directed lines in Figure 3(b).

In Line 16 of Algorithm 1, if we blindly apply a skyline algorithm over the candidate groups $S1^+ \cup S2^+$, it may perform unnecessary pairwise group comparisons. We further reduce comparisons as follows. By definition groups in $S1^+$ (i.e., Sky^{n-1}_k) do not dominate each other. To find Sky^n_k, we do not need to compare any two groups within $S1^+$. In other words, groups in $S1^+$ can only be possibly dominated by groups in $S2^+$. In Line 16 of Algorithm 1, we can first put all candidate groups from $S1^+$ into Sky^n_k, then iterate through $S2^+$ and compare each group G in $S2^+$ with the current groups in Sky^n_k. If any current group in Sky^n_k is dominated by G, then that group is removed from Sky^n_k. If G is not dominated by any current group, it is inserted into Sky^n_k. It is essentially the nested-loop skyline algorithm in [5].

4.2 Iterative Algorithm Based on Weak Candidate-Generation Property

The idea of weak candidate-generation property (Definition 2) can be summarized as follows - Given a skyline group G and any i, at least one i-tuple sub-group of G must be a skyline i-tuple group. Based on this property, Algorithm 2 iteratively generates candidate i-tuple groups by adding new tuples into skyline $(i-1)$-tuple groups (Line 6-12) and applies skyline algorithm over these candidates to find skyline i-tuple groups (Line 14). At every step of iteration, the algorithm only needs to generate i-tuple candidates by extending skyline $(i-1)$-tuple groups instead of all $(i-1)$-tuple groups. Hence it effectively prunes candidate groups by generation.

4.3 From Distinct Vectors to All Skyline Groups

For certain aggregate functions, specifically MIN and MAX, even the output size - i.e., the number of skyline groups produced - may be too large to explicitly compute and store. As discussed in Section 3.2, for output compression, we only need to retain one representative skyline group for each distinct aggregated vector. To be more specific, it is sufficient for Sky^n_k in Algorithm 1 and Sky_k in Algorithm 2 to contain one representative group for each distinct

Algorithm 2: $sky\_group(k, n)$: Iterative algorithm based on weak candidate-generation property

Input: n: input tuples T_n=$\{t_1, \ldots, t_n\}$; k: group size; $k \leq n$
Output: Sky_k: skyline k-tuple groups among T_n

```
1  C_1 ← T_n;
2  Sky_1 ← skyline(C_1);
3  for i ← 2 to k do
4      //generate candidate i-tuple groups C_i from skyline i−1-tuple
       groups Sky_{i−1}.
5      C_i ← φ;
6      foreach G ∈ Sky_{i−1} do
7          foreach t ∈ T_n do
8              //generate candidate group
9              if t ∉ G then
10                 G' ← G∪{G};
11             if G' ∉ C_i then
12                 C_i ← C_i∪{G'};
13     //generate skyline i-tuple groups Sky_i based on candidates C_i
14     Sky_i ← skyline(C_i);
15 return Sky_k
```

Algorithm 3: Finding skyline groups with identical aggregated vectors (MIN function)

Input: input tuples R; k: group size; $k < |R|$
Output: Sky: skyline k-tuple groups for R

```
1  Sky ← φ;
2  T ← remove k-dominator tuples from R;
3  n ← |T|; /* number the tuples in T as t_1, ..., t_n */
4  Sky_k ← sky_group(k, n); /* Algorithm 1 or Algorithm 2 */
5  foreach skyline k-tuple group G ∈ Sky_k do
6      R_G ← the set of tuples in R that dominate or are equivalent to
       the aggregated vector of G;
7      foreach k-combination G' of tuples in R_G do
8          Sky ← Sky ∪ {G'};
9  return Sky;
```

Algorithm 4: Finding skyline groups with identical aggregated vectors (MAX function)

Input: input tuples R; k: group size; $k < |R|$
Output: Sky: skyline k-tuple groups among R

```
1  Sky ← φ;
2  T ← remove k-dominator tuples from R;
3  n ← |T|; /* number the tuples in T as t_1, ..., t_n */
4  Sky_k ← sky_group(k, n); /* Algorithm 1 or Algorithm 2 */
5  foreach skyline k-tuple group G ∈ Sky_k do
6      v ← the aggregated vector of G
7      candidate_group ← φ;
8      i ← 1;
9      p[1] ← null;
10     while i > 0 do
11         /* Note that it is fine to select a tuple multiple times because a
           tuple can get the same value as v on multiple dimensions. */
12         candidate_group ← candidate_group \ {p[i]};
13         p[i] ← get the next tuple in R that has v's value on the ith
           dimension;
14         if p[i] == null then
15             i ← i−1;
16             continue;
17         candidate_group ← candidate_group ∪ {p[i]};
18         if |candidate_group| > k then
19             continue;
20         if i==d then
21             /* d is the number of dimensions. */
               k' ← k − |candidate_group|;
22             if k'==0 then
23                 Sky ← Sky ∪ {candidate_group};
24             else
25                 R' ← R \ candidate_group;
26                 foreach k'-tuple combination G' among the tuples
                   in R' do
27                     Sky ← Sky ∪ {candidate_group ∪ G'};
28         else
29             i ← i + 1;
30             p[i] ← null;
31 return Sky;
```

aggregated vector of k-tuple groups. It can be easily achieved by a simple modification of the skyline algorithm at Line 16 of Algorithm 1 and Line 14 of Algorithm 2. Whenever a candidate group is compared with current groups in the skyline, we prune it if it is equivalent to some existing group. This will further reduce the size of candidate groups and the number of group comparisons in succeeding iterations.

For input pruning, in the case of SUM and MIN, we remove all tuples dominated by at least k others. In the case of MAX, we remove all tuples not on the skyline. We showed in Section 3.3 that such input pruning techniques are safe - i.e., we will still obtain all distinct vectors and their representatives.

It is our belief that in many cases distinct vectors and their representative groups suffice. If an application requests all skyline groups instead of only their representatives, various postprocessing steps are required, due to output compression and input pruning. Below we discuss such postprocessing for individual functions.

Note that the same Algorithm 1 and 2 work if we do not apply output compression and input pruning. However, even if our application is to ultimately find all skyline groups, it is still beneficial to apply these two techniques and use postprocessing steps to find all skyline groups. Output compression and input pruning together not only reduce the output size, but also save computational cost by allowing the algorithms to deal with smaller input and intermediate results. In Section 5 we present experimental results to compare the execution time of our methods with and without k-dominator tuple pruning. The results verify the benefit of applying this pruning technique regardless of the ultimate output– representative groups for all distinct aggregated vectors or all skyline groups.

SUM: No postprocessing is necessary for SUM. First, a k-dominator tuple cannot appear in any skyline k-tuple group, as discussed in Section 3.3. Thus, input pruning will not trigger postprocessing for SUM. Second, if the ultimate goal is to fetch all skyline groups, output compression should not be applied, because there is no effective way of reconstructing skyline groups from distinct aggregated vectors. In Line 16 of Algorithm 1, all skyline i-tuple groups should be retained, without applying the aforementioned simple modification that removes equivalent groups. Note that SUM only satisfies the order-specific property. Thus, only Algorithm 1 applies.

MIN: Two factors contribute to the need for postprocessing. First, the pruned k-dominator tuples may appear in skyline groups. Second, the aforementioned equivalent group removal performed at Line 16 of Algorithm 1 and Line 14 of Algorithm 2 will only keep one representative for each distinct aggregated vector. Note that both algorithms are applicable to MIN since MIN satisfies both order-specific and weak candidate-generation properties. At the end of both algorithms, we obtain Sky_k, which contains representatives of all distinct aggregated vectors, but not necessarily all skyline k-tuple groups. To generate all skyline groups from Sky_k for MIN, we follow Algorithm 3. For each representative group, we find all the tuples that dominate or are equal to its aggregated vec-

						PPG	RBG	APG	SPG	BPG
G1	Carmelo Anthony	Kobe Bryant	Kevin Durant	LeBron James	Dwyane Wade	283.2	63.4	52.2	15.2	7.6
G2	Andrew Bogut	Marcus Camby	Monta Ellis	Dwight Howard	Josh Smith	166.2	96.4	32.2	13.4	19.4
G3	Trevor Ariza	Monta Ellis	Dwyane Wade	Dwight Howard	Josh Smith	202	72.6	43.2	16.6	14
G4	Carlos Boozer	Baron Davis	LeBron James	Rajon Rondo	Chris Paul	193.8	61.2	80.6	17.6	4.8
G5	Andrew Bogut	LeBron James	Chris Paul	Dwight Howard	Jason Kidd	185.8	81	64	14	13.8

Table 3: Sample skyline groups from 512 players, 5 players per group

tor. Any $k-$combination of these tuples is a skyline k-tuple group. This is based on the results from Section 3.2.

MAX: Algorithms 1 and 2 are both applicable to MAX. Similar to MIN, MAX needs postprocessing due to both input pruning and output compression. We thus devise Algorithm 4 to produce all skyline groups from representative groups.

For each representative group G that is found by Algorithms 1 and 2, Algorithm 4 uses a backtracking process to find all skyline groups that are equivalent to G. Denote the aggregated vector for G as v. On each dimension, we maintain a list of tuples from R (all input tuples to be considered) that attain v's value on that dimension. We use the backtracking algorithm to enumerate all possible groups of the tuples from these lists, such that the groups have the same aggregated vector v and have less than or equal to k tuples. If a group has less than k tuples, it means there can be some "free" tuples. Any combination of other tuples will complement this group to form a skyline k-tuple group (Line 25-27).

A special case for MAX function is when there is only one distinct aggregated vector, i.e., all skyline k-tuple groups reach the highest possible value on every dimension. In Algorithms 1 and 2, whenever an i-tuple candidate group ($i \leq k$) is generated, we test if this group attains the highest possible value on every attribute. If so, we has already found the aggregated vector for all skyline groups. Using that vector, we either find one representative group or all skyline groups, by a backtracking process that is essentially the same as Algorithm 4. We omit the details.

5. EXPERIMENTS

5.1 Experimental Setup

All the algorithms were implemented in C++. We executed all our experiments on a Dell PowerEdge 2900 III server running Linux kernel 2.6.27-7, with dual quad-core Xeon 2.0GHz processors, 2x6MB cache, 8GB RAM, and three 250GB SATA hard drivers in RAID5.

Datasets: We collected 512 tuples of NBA players who had played in the 2009 regular season. [4] The tuple of each player has five statistics that measure the player's performance. The statistics are points per game (PPG), rebounds per game (RPG), assists per game (APG), steals per game (SPG), and blocks per game (BPG). Players and groups of players are compared by these statistics and their aggregates.

Another dataset used in our experiments is a collection of 35002 tuples of stocks for all the publicly traded firms as of December 31st, 2009 in the US, Europe, Australia, New Zealand, Japan, China, India, Canada, Latin America, Eastern Europe, Middle East, Africa and some emerging markets from Asia. [5] Each tuple has four attributes, corresponding to market cap, stock price, interest coverage ratio and net income. All the values were calculated in US dollars.

Aggregate Functions and Methods Compared: We investigated the performance of the two algorithms discussed in Section 4, namely the algorithms based on order-specific property (OSM) and weak candidate-generation property (WCM). We also compared these methods with the exhaustive method (EXM) that considers all possible groups. We executed these methods for the aggregate functions discussed in previous sections– SUM, MIN, and MAX.

Parameters: We ran our experiments under combinations of two parameter values, which are number of tuples, i.e., dataset size (n) and number of tuples per group, i.e., group size (k).

Values Measured: For each applicable combination of aggregate function, method, and parameter values, we measured the execution time needed to find all distinct aggregate vectors and their representative groups, as well as the time to find all skyline groups. Besides execution time, we also measured the total number of candidate groups generated and number of pairwise group (aggregated vector) comparisons in the process. Due to the iterative nature of OSM and WCM, they call the basic skyline function multiple times. Hence, the total number of generated candidate groups is the cumulative sizes of inputs to all skyline function invocations. Furthermore, OSM produces candidate groups by merging two disjoint sets of smaller groups. Here input size was calculated as the summation of the sizes of disjoint sets.

n		$k=1$			$k=3$			$k=5$		
		G	S	V	G	S	V	G	S	V
100	SUM	19	19	19	4495	325	325	435897	1461	1461
	MIN		19	19		122	122		231	231
	MAX		19	19		34	27		2	1
200	SUM	22	22	22	7770	594	594	1.5×10^6	3541	3541
	MIN		22	22		172	172		369	369
	MAX		22	22		28	28		1	1
300	SUM	24	24	24	19600	420	420	5.9×10^6	3155	3155
	MIN		24	24		192	192		527	527
	MAX		24	24		6	6		296	1
400	SUM	29	29	29	37820	582	582	2.1×10^7	4042	4042
	MIN		29	29		261	261		705	705
	MAX		29	29		7	7		396	1
500	SUM	30	30	30	37820	757	757	2.9×10^7	5324	5324
	MIN		30	30		309	309		832	832
	MAX		30	30		19	19		496	1

Table 4: Number of all groups (G), skyline groups (S), and distinct vectors for skyline groups (V), under different n, k, and functions

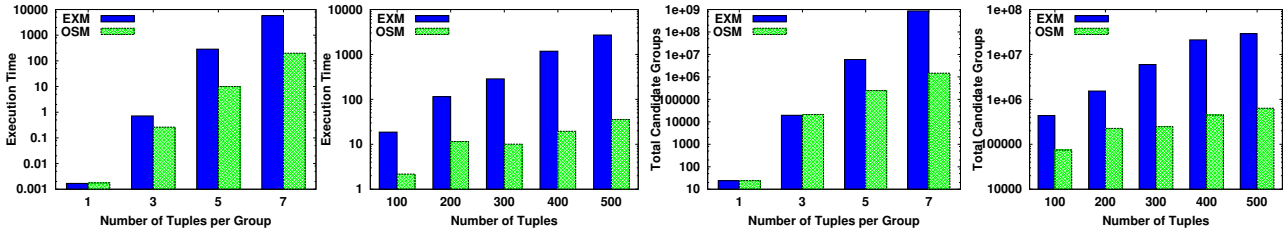
5.2 Experimental Results

Sample Resultant Skyline Groups: Table 3 shows several sample skyline 5-tuple groups under aggregate function SUM, from the 512-player NBA dataset. There are totally 83 players that are dominated by less than 5 other players and 5324 skyline 5-tuple groups. We see from the sample groups that they are formed by elite players and have different strengths. For instance, G1 is excellent in scoring (PPG), G2 excels in defense (RBG and BPG), and G3 is a very balanced group that is strong on many aspects although not the best on any dimension.

Size of Output under Different Functions: Table 4 shows, for different n, k, and aggregate functions, the number of all possible groups (G), the number of all skyline groups (S), and the number of distinct aggregated vectors (V) for the skyline groups. It can be seen that G quickly becomes very large (e.g., 30 million for n=500

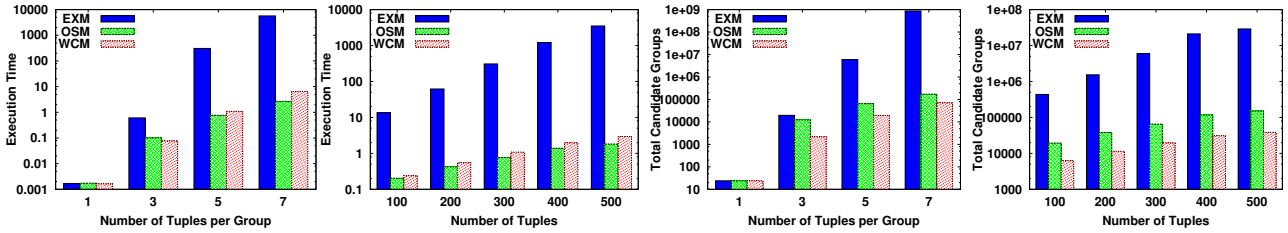
[4]The NBA dataset is collected from `http://www.databasebasketball.com/`.

[5]The stock dataset is collected from `http://pages.stern.nyu.edu/~adamodar/New_Home_Page/data.html`.



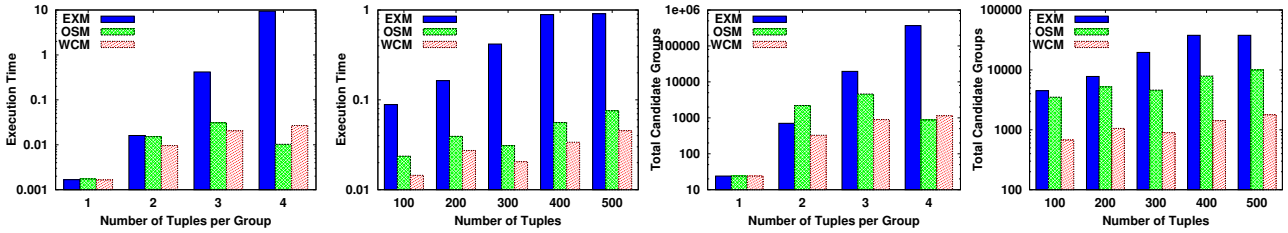
(a) $n = 300$ (b) $k = 5$ (c) $n = 300$ (d) $k = 5$

Figure 4: (a)-(b): Execution time (in seconds, logarithmic scale) and (c)-(d): number of candidate groups (logarithmic scale), SUM



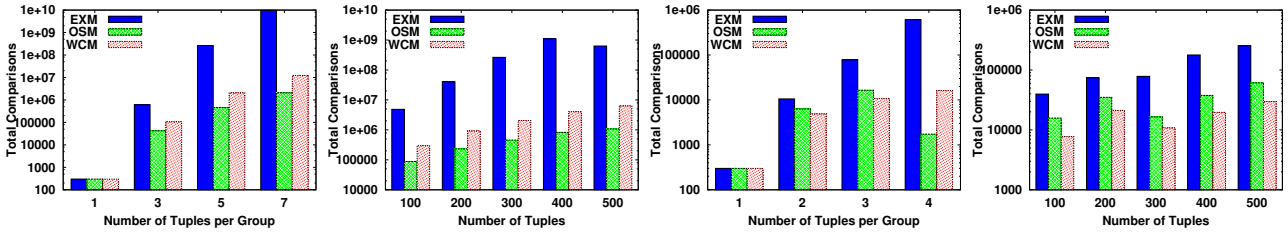
(a) $n = 300$ (b) $k = 5$ (c) $n = 300$ (d) $k = 5$

Figure 5: (a)-(b): Execution time (in seconds, logarithmic scale) and (c)-(d): number of candidate groups (logarithmic scale), MIN



(a) $n = 300$ (b) $k = 3$ (c) $n = 300$ (d) $k = 3$

Figure 6: (a)-(b): Execution time (in seconds, logarithmic scale) and (c)-(d): number of candidate groups (logarithmic scale), MAX



(a) $n = 300$ (b) $k = 5$ (c) $n = 300$ (d) $k = 3$

Figure 7: Number of pairwise group comparisons by different methods for MIN (a)-(b) and MAX (c)-(d)

and k=5), which indicates that EXM will suffer due to the large space of possible answers.

Among the 3 functions, in general SUM has the largest number of skyline groups and MAX results in the smallest output size. This is due to the intrinsic characteristics of these functions. In computing the aggregated vector for a group, SUM reflects the strength of all group members on each dimension. Hence it is more difficult for a group to dominate or equal to another group on every dimension. On the contrary, MIN (MAX) chooses the lowest (highest) value among group members on each dimension. Hence skyline groups are formed by relatively small number of phenomenal players.

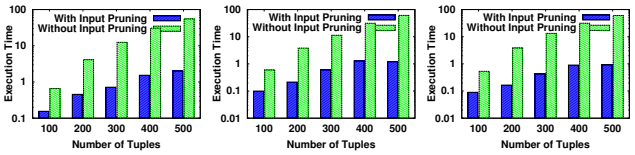
We also observed that it is rare under SUM and MIN to have multiple skyline groups sharing the same aggregated vector. With regard to MAX, there are such cases (e.g., n=100, k=3 and k=5). Moreover, under MAX, when group size k is larger than or equal to the number of dimensions (5, as mentioned in Section 5.1), all skyline groups have the same aggregated vector that attains the highest value on every dimension.

Proposed Methods vs. Exhaustive Method: Figure 4-6 show the execution time and number of generated candidate groups, by EXM/OSM/WCM under all applicable functions (SUM, MIN, MAX). Figure 7 further shows the number of pairwise group (aggregated vector) comparisons performed by these algorithms under MIN and MAX, to offer a closer look at their performance. In sub-figure (a) and (c) of these figures, we fix the size of dataset (n) to 300 tuples and vary group size (k). In sub-figure (b) and (d) of these figures, we fix the group size (k=5 for SUM/MIN and k=3 for MAX) and vary dataset size. We observed that OSM/WCM performed sub-

stantially (often orders of magnitude in execution time) better than the exhaustive method EXM. Due to the exhaustive nature of EXM, it produced much more candidate groups than OSM/WCM did and thus incurred much more pairwise group (aggregated vector) comparisons inside skyline function invocations.

n	$k=1$	$k=3$	$k=5$	$k=7$
100	19	31	37	44
200	22	37	47	57
300	24	50	61	67
400	29	62	78	86
500	30	62	83	94

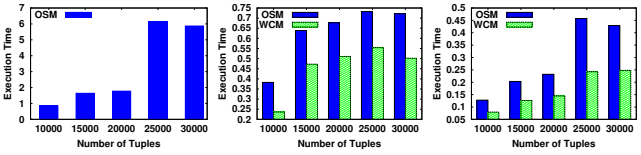
Table 5: Number of tuples that are dominated by less than k tuples



(a) SUM (b) MIN (c) MAX

Figure 8: Effect of input pruning on EXM, $k = 3$

Effect of Input Pruning: Input pruning had a good impact on the performance of all algorithms, since it significantly reduced the size of input. Table 5 shows that, in all considered cases, less than 100 tuples remained after k-dominator tuple pruning was applied. Figure 8 shows that substantial saving on execution time is achieved for all functions, even by the least efficient exhaustive method EXM.



(a) SUM (b) MIN (c) MAX

Figure 9: Execution time of OSM/WCM on stock dataset, $k = 3$

Search Space Pruning Power of OSM and WCM: Figure 5, 6 and 7 compare OSM and WCM, in terms of execution time, number of candidate groups produced, and number of pairwise group (aggregated vector) comparisons incurred. We observed that WCM performed better than OSM under MAX but OSM won for MIN on the NBA dataset. With regard to MAX, WCM demonstrated better pruning power in most cases because it resulted in both less candidate groups (Figure 6(c) and 6(d)) and less pairwise group comparisons (Figure 7(c) and 7(d)). With regard to MIN, even though WCM produced less candidate groups (Figure 5(c) and 5(d)), it required more group comparisons (Figure 7(a) and7(b)). Hence it lost in comparison with OSM under MIN for NBA dataset. However, another set of experiments actually showed that WCM outperformed OSM under both MIN and MAX on the larger stock dataset (Figure 9). (Due to space limitations, we are only showing the figures of execution time for experiments on the stock dataset, omitting detailed figures of number of candidate groups and group comparisons.)

These results can be explained by the analysis in Section 3.4.1. Recall that, to prune based on the order-specific property, OSM has to compute for every $h \in [k, n-k]$ the aggregate vectors of all skyline $1, 2, \ldots, \min(k, h)$-tuple groups over the first h tuples, because any of these groups may grow into a skyline k-tuple group when latter tuples are brought into consideration. Given a large n, the order-specific pruning process hence incurs a significant overhead. In contrast, the weak candidate-generation property exploited by WCM avoids the pitfall of order-specific property by removing the requirement of enumerating all tuples in order and generating skyline groups for each subset of tuples along the way. This explains why, for MIN, WCM outperformed OSM on the larger stock dataset but lost on the smaller NBA dataset.

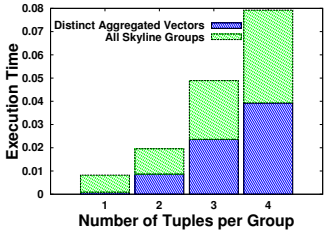


Figure 10: Finding all skyline groups from distinct aggregated vectors for MAX, n= 100, OSM

Cost of Postprocessing for Getting all Skyline Groups: Figure 10 shows the cost (in execution time) of postprocessing for obtaining all skyline groups from distinct aggregate vectors. Due to space limitations, we are only showing the results on the NBA dataset, for $n = 100$, MAX aggregate function, and OSM algorithm. We can see that in these configurations finding all skyline groups only doubled the execution time. This verifies that, even though the problem of finding all skyline groups from distinct aggregate vectors is an NP-hard problem, in practice it is usually efficient due to the small number of tuples that can "hit" MAX attribute values, as explained in Section 3.2.

6. RELATED WORK

Skyline query has been intensively studied over the last decade. Kung et al. [13] first proposed in-memory algorithms to tackle the skyline problem, which was named maximal vector problem in their work. Börzsönyi et al. [5] was the original work that studied how to process skyline queries in database systems. Since then, skyline query processing has been an active area of research and many techniques are developed. This line of research includes proposals of improved algorithms [10,12], progressive skyline computation [14,20,24], query optimization [8], to name just a few. Other variants of skyline queries have also been studied, including skyline cube which aims to answer skyline queries over any combination of dimensions [22, 28], skyline computation in data streams [17], uncertain databases [21], and low-cardinality domains [19], reverse skyline queries [11, 16], spatial skyline queries [23], privacy skyline [9], parallel and distributed computation [4, 27], etc.

We apply skyline algorithms over candidate groups to find skyline groups but our methods are orthogonal to specific choices of skyline algorithms. The one we currently implement is a simple nested-loop algorithm. It is equivalent to the block nested-loop algorithm in [5], when we do not consider external memory, i.e., when everything is done in main memory.

We are not aware of prior work on the notion of skyline groups proposed in this paper. Several prior works studied related but different concepts. The most related ones are [15] and [2]. They both consider the problem of forming expert teams to solve tasks. In [2], both team members and tasks are modeled as vectors in the same multi-dimensional skill space. A team is also an aggregated vector of its members. This paper is different from ours on two important aspects. (1) The goodness of teams is measured by how well they match the given tasks. Hence, teams are formed on a task-dependent basis. Our paper chooses groups based on how good

they are in an absolute sense, i.e., there is no given task. (2) The closeness between a team and a task is a scalar value, measured by a scoring function. Hence teams are ranked by their scores. On the contrary, we do not use a scoring function to combine the values on individual dimensions. Therefore the groups in our case are compared accroding to their skyline-based dominance relationship. Due to these differences, the techniques proposed there are not applicable to our setting. In [15], the skill space is Boolean, i.e., an individual skill is either present (required) in the profile of an expert (task) or not. Instead of deciding on how well teams match tasks, this work focuses on measuring if the members in a team can effectively collaborate with each other, based on information from social networks.

Moreover, in [3] groups defined by GROUP BY in SQL are compared with each other by their multiple aggregates and those groups that are not dominated by any other groups are called skyline groups. In our paper, the skyline groups are formed by combinations of k tuples in a tuple set. Zhang et. al. [29] studied set preferences where the preference relationships between k-subsets of tuples are based on features of k-subsets. The features are more general than numeric aggregate functions considered in this paper. The preferences given on each individual feature form a partial order over the k-subsets instead of a total order by numeric values. Zhang et. al. proposed two optimization techniques for set preference queries. The *superpreference* idea is based on an intuition similar to k-dominated pruning in this paper. The *M-relation* idea combines tuples that are equivalent under preference relationships. Pruning properties such as the ones proposed in this paper are not applicable for their set preference queries because of the more flexible semantics of their queries.

A large number of skyline points may exist in a given dataset, due to various reasons such as high dimensionality. Such large size of skyline hinders the usefulness of skyline to end users. Researchers have noticed this issue and various approaches are proposed to alleviate the problem. One direction is to perform skyline analysis in subspaces instead of the original full space [22, 26]. Moreover, Chan et. al. [7] propose to return only frequent points and they measure the frequency of a point by how often it is in the skyline of different subspaces. Lin et. al. [18] select k most representative points such that the total number of data points dominated by the k points is maximized. Tao et. al. [25] define representative skyline points differently, aiming at minimizing the maximal distance between non-representative skyline points and their closest representatives. Chan et. al. [6] define k-dominant skyline as the set of points that are not dominated by any other points in any k-attribute subspace. Our output compression is also for addressing the problem of large skyline, although the large skyline size is due to a reason specific to our problem setting– groups having identical aggregated vectors.

7. CONCLUSION

In this paper, we defined a novel problem of finding skyline groups which lends itself to many applications such as online fantasy sports games, expert finding, and project team formation. We developed novel algorithmic techniques on output compression, input pruning, and search space pruning to address the problem. In particular, for search space pruning, we identified a number of anti-monotonic properties to efficiently remove non-skyline groups from consideration. Based on the properties, we developed dynamic programming and iterative algorithms for skyline group search. Experimental results on real-world datasets verify that the proposed algorithms achieve orders of magnitude performance gain over the exhaustive method.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, 1994.

[2] A. Anagnostopoulos, L. Becchetti, C. Castillo, A. Gionis, and S. Leonardi. Power in unity: forming teams in large-scale community systems. In *CIKM*, 2010.

[3] S. Antony, P. Wu, D. Agrawal, and A. El Abbadi. Moolap: Towards multi-objective olap. In *ICDE*, 2008.

[4] W.-T. Balke, U. Gï£¡ntzer, and J. Zheng. Efficient distributed skylining for web information systems. In *EDBT*. 2004.

[5] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.

[6] C. Chan, H. Jagadish, K. Tan, A. Tung, and Z. Zhang. Finding k-dominant skylines in high dimensional space. In *SIGMOD*, 2006.

[7] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. Tung, and Z. Zhang. On high dimensional skylines. In *EDBT*. 2006.

[8] S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. In *ICDE*, 2006.

[9] B.-C. Chen, K. LeFevre, and R. Ramakrishnan. Privacy skyline: privacy with multidimensional adversarial knowledge. In *VLDB*, 2007.

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

[11] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *VLDB*, 2007.

[12] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *VLDB*, 2005.

[13] H.T.Kung, F.Luccio, and F.P.Preparata. On finding the maxima of a set of vectors. *JACM*, 22(4), 1975.

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

[15] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *KDD*, 2009.

[16] X. Lian and L. Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *SIGMOD*, 2008.

[17] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: efficient skyline computation over sliding windows. In *ICDE*, 2005.

[18] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: The k most representative skyline operator. In *ICDE*, 2007.

[19] M. Morse, J. M. Patel, and H. V. Jagadish. Efficient skyline computation over low-cardinality domains. In *VLDB*, 2007.

[20] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1), 2005.

[21] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *VLDB*, 2007.

[22] J. Pei, Y. Yuan, X. Lin, W. Jin, M. Ester, Q. Liu, W. Wang, Y. Tao, J. X. Yu, and Q. Zhang. Towards multidimensional subspace skyline analysis. *TODS*, 31(4), 2006.

[23] M. Sharifzadeh and C. Shahabi. The spatial skyline queries. In *VLDB*, 2006.

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

[25] Y. Tao, L. Ding, X. Lin, and J. Pei. Distance-based representative skyline. In *ICDE*, 2009.

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

[27] P. Wu, C. Zhang, Y. Feng, B. Zhao, D. Agrawal, and A. El Abbadi. Parallelizing skyline queries for scalable distribution. In *EDBT*. 2006.

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

[29] X. Zhang and J. Chomicki. Preference queries over sets. In *ICDE*, 2011.