EFFICIENT EVALUATION OF CONTEXTUAL AND REVERSE PARETO-OPTIMALITY QUERIES

by

AFROZA SULTANA

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
MAY 2018

To my parents, Dr. Md. Abdullah-Hel-Kafi and Dr. Aklima Khatun.

To my brother, Ahsanul Arefin (Kallol).

ACKNOWLEDGEMENTS

First, I would like to express my sincerest gratitude and utter respect to my supervisor Dr. Chengkai Li, Associate Professor, Associate Chair of the Department of Computer Science and Engineering, and Director of the Innovative Database and Information Systems Research Laboratory (IDIR) at the University of Texas at Arlington, for his continuous support throughout of my Ph.D. journey, for his patience, generosity, guidance, motivation, and immense knowledge. I am very fortunate to have him as my mentor. He relied on my strength and interest. He taught me to speak up. He encouraged me in defining new problems and had long discussions with me in this regard. The research in this dissertation would not have been possible without his constant supervision. The learning was not limited to study or research; I believe his mentoring improved me in many aspects.

I would like to extend my gratitude to the rest of my Ph.D. dissertation committee: Dr. Sharma Chakravarthy, Dr. Gautam Das, Dr. Jun Yang, and Dr. Cong Yu for their insightful comments and encouragement.

This work would not have been possible without the financial support of Department of Computer Science and Engineering, the University of Texas at Arlington. I am especially grateful to Dr. Bahram Khalili, Graduate Advisor, who eased difficulties in many instances. I thank Administrative Staffs Pamela McBride and Sherri Gotcher for their time and cooperation. I wish to thank my labmates of IDIR Lab and friends from UT Arlington for assisting me in different aspects.

I appreciate all the teachers who taught me during the years I spent in school, first in Bangladesh Community School, Tripoli, then in Viqarunnisa Noon College, then in Bangladesh University of Engineering and Technology (BUET) and finally in this university at the United States. I am thankful to my undergraduate thesis supervisor Dr. Mahmuda Naznin for discovering my abilities and encouraging me to pursue graduate studies in the USA.

Special thanks to the Doctors and Nurses of the UTA Health Center who made it possible for me to concentrate on my work.

Words cannot express how indebted I am to my parents, Dr. Md. Abdullah-Hel-Kafi and Dr. Aklima Khatun as well as my brother, Ahsanul Arefin (Kallol) for their illimitable patience, sacrifices, support, love, and encouragement throughout my life. I am thankful to my adorable nephew Md. Abdullah-Al-Shams (Kiron) for being my source of peace and joy. I appreciate my sister-in-law Asifa Hossain (Jyoti) for her support.

Finally, I thank Almighty for giving me the endurance, patience, and perseverance to successfully complete my degree.

APRIL 5, 2018

ABSTRACT

EFFICIENT EVALUATION OF CONTEXTUAL AND REVERSE PARETO-OPTIMALITY QUERIES

AFROZA SULTANA, Ph.D.

The University of Texas at Arlington, 2018

Supervising Professor: Dr. Chengkai Li

Many real-world applications analyze data to find objects that “stand out” with regard to various contexts and ways of valuing the objects. Examples of such application scenarios include vendors recommending products to potential customers, social networks improving content selection for users, and Google Scholar notifying newly published articles based on profiles. Besides, journalists identify conditions to substantiate the significance of an event or the interestingness of an object. Interesting events can be retrieved from stock data, weather data, and criminal records. Apart from journalists, those events convey significant information for financial analysts, scientist, and citizens.

The aforementioned application needs can be modeled as contextual and reverse Pareto-optimality queries. Given a set of objects and a way of valuing them, a Pareto-optimality query results in a set of objects of which each resulting object is not worse than any other object in the set. The resulting objects are called Pareto-optimal objects or Pareto frontier. A “contextual” Pareto-optimal object stands out against other objects in a context. A reverse Pareto-optimality query finds contexts where a given object belongs to the Pareto frontiers.

We formalize the real-world problem of finding outstanding objects as finding Pareto frontiers. In an ever-growing database, a straightforward brute-force approach to answering the contextual and reverse Pareto-optimal queries would compute Pareto frontiers in every context with regard to each existing object, separately. To resolve this drawback, we pursue object reduction, context pruning, and sharing computation across ways of valuing. In this dissertation, we study efficient evaluation of contextual and reverse Pareto-optimality queries.

We first discuss methods for efficient evaluation of contextual and reverse Pareto-optimality queries in the context of modeling and satisfying user preferences. Specifically, we study the problem of continuous object dissemination—given a large number of users and continuously arriving new objects, deliver an incoming object to all users who prefer the object. Many real-world applications analyze users' preferences for effective object dissemination. For continuously arriving objects, timely finding users who prefer a new object is challenging. We consider an append-only table of objects with multiple attributes and users' preferences on individual attributes are modeled as strict partial orders. An object is preferred by a user if it belongs to the Pareto frontier with respect to the user's partial orders. Users' preferences can be similar. Exploiting shared computation across similar preferences of different users, we design algorithms to find target users of a new object. In order to find users of similar preferences, we study the novel problem of clustering users' preferences that are represented as partial orders. We also present an approximate solution to the problem which is more efficient than the exact one while ensuring sufficient accuracy. Furthermore, we extend the algorithms to operate under the semantics of sliding window. We present the results from comprehensive experiments for evaluating the efficiency and effectiveness of the proposed techniques.

We further discuss efficient evaluation of contextual and reverse Pareto-optimality queries in the form of finding new, prominent situational facts, which are emerging statements about objects that stand out within certain contexts. Many such facts are newsworthy—e.g., an athlete's outstanding performance in a game, or a viral video's impressive popularity. Automated identification of these facts assists journalists in reporting. More specifically, we consider an ever-growing table of objects with dimension and measure attributes. A situational fact is a "contextual" skyline tuple that stands out against historical tuples in a context specified by a conjunctive constraint involving dimension attributes, while the objects are being compared on a set of measure attributes. New tuples are constantly added to the table, reflecting events happening in the real world. Our goal is to discover constraint-measure pairs that qualify a new tuple as a contextual skyline tuple and discover them quickly before the event becomes yesterday's news. A brute-force approach requires exhaustive comparison of the new tuple with every existing tuple, under every constraint, and in every measure subspace. We design algorithms in response to these challenges using three corresponding ideas—tuple reduction, constraint pruning, and sharing computation across measure subspaces. We further extend the algorithms to allow for updates on data. We also adopt a simple prominence measure to rank a large number of discovered facts. Experiments over three real datasets validate the effectiveness and efficiency of our techniques.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

Introduction

Nowadays, many real-world applications analyze data in order to find objects that "stand out" with regard to various contexts and ways of valuing the objects. Examples of such application scenarios are not limited to vendors recommending products to target markets and journalists identifying conditions to substantiate the significance of an event or the prominence of an object. Such application needs can be interpreted as *contextual* and *reverse Pareto-optimality queries*. *Pareto-optimization* [1] is a well-studied term named after Vilfredo Pareto, since the concept was applied in economic efficiency and income distribution. After that, it has been explored in other academic fields including computer science. Given a set of objects and a way of valuing them, a Pareto-optimality query finds a set of objects of which each resulting object is not worse than any other object in the set [1]. The query result set is known as *Pareto-optimal objects* or *Pareto frontier* [1].

A *contextual Pareto-optimal object* is not worse than any other object in a context when a way of valuing is considered. Consider a set of contexts $\mathcal{C}$ and a table of objects $\mathcal{O}$ that are described by a set of attributes $\mathcal{D}$. An object o is *Pareto-optimal* with respect to $c \in \mathcal{C}$, if no other object in $\mathcal{O}$ is better than it. In this dissertation, we define *reverse Pareto-optimality query* as finding the contexts where a certain object belongs to the Pareto frontier. Formally, given an object o, the reverse Pareto-optimality query results in $C \subseteq \mathcal{C}$ such that o belongs to the Pareto frontiers for all $c \in C$. To compute reverse Pareto-optimality query, we further study *contextual Pareto-optimality query* which finds the contextual Pareto-optimal objects under a context $c \in \mathcal{C}$.

In an ever-growing database table, new objects are constantly appended to the table and thus Pareto-optimal objects are updated frequently. We aim at finding contexts where a new object acquires Pareto-optimality. A straightforward brute-force approach to answering the contextual and reverse Pareto-optimal queries would compute Pareto frontier under every context with regard to each object, separately. In case of large number of contexts along with streaming objects, the challenge lies in efficient computation. To resolve this issue, we pursue object reduction, context pruning, and sharing computation across ways of valuing. In this dissertation, we study efficient evaluation of contextual and reverse Pareto-optimality queries with regard to two types of queries, namely *preference query* [25, 12] and *skyline query* [9].

1.1 Pareto-optimality with Regard to Preference Query

Most of the existing studies on Pareto-optimality queries assume numeric attributes only. Kießling [25] defined preferences as *strict partial orders* where preference query returns the best matches. Given a preference, a Pareto-optimal object is not preferred over by any other object. Formally, given a set of attributes $d \in \mathcal{D}$, a preference c on $d \in \mathcal{D}$ is a strict partial order $\succ_c^d$, where $\succ_c^d$ is a binary relation over $dom(d)$—the domain of d. "$(x, y) \in \succ_c$" is interpreted as "c prefers x to y". Given a table of objects $\mathcal{O}$, with regard to c, any object o in the preference query result tells the absence of o' such that "c prefers o' to o". With this schema, a preference corresponds to a context. Therefore, a reverse Pareto-optimality query finds the preferences where a certain object acquires Pareto-optimality. In addition, a contextual Pareto-optimality query results in Pareto frontier with regard to the corresponding preference.

In Chapter 2, we study the problem of continuous object dissemination and formalize it as finding Pareto-optimal objects regarding strict partial orders. Given a large number of users and continuously arriving objects, our goal is to swiftly disseminate a newly arrived object to a user if the user's preferences—modeled as strict partial orders on individual attributes—approve the object as Pareto-optimal. We devise efficient solutions exploiting shared computation across similar preferences of different users. We study the novel challenge of clustering user preferences represented as strict partial orders. Particularly we design similarity measures for such preferences. To address performance degradation due to small clusters, we present an approximate similarity measure that achieves high efficiency and accuracy of answers. We extend our proposed solutions to deal with Pareto frontier maintenance under sliding window. We conduct extensive experiments using simulations on two real datasets (a movie dataset and a publication dataset). The results demonstrate clear strengths of our solutions in comparison with baselines, in terms of execution time and efficacy.

1.2 Pareto-optimality with Regard to Skyline Query

In recent years, skyline query becomes a well-explored perspective of Pareto-optimality query, since Börzsönyi et al. [9] brought the concept to the database field. Given a set of tuples R as well as a set of measure attributes $M \subseteq \mathcal{M}$ describing them, a tuple t' *dominates* t if t' is better than or equal to t on every measure attribute in that combination and better than t on at least one of the attribute. A tuple t is a *skyline tuple* in M, if it is not *dominated by* any other tuple in R. In this case, each tuple represents an object and the combination of measure attributes interprets the way of valuing the objects. Let incorporate a set of *dimension* attributes with this schema. Thereafter, a context is a conjunctive constraint defined on a subset of the dimension attributes $D \subseteq \mathcal{D}$. A *contextual skyline tuple* t is not dominated by any other tuple in that context. In this scenario, a reverse Pareto-optimality query finds the contexts where a given tuple belongs to the skyline. Besides, a contextual Pareto-optimality query results in skyline tuples under the constraint.

In Chapter 3, we study the novel problem of finding situational facts and formalize it as discovering constraint-measure pairs that qualify a tuple as a contextual skyline tuple. We devise efficient algorithms based on three main ideas—tuple reduction, constraint pruning and sharing computation across measure subspaces. We further extend the algorithms to consider an instant dataset as input and thus allow live news streaming. We use a simple prominence measure for ranking situational facts and discovering prominent situational facts. We conduct extensive experiments on three real datasets (two NBA datasets and weather dataset) to investigate their prominent situational facts and to study the efficiency of various proposed algorithms and their tradeoffs.

CHAPTER 2

Continuous Object Dissemination

2.1 Overview

Many applications serve users better by disseminating objects to the users according to their preferences. User preferences can be modeled via a variety of means including *collaborative filtering* [33], *top-k ranking* [16, 17], *skyline* [9], and general *preference queries* [25, 12]. In various scenarios, users' preferences stand or only change occasionally, while the objects keep coming continuously. Such scenarios warrant the need for a capability of continuous monitoring of preferred objects. While previous studies have made notable contributions on continuous evaluation of skyline [44, 27] and top-*k* queries [49], we note that two important considerations are missing from prior works:

- *Many users*: There may be a large number of users and the users may have similar preferences. Prior studies focus on the query needs of one user and thus their algorithmic solutions can only be applied separately on individual users. A solution can potentially attain significant query performance gain by leveraging users' *common preferences*.
- *Partially ordered attributes*: Prior works focus on top-*k* and skyline queries. In multi-objective optimization, a more general concept than skyline is *Pareto frontier*. Consider a table of objects with a set of attributes. An object is *Pareto-optimal* (i.e., it belongs to the Pareto frontier) if and only if it is not dominated by any other object [6, 26]. Object *y* dominates *x* if and only if *y* is better than or equal to *x* on every attribute and is better on at least one attribute. In defining the *better-than* relations, most studies on skyline queries assume a total order on the ordinal or numeric values of an attribute, except for [31, 52] which consider strict partial orders. The psychological nature of human's preferences

determines that it is not always natural to enforce a total order. Oftentimes real-world preferences can only be modeled as strict partial orders [25, 12, 31].

Consider the following motivating applications which monitor Pareto frontiers on partially ordered attributes for many users.

- Social network content and news delivery: It is often impossible and unnecessary for a user to keep up with the plethora of updates (e.g., news feeds in Facebook) from their social circles. When a new item is posted, if the item is Pareto-optimal with respect to a user, it can be displayed above other updates in the user's view. Similar ideas can be adopted by mass media to ensure their news reaches the right audience. User preferences can be modeled on content creator, topic, location, and so on. Enforcing total orders on such attributes is both cumbersome and unnatural.
- Publication alerts: Bibliography servers such as PubMed and Google Scholar can notify users about newly published articles matching their preferences on venues and keywords. Such attributes do not welcome total orders either.
- Product recommendation: When a new product becomes available, a retailer can notify customers who may be interested. It can distill customers' preferences on product specifications (e.g., brand, display and memory for laptops) from profiles, past transactions and website browsing logs. Example 1 discusses this application more concretely.

Example 1. Consider an inventory of laptops in Table 2.1 and customers' preferences on individual product attributes (display, brand and CPU) modeled as strict partial orders in Table 2.2. For an attribute, the corresponding strict partial order is depicted as a directed acyclic graph (DAG), more specifically a Hasse diagram. Given two values x and y in the attribute's domain, the existence of a path from x to y in the DAG implies that x is preferred to y. With respect to customer c_1 and attribute brand, the path from *Lenovo* to *Toshiba* implies that c_1 prefers *Lenovo* to *Toshiba*. There is no path between *Toshiba* and *Samsung*, which indicates c_1 is indifferent between the two brands.

The strict partial orders on various attributes together represent a customer's preferences on objects. For instance, c_1 prefers o_2=$\langle 14$, *Apple*, *dual*$\rangle$ to o_1=$\langle 12$, *Apple*, *single*$\rangle$, since they prefer $13-15.9$ to $10-12.9$ on display and *dual* to *single* on CPU. With regard to o_1 and o_3=$\langle 15$, *Samsung*, *dual*$\rangle$, c_1 does not prefer one over the other because, though they prefer $13-15.9$ to $10-12.9$ and *dual* to *single*, they prefer *Apple* to *Samsung* on brand.

According to the data in Tables 2.1 and 2.2, if the existing products are o_1 to o_{14} (ignore o_{15} and o_{16} for now), the Pareto frontiers of c_1 and c_2 are $\{o_2\}$ and $\{o_2, o_3\}$, respectively. Suppose o_{15}=$\langle 16.5$, *Lenovo*, *quad*$\rangle$ just becomes available. For c_1, o_{15} does not belong to the Pareto frontier. It is dominated by o_2, because c_1 prefers 14-inch display over 16.5-inch, *Apple* over *Lenovo*, and *dual*-core CPU over *quad*-core CPU. However, o_{15} is a Pareto-optimal object for c_2 since it is not dominated by any other object according to c_2's preferences. It is thus recommended to c_2, and the Pareto frontier of c_2 is updated to $\{o_2$, o_3, $o_{15}\}$. $\triangle$

This chapter **formulates the problem of *continuous monitoring of Pareto frontiers***: given a large number of users and continuously arriving new objects, for each newly arrived object, discover all users for whom the object is Pareto-optimal. Users' preferences are modeled as strict partial orders, one for each attribute domain of the objects.

It is key to devise an efficient approach to this problem. The value of a Pareto-optimal object diminishes quickly; the earlier it is found to be worth recommendation, the better. For instance, a status update in a social network keeps getting less relevant since the moment it is posted; a customer's need for a product may be fulfilled by a less preferred choice, if an even better option was not shown to the customer in time.

A simple, brute-force approach is to, given a newly arrived object, compute for every user if the object belongs to the Pareto frontier with respect to the user's preferences. This

	display	brand	CPU
o_1	12	*Apple*	*single*
o_2	14	*Apple*	*dual*
o_3	15	*Samsung*	*dual*
o_4	19	*Toshiba*	*dual*
o_5	9	*Samsung*	*quad*
o_6	11.5	*Sony*	*single*
o_7	9.5	*Lenovo*	*quad*
o_8	12.5	*Apple*	*dual*
o_9	19.5	*Sony*	*single*
o_{10}	9.5	*Lenovo*	*triple*
o_{11}	9	*Toshiba*	*triple*
o_{12}	8.5	*Samsung*	*triple*
o_{13}	14.5	*Sony*	*dual*
o_{14}	17	*Sony*	*single*
o_{15}	16.5	*Lenovo*	*quad*
o_{16}	16	*Toshiba*	*single*

Table 2.1: Product table.

entails continuous maintenance of Pareto frontier for each and every user. The brute-force approach is subject to a clear drawback—repeated and wasteful maintenance of Pareto frontier for every user.

Sharing computation across users To tackle the aforementioned drawback, we partly resort to sharing computation across users. The challenge lies in the diversity of corresponding partial orders—a Pareto-optimal object with respect to one user may or may not be in the Pareto frontier for another user. Nonetheless, users have common preferences. In Table 2.2, both c_1 and c_2 prefer $13 - 15.9$ inch display the most. Both prefer *Apple* and *Lenovo* to *Toshiba* and *Sony*, and they both prefer *single*-core CPU the least. In Table 2.2, U is a *virtual user* whose partial orders depict the common preferences of c_1 and c_2. Intuitively, users having similar preferences can be clustered together.

We thus design algorithms to mitigate repetitive computation via sharing computation across similar preferences of users. To intuitively understand the idea, consider two

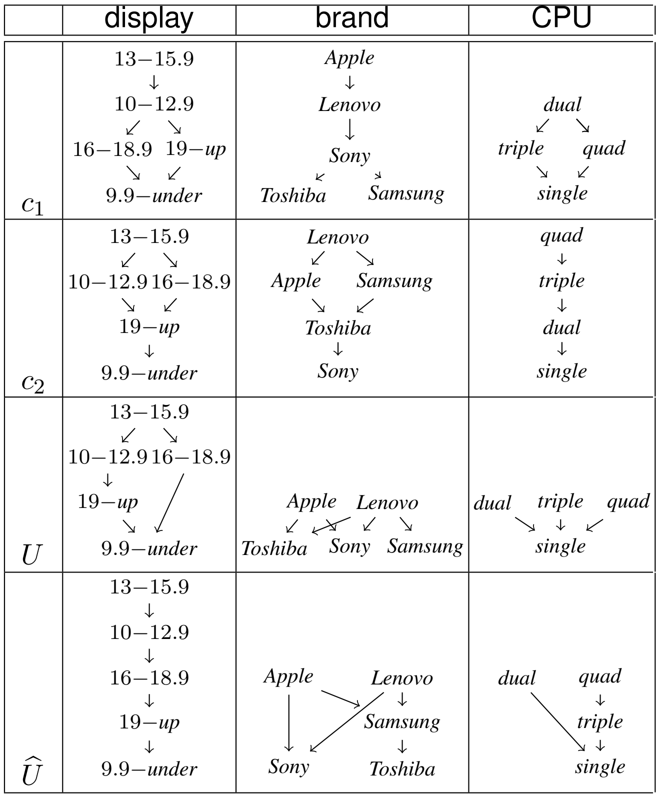


Table 2.2: User preferences. U=$\{c_1$,$c_2\}$.

example scenarios. i) If o is dominated by o' with respect to the common preferences of a set of users, then o is disqualified in Pareto-optimality for all users in the set. In Example 1, consider o_{16}=$\langle 16, \textit{Toshiba}, \textit{single}\rangle$ as the new object. With respect to U, o_{16} is dominated by both o_2=$\langle 14, \textit{Apple}, \textit{dual}\rangle$ and o_{15}=$\langle 16.5$, *Lenovo*, *quad*$\rangle$. Therefore, o_{16} belongs to the Pareto frontier of neither c_1 nor c_2. ii) Before the arrival of o_2, obviously o_1=$\langle 12$, *Apple*, *single*$\rangle$ is the only Pareto-optimal object for U, c_1 and c_2. Now consider the entrance of o_2. As o_1 is dominated by o_2 with respect to U, o_1 is replaced by o_2 in the Pareto frontier. This comparison is sufficient to decide that o_1 is dominated by o_2 for both c_1 and c_2.

Clustering users To find users sharing similar preferences, we study the novel problem of clustering strict partial orders, which are used to model the preferences of both users and clusters. We measure the similarity between clusters and users by their common preferences. Such similarity measures factor in the different significance of preferences at

$\begin{tikzcd}[column sep=small] \textit{Apple} \arrow[d] & & \textit{Toshiba} \arrow[ddl] \\ \textit{Lenovo} \arrow[dr] & & \\ & \textit{Samsung} & \end{tikzcd}$ c_1	$\begin{tikzcd}[column sep=small] \textit{Apple} \arrow[dr] & & \textit{Toshiba} \arrow[dl] \\ & \textit{Lenovo} \arrow[d] & \\ & \textit{Samsung} & \end{tikzcd}$ c_2	$\begin{tikzcd}[column sep=small] \textit{Apple} \arrow[d] & & \textit{Toshiba} \arrow[ddl] \\ \textit{Lenovo} \arrow[dr] & & \\ & \textit{Samsung} & \end{tikzcd}$ U_1
$\begin{tikzcd}[column sep=small] & \textit{Samsung} \arrow[d] & \\ & \textit{Lenovo} \arrow[dl] \arrow[dr] & \\ \textit{Apple} & & \textit{Toshiba} \end{tikzcd}$ c_3	$\begin{tikzcd} \textit{Samsung} \arrow[d] \\ \textit{Lenovo} \arrow[d] \\ \textit{Apple} \arrow[d] \\ \textit{Toshiba} \end{tikzcd}$ c_4	$\begin{tikzcd}[column sep=small] & \textit{Samsung} \arrow[d] & \\ & \textit{Lenovo} \arrow[dl] \arrow[dr] & \\ \textit{Apple} & & \textit{Toshiba} \end{tikzcd}$ U_2
$\begin{tikzcd}[column sep=small] & \textit{Lenovo} \arrow[dl] \arrow[dr] & \\ \textit{Apple} \arrow[dr] & & \textit{Toshiba} \arrow[dl] \\ & \textit{Samsung} & \end{tikzcd}$ c_5	$\begin{tikzcd}[column sep=small] & \textit{Lenovo} \arrow[d] & \\ & \textit{Apple} \arrow[dl] \arrow[dr] & \\ \textit{Toshiba} & & \textit{Samsung} \end{tikzcd}$ c_6	$\begin{tikzcd}[column sep=small] & \textit{Lenovo} \arrow[dl] \arrow[dr] & \\ \textit{Apple} \arrow[d] & & \textit{Toshiba} \\ \textit{Samsung} & & \end{tikzcd}$ U_3

Table 2.3: User preferences with respect to brand. U_1=$\{c_1$,$c_2\}$, U_2=$\{c_3$,$c_4\}$, U_3=$\{c_5$,$c_6\}$.

various levels of the partial orders. Table 2.3 depicts six customers' preferences on brand, in which c_4, c_5, and c_6 prefer *Lenovo* to all other brands except that c_4 prefers *Samsung* over *Lenovo*. Consider the objects in Table 2.1. For both c_5 and c_6, the Pareto frontiers contain $\{o_7, o_{10}, o_{15}\}$, while c_4 has $\{o_3, o_5, o_{12}\}$ as its Pareto frontier. We can say that c_5 and c_6 are more similar than c_4 and c_5 or c_4 and c_6.

Approximation The clustering algorithm may produce clusters that comprise few users, due to diverse preferences. With small clusters, the shared computation mentioned above may not pay off its overhead. Our response to this challenge is to use approximation. As in many data retrieval scenarios, insisting on exact answers is unnecessary and answers in close vicinity of the exact ones can be just good enough. Specifically, given a set of users, if a sizable subset of the users agree with a preference, the preference can be considered an approximate common preference. This relaxation eases the aforementioned concern regarding small clusters as more approximate common preferences lead to larger clusters. As an example, in Table 2.2, while c_2 does not share with c_1 the preference of *Apple* over *Samsung*, its preference does not oppose it either. We can consider "*Apple* over *Samsung*" as an *approximate common preference*. A possible set of approximate common preferences of c_1 and c_2 form the strict partial orders in the row for virtual user $\widehat{U}$.

Alive objects Objects can have limited lifetime. The trends in social networks and news media change rapidly. Similarly, in any inventory, products become unavailable over time. In these scenarios users look for *alive* objects only. To meet this real-world requirement, we further extend our algorithms to operate under the semantics of a *sliding window* and thus to disseminate an object only during its lifespan.

In summary, the contributions of this chapter are as follows:

- We study the problem of continuous object dissemination and formalize it as finding Pareto-optimal objects regarding partial orders. Given a large number of users and continuously arriving objects, our goal is to swiftly disseminate a newly arrived object to a user if the user's preferences—modeled as strict partial orders on individual attributes—approve the object as Pareto-optimal (Section 2.3).
- We devise efficient solutions exploiting shared computation across similar preferences of different users (Section 2.4).
- We study the novel challenge of clustering user preferences represented as strict partial orders. Particularly we design similarity measures for such preferences (Section 2.5).
- To address performance degradation due to small clusters, we present an approximate similarity measure that achieves high efficiency and accuracy of answers (Section 2.6).
- We extend our proposed solutions to deal with Pareto frontier maintenance under sliding window (Section 2.7).
- We conduct extensive experiments using simulations on two real datasets (a movie dataset and a publication dataset). The results demonstrate clear strengths of our solutions in comparison with baselines, in terms of execution time and efficacy (Section 2.8).

2.2 Related Work

Pareto-optimality is a subject of extensive investigation. Its study in the computing fields can be dated back to *admissible points* [6] and *maximal vectors* [26]. Börzsönyi et al. [9] introduced the concept of skyline—a special case of Pareto frontier—in which all attributes are numeric and amenable to total orders. Kießling [25] defined preferences as strict partial orders on which preference queries operate. After that, several studies specialized on skyline query evaluation over categorical attributes [10, 32, 31, 52], among which [32, 31, 52] particularly considered query answer maintenance and only [31, 52] allow partial orders on attribute values. Nevertheless, they all consider only one user and none utilizes shared computation across multiple users' partial orders.

Given a set of objects, Wong et al. [42, 41, 43] identify the minimum set of preference relations that preclude an object from being in the Pareto frontier. This minimum set is the combination of each possible preference relation with regard to the values of all unique objects in the set. In case of any update in the object set, the minimum disqualifying condition must be recomputed. Hence, it is not designed for continuously arriving objects.

Vlachou et al. [38, 39] and Yu et al. [49] aimed at finding all users who view a given object as one of their top-k favourites, i.e., the results of a reverse top-k query. Dellis et al. [15] studied reverse skyline query—selecting users to whom a given object is in the skyline. These works consider only numeric attributes. There is no clear way to extend them for categorical attributes or even partial orders.

All these studies, while about object dissemination, focused on different aspects of the problem than ours. In this chapter, we study Pareto frontier maintenance while exploiting shared computation across users' preferences. Besides, as explained in [35], no prior work studied similarity measures for partial orders or how to cluster partial orders.

2.3 Problem Statement

$\mathcal{O}$	set of objects
$d \in \mathcal{D}$	attribute
$c \in \mathcal{C}$	user
$\succ_c^d$	binary relation over $dom(d)$ with regard to c's preference
$o' \succ_c o$	c prefers o' to o
$\mathcal{P}_c$	the Pareto frontier with regard to c
$\mathcal{C}_o$	the target users of o
$U \subseteq \mathcal{C}$	set of users
$sim(U_1, U_2)$	the similarity measure between two clusters U_1 and U_2
S_U^d	the maximal values of $\succ_U^d$
h	branch cut in dendrogram

Table 2.4: Notations.

This section provides a formal description of our data model and problem statement. Table 3.3 lists the major notations. Consider a set of users $\mathcal{C}$ and a table of objects $\mathcal{O}$ that are described by a set of attributes $\mathcal{D}$. For each user $c \in \mathcal{C}$, their preference regarding $\mathcal{O}$ is represented by strict partial orders. For each attribute $d \in \mathcal{D}$, the strict partial order corresponding to c's preference on d is a binary relation over $dom(d)$—the domain of d, as follows.

Definition 1 (Preference Relation and Tuple)**.** Given a user $c \in \mathcal{C}$ and an attribute $d \in \mathcal{D}$, the corresponding *preference relation* is denoted $\succ_c^d$. For two attribute values $x, y \in dom(d)$, if (x, y) belongs to $\succ_c^d$ (i.e., $(x, y) \in \succ_c^d$, also denoted $x \succ_c^d y$), it is called a *preference tuple*. It is interpreted as "user c prefers x to y on attribute d". A preference relation is irreflexive ($(x, x) \notin \succ_c^d$) and transitive ($(x, y) \in \succ_c^d \wedge (y, z) \in \succ_c^d \Rightarrow (x, z) \in \succ_c^d$), which together also imply asymmetry ($(x, y) \in \succ_c^d \Rightarrow (y, x) \notin \succ_c^d$). $\triangle$

Definition 2 (Object Dominance)**.** A user c's preferences regarding all attributes induce another strict partial order $\succ_c$ that represents c's preferences on objects. Given two objects $o, o' \in \mathcal{O}$, c prefers o' to o if o' is identical or preferred to o on all attributes and o' is preferred to o on at least one attribute. More formally, $o' \succ_c o$ (called o' *dominates* o), if and only if $(\forall d \in \mathcal{D} : o.d = o'.d \vee o'.d \succ_c^d o.d) \wedge (\exists d \in \mathcal{D} : o'.d \succ_c^d o.d)$. If $(\forall d \in \mathcal{D} : o.d = o'.d)$, we say that o and o' are *identical*, denoted as $o = o'$. △

Definition 3 (Pareto Frontier)**.** An object o is *Pareto-optimal* with respect to c, if no other object in $\mathcal{O}$ dominates it. The set of *Pareto-optimal objects* (i.e., the *Pareto frontier*) in $\mathcal{O}$ for c is denoted $\mathcal{P}_c$, i.e., $\mathcal{P}_c = \{o \in \mathcal{O} | \nexists o' \in \mathcal{O} \text{ s.t. } o' \succ_c o\}$. Note that the concept of skyline points [9] is a specialization of the more general Pareto frontier, in that the preference relations for skyline points are defined as total orders (with ties) instead of general strict partial orders. △

Definition 4 (Target Users)**.** Given an object o, the set of all users for whom o belongs to their Pareto frontiers are called the *target users*. The target user set is denoted $\mathcal{C}_o$, i.e., $\mathcal{C}_o = \{c \in \mathcal{C} | o \in \mathcal{P}_c\}$. △

Example 2. Consider Table 2.1 and Table 2.2. $\mathcal{O} = \{o_1, o_2, \ldots, o_{15}\}$ (ignore o_{16} for now), $\mathcal{C} = \{c_1, c_2\}$, and $\mathcal{D} = \{\textsf{display, brand, CPU}\}$. With respect to c_1, (10−12.9, 16−18.9), (*Apple*, *Samsung*) and (*dual*, *triple*) are some of the preference tuples on attributes display, brand and CPU, respectively. Similarly, for c_2, (16−18.9, 19−up), (*Toshiba*, *Sony*) and (*triple*, *dual*) are some sample preference tuples.

$\mathcal{P}_{c_1} = \{o_2\}$, since all other objects are dominated by o_2 with respect to c_1. $\mathcal{P}_{c_2} = \{o_2, o_3, o_{15}\}$, as o_2, o_3 and o_{15} dominate $\{o_1, o_4, o_6, o_8, o_9, o_{13}\}$, $\{o_4, o_6, o_8, o_{13}\}$ and $\{o_4, o_5, o_7, o_{10}, o_{11}, o_{12}, o_{14}\}$, respectively. Therefore, $\mathcal{C}_{o_2} = \{c_1, c_2\}$ and $\mathcal{C}_{o_3} = \mathcal{C}_{o_{15}} = \{c_2\}$. Objects other than o_2, o_3, o_{15} do not have target users in $\mathcal{C}$, i.e., $\mathcal{C}_o = \phi, \forall o \in \mathcal{O} - \{o_2, o_3, o_{15}\}$. △

Problem Statement The problem of *continuous monitoring of Pareto frontiers* is, given a set of users $\mathcal{C}$, their preference relations on attributes $\mathcal{D}$, and a set of continuously growing objects $\mathcal{O}$ with the latest object o, find $\mathcal{C}_o$—the target users of o.

In this problem setting, we assume a sizable preference relation is available for each user. In reality, we have insufficient information about the preferences of a less active user, i.e., the corresponding partial orders may contain very few preference tuples. In the extreme case, a new user, for whom we have no information regarding their preferences, admits all objects as Pareto-optimal. Such less active users and new users are the subject of the well-known *cold-start* problem in recommendation systems, which is outside of the scope of this work.

2.4 Sharing Computation across Users

Algorithm Baseline A simple method to our problem will check, for every user, whether a new object belongs to the corresponding Pareto frontier. The pseudo code of this approach, named Baseline, is shown in Algorithm 1. Upon the arrival of a new object o, for every user c, it sequentially compares o with the current Pareto-optimal objects in $\mathcal{P}_c$. 1) If o is dominated by any o' or o is identical to o', further comparison with the remaining objects in $\mathcal{P}_c$ is skipped. In the case of o being dominated by o', o is disqualified from being a Pareto-optimal object; if o is identical to o', then o is Pareto-optimal, i.e., it is inserted into $\mathcal{P}_c$. 2) If o dominates any o', o' is discarded from $\mathcal{P}_c$. It can be concluded already that o belongs to $\mathcal{P}_c$, but the comparisons should continue since o may dominate other existing objects in $\mathcal{P}_c$. 3) If o is not dominated by any object in $\mathcal{P}_c$, it becomes an element of $\mathcal{P}_c$. Readers familiar with the literature on skyline queries may have realized that the gist of the algorithm is essentially the basic skyline query algorithm [9]. The crux of its operation is based on an important property, that it suffices to compare new objects with only the Pareto-optimal objects, since any new object dominated by a non Pareto-optimal object must be dominated by some Pareto-optimal objects too.

With regard to a user c, the complexity of finding the Pareto frontier among n objects is $O(n^2)$. Algorithm 1 needs $O(n^2 \cdot |\mathcal{C}|)$ time to compute the Pareto frontiers for all users in $\mathcal{C}$. The drawback of Baseline is it repeatedly applies the same procedure for every user. In

```
Algorithm 1: Baseline
    Input: $\mathcal{C}$: all users; $\mathcal{O}$: existing objects; $o$: a new object
    Output: $\mathcal{C}_o$: target users of $o$
1   $\mathcal{C}_o \leftarrow \emptyset$;
2   foreach $c \in \mathcal{C}$ do
3       updateParetoFrontier$(c, o)$;
4   return $\mathcal{C}_o$;

    Procedure: updateParetoFrontier $(c, o)$
1   $isPareto \leftarrow$ true;
2   foreach $o' \in \mathcal{P}_c$ do
3       if $o \succ_c o'$ then
4           $\mathcal{P}_c \leftarrow \mathcal{P}_c - \{o'\}; \mathcal{C}_{o'} \leftarrow \mathcal{C}_{o'} - \{c\}$;
5       else if $o' \succ_c o$ then $isPareto \leftarrow$ false;break ;
6       else if $o'.\mathcal{D} = o.\mathcal{D}$ then $isPareto \leftarrow$ true;break ;
7   if $isPareto$ then
8       $\mathcal{P}_c \leftarrow \mathcal{P}_c \cup \{o\}; \mathcal{C}_o \leftarrow \mathcal{C}_o \cup \{c\}$;
```

terms of computation efficiency, the approach may become particularly unappealing when there are a large number of users and new objects constantly arrive. To counter this drawback, our idea is to share computations across the users that exhibit similar preferences. To this end, our method is simple and intuitive. If several users share a set of preference tuples, it is only necessary to compare two objects once, if they attain the attribute values in the preference tuples. If an object is dominated by another object according to these common preference tuples, it is dominated with respect to all users sharing the same preferences. This idea guarantees to filter out only "true negatives" for these users, and it only needs to further discern "false positives" for each individual user.

Definition 5 (Common Preference Tuple and Relation)**.** Given a set of users $U \subseteq \mathcal{C}$, an attribute $d \in \mathcal{D}$, and two values $x, y \in dom(d)$, if (x, y) belongs to preference relation $\succ_c^d$

for all $c \in U$, then it is called a *common preference tuple*. The set of common preference tuples of U on attribute d is denoted $\succ_U^d$, i.e., $\succ_U^d = \bigcap_{c \in U} \succ_c^d$. By definition, $\succ_U^d$ also represents a strict partial order (Theorem 1, proof omitted). We call it a *common preference relation*. It can be viewed as the preference of a virtual user that is denoted U. $\triangle$

Theorem 1. *$\succ_U^d$ is a strict partial order.* $\triangle$

Since, for each d, $\succ_U^d$ is a strict partial order, the set of users' preferences (i.e., the virtual user U's preferences) regarding all attributes in $\mathcal{D}$ induce another strict partial order $\succ_U$ on objects.

Definition 6 (Pareto Frontier for U)**.** An object o is *Pareto-optimal* with respect to U if no other object dominates it according to $\succ_U$. The Pareto frontier of $\mathcal{O}$ for U is denoted $\mathcal{P}_U$, i.e., $\mathcal{P}_U = \{o \in \mathcal{O} | \nexists o' \in \mathcal{O} \text{ s.t. } o' \succ_U o\}$. $\triangle$

Example 3. From Table 2.2, $\succ_{c_1}^{\text{CPU}} = \{$(*dual*, *single*), (*dual*, *quad*), (*dual*, *triple*), (*triple*, *single*), (*quad*, *single*)$\}$ and $\succ_{c_2}^{\text{CPU}} = \{$(*dual*, *single*), (*triple*, *single*), (*quad*, *single*), (*triple*, *dual*), (*quad*, *dual*), (*quad*, *triple*)$\}$. According to Definition 5, the common preference relation of c_1 and c_2 is $\succ_{\{c_1,c_2\}}^{\text{CPU}} = \{$(*dual*, *single*), (*triple*, *single*), (*quad*, *single*)$\}$. Similarly we can derive $\succ_{\{c_1,c_2\}}^{\text{display}}$ and $\succ_{\{c_1,c_2\}}^{\text{brand}}$. In Table 2.2, the three partial orders are depicted in a row labeled as a virtual user U. The Pareto frontier of U is $\mathcal{P}_U = \{o_2, o_3, o_{10}, o_{15}\}$. $\triangle$

Theorem 2. *Given any set of users U, for all $c \in U$, $\mathcal{P}_U \supseteq \mathcal{P}_c$ and $\overline{\mathcal{P}}_U \subseteq \bigcap_{c \in U} \overline{\mathcal{P}}_c$.* $\triangle$

Proof: We prove by contradiction. Suppose that there exists $c \in U$ such that $\mathcal{P}_U \not\supseteq \mathcal{P}_c$, which would mean there exists $o \in \mathcal{O}$ such that $o \in \mathcal{P}_c$ and $o \notin \mathcal{P}_U$. That implies the existence of an $o' \in \mathcal{O}$ such that $o' \succ_U o$ and $o' \not\succ_c o$. However, by Definition 5, $o' \succ_U o$ implies $o' \succ_c o$. Therefore, the existence of o' is impossible. This contradiction eventually leads to that $\mathcal{P}_U \supseteq \mathcal{P}_c$. Hence, $\mathcal{P}_U \supseteq \bigcup_{c \in U} \mathcal{P}_c$, which implies $\overline{\mathcal{P}}_U \subseteq \bigcap_{c \in U} \overline{\mathcal{P}}_c$ according to De Morgan's laws.

Lemma 1. *Given any set of users U, for all $c \in U$, $\mathcal{P}_c = \{o \in \mathcal{P}_U | \nexists o' \in \mathcal{P}_U$ s.t. $o' \succ_c o\}$.* $\triangle$

Example 4. In Table 2.2, $\mathcal{P}_U = \{o_2, o_3, o_{10}, o_{15}\}$ and $\mathcal{P}_{c_1} \cup \mathcal{P}_{c_2} = \{o_2, o_3, o_{15}\}$. $\mathcal{P}_U \supseteq \mathcal{P}_{c_1} \cup \mathcal{P}_{c_2}$. Moreover, $\overline{\mathcal{P}}_U$ = $\{o_1, o_4, o_5, o_6, o_7, o_8, o_9, o_{11}, o_{12}, o_{13}, o_{14}\}$ and $\overline{\mathcal{P}}_{c_1} \cap \overline{\mathcal{P}}_{c_2} = \{o_1, o_4, o_5, o_6, o_7, o_8, o_9, o_{10}, o_{11}, o_{12}, o_{13}, o_{14}, o_{15}\}$. $\overline{\mathcal{P}}_U \subseteq \overline{\mathcal{P}}_{c_1} \cap \overline{\mathcal{P}}_{c_2}$. $\triangle$

Theorem 2 suggests an appealing quality of the common preference relations of U. By $\mathcal{P}_U \supseteq \mathcal{P}_c$, the Pareto frontier of U subsumes the Pareto frontier of every user member in U. What it means is that, if we simply compute the Pareto frontier of U, we get to retain all the objects that we eventually look for. Consider $\mathcal{P}_c$ as the ground truth and $\mathcal{P}_U$ as the predictions. The objects that are filtered out ($\overline{\mathcal{P}}_U$) are all "true negatives" and there are no "false negatives". The set $\mathcal{P}_U$ may contain "false positives", which we just need to throw out after further verification, as Lemma 1 suggests.

This approach's merit is the potential saving on object comparisons. For a cluster of users, many non Pareto-optimal objects may be filtered out altogether for all the users, without incurring the same comparisons repeatedly for each user.

To capitalize on the above ideas, our method must answer three questions. (1) How to find users sharing similar preferences? (2) For a set of similar users U, how to maintain the corresponding Pareto frontier $\mathcal{P}_U$ based on their common preference relations $\succ_U^d$ for different attributes d? (3) For each user c in U, how to discern the "false positives" in $\mathcal{P}_U$ and thus find $\mathcal{P}_c$. Note that the second and the last challenges need to be addressed for constantly arriving new objects.

For (1), our method is to cluster users based on the similarity between their preference relations. While many clustering methods have been developed for various types of data, none is specialized in clustering partial orders. Our clustering method is discussed in Section 2.5. For (2) and (3), our algorithm takes a *filter-then-verify* approach and is thus named FilterThenVerify, of which the pseudo code is displayed in Algorithm 2.

Algorithm FilterThenVerify Upon the arrival of a new object o, for every cluster U, FilterThenVerify compares o with the current members of $\mathcal{P}_U$ based on the preference relations of the virtual user U. Various actions are taken, depending on the comparison outcomes, as follows:

I) If o dominates any o' in $\mathcal{P}_U$ according to $\succ_U^d$ of all relevant d, o' is removed from $\mathcal{P}_U$ (Line 7 of Procedure updateParetoFrontierU in Algorithm 2). For every $c \in \mathcal{C}$ such that $o' \in \mathcal{P}_c$, o' is also discarded from $\mathcal{P}_c$ (Line 6 of Procedure updateParetoFrontierU).

II) If o is dominated by any o' in $\mathcal{P}_U$, then o does not occupy the Pareto frontier of any user in U (Theorem 2). Further operations involving o are unnecessary (Line 8 of Procedure updateParetoFrontierU).

III) After comparing o with all current objects in $\mathcal{P}_U$, if it is realized that o is not dominated by any o', then o becomes a member of $\mathcal{P}_U$ (Line 9 of updateParetoFrontierU). Furthermore, for each $c \in U$, o is further compared with the members of $\mathcal{P}_c$ based on the preference relations of c, by using Procedure updateParetoFrontier of Algorithm1 (Line 6 of Algorithm2).

Example 5. In this example we explain the execution of FilterThenVerify on Table 2.1 and Table 2.2. Suppose users c_1 and c_2 form a cluster U, of which the preference relations are depicted in Table 2.2. The existing objects are o_1 to o_{14}, and $o_{15} = \langle 16.5''$, *Lenovo*, *quad*$\rangle$ is the object that just becomes available. Before o_{15} arrives, the Pareto frontier of U is $\mathcal{P}_U = \{o_2, o_3, o_7, o_{10}\}$. The algorithm starts by comparing o_{15} with each element in $\mathcal{P}_U$. As o_{15} dominates $o_7 = \langle 9.5''$, *Lenovo*, *quad*$\rangle$ according to U's preference relations, o_7 is discarded from $\mathcal{P}_U$. Before o_{15} arrives, o_7 belongs to $\mathcal{P}_{c_2}$ and $\mathcal{C}_{o_7}$=$\{c_2\}$. Hence, o_7 is removed from $\mathcal{P}_{c_2}$ and $\mathcal{C}_{o_7}$ becomes empty. o_{15} does not dominate any other object in $\mathcal{P}_U$. It is not dominated by any either. Therefore, it is inserted into $\mathcal{P}_U$.

o_{15} is further compared with the existing members of $\mathcal{P}_{c_1}$ and $\mathcal{P}_{c_2}$. It is dominated by o_2=$\langle 14''$, *Apple*, *dual*$\rangle$ according to c_1's preference relations. Thus it is not part of $\mathcal{P}_{c_1}$. According to c_2's preferences, o_{15} does not dominate any existing Pareto optional object

Algorithm 2: FilterThenVerify

```
Input: $U_1$, $U_2$,..., $U_n$: clusters of users; $\mathcal{O}$: existing objects; $o$: a new object
Output: $\mathcal{C}_o$: target users of $o$
1  $\mathcal{C}_o \leftarrow \emptyset$;
2  foreach $U \in \{U_1, U_2, ..., U_n\}$ do
3      $isPareto \leftarrow$ updateParetoFrontierU$(U, o)$;
4      if $isPareto$ then
5          foreach $c \in U$ do
6              updateParetoFrontier$(c, o)$; //Algorithm 1
7  return $\mathcal{C}_o$;

Procedure: updateParetoFrontierU $(U, o)$
1  $isPareto \leftarrow$ true;
2  foreach $o' \in \mathcal{P}_U$ do
3      if $o \succ_U o'$ then
4          foreach $c \in U$ do
5              if $o' \in \mathcal{P}_c$ then
6                  $\mathcal{P}_c \leftarrow \mathcal{P}_c - \{o'\}$; $\mathcal{C}_{o'} \leftarrow \mathcal{C}_{o'} - \{c\}$;
7          $\mathcal{P}_U \leftarrow \mathcal{P}_U - \{o'\}$;
8      else if $o' \succ_U o$ then $isPareto \leftarrow$ false; break ;
9  if $isPareto$ then $\mathcal{P}_U \leftarrow \mathcal{P}_U \cup \{o\}$ ;
10 return $isPareto$;
```

(except the aforementioned o_7 which by now is already discarded). Therefore $\mathcal{P}_{c_2}$ is not further changed and o_{15} becomes part of $\mathcal{P}_{c_2}$. Overall, $\mathcal{C}_{o_{15}}$=$\{c_2\}$.

Moreover, consider the arrival of $o_{16} = \langle 16''$, *Toshiba*, *single*$\rangle$ after o_{15}. In the process of comparing o_{16} with $\mathcal{P}_U = \{o_2, o_3, o_{10}, o_{15}\}$, it is realized that o_{16} is dominated by o_2 according to U's preference relations. Therefore, it does not belong to $\mathcal{P}_U$. It is thus unnecessary to further compare o_{16} with $\mathcal{P}_{c_1}$ or $\mathcal{P}_{c_2}$. $\mathcal{C}_{o_{16}}$=$\emptyset$. Thereby, updateParetoFrontierU acts as a sieve to filter out non Pareto-optimal objects such as o_{16}. In this way FilterThen-Verify reduces computation cost by avoiding repeated comparisons with such objects. $\triangle$

Complexity Analysis of Algorithm 2 As we discussed earlier, given a user c, the complexity of finding Pareto frontier among the n objects is $O(n^2)$. Assume k is the number of clusters. With regard to the virtual user for each cluster U, the complexity of finding Pareto frontier $\mathcal{P}_U$ among the n objects is $O(n^2 \cdot k)$ (calling Procedure updateParetoFrontierU in Line 3 of Algorithm 2). Assume each $\mathcal{P}_U$ on average has m objects. In Lines 4-6, Algorithm 2 finds $\mathcal{P}_c$ from $\mathcal{P}_U$ for each user c in U (recall that $\mathcal{P}_U \supseteq \mathcal{P}_c$). As Lines 4-6 iterate for each cluster (Line 2), the algorithm eventually computes $\mathcal{P}_c$ for each $c \in \mathcal{C}$. Therefore, the complexity of finding Pareto frontier $\mathcal{P}_c$ among the m objects is $O(m^2 \cdot |\mathcal{C}|)$. Overall, FilterThenVerify needs $O(n^2 \cdot k + m^2 \cdot |\mathcal{C}|)$ time to find the target users for all objects. We compare FilterThenVerify and Baseline in terms of time complexity. Apparently $k < |\mathcal{C}|$ and $m < n$. Thus, $n^2 \cdot k < n^2 \cdot |\mathcal{C}|$ and $m^2 \cdot |\mathcal{C}| < n^2 \cdot |\mathcal{C}|$.

2.5 Similarity Measures for Clustering User Preferences

This section discusses how to cluster users based on their preference relations. Our focus is on the similarity measures rather than the clustering method. The method we adopt is the conventional hierarchical agglomerative clustering algorithm [19]. The pseudo code is described in Algorithm 3. At every iteration, the method merges the two most similar clusters. The common preference relation of the merged cluster U on each attribute d, i.e., $\succ_U^d$, is computed. It then calculates the similarity between U and each remaining cluster. Given two clusters U_1 and U_2, their similarity $sim(U_1, U_2)$ is defined as the summation of the similarities between their preference relations on individual attributes, as follows. This resembles the high-level idea of using L_1 norm distance between centroids for measuring inter-cluster similarity in conventional hierarchial clustering.

Algorithm 3: Hierarchical Clustering of Users' Preference Relations

```
Input: $\mathcal{C}$: all users
1  $N \leftarrow |\mathcal{C}|$;
2  for $i = 1$ to $N$ do
3      for $j = 1$ to $N$ do
4          Compute $sim(c_i, c_j)$;
5  repeat
6      Merge two closest clusters of maximum similarity measures as $U$;
7      $N \leftarrow N - 1$;
8      foreach $d \in \mathcal{D}$ do
9          Compute $\succ_U^d$;
10     for $i = 1$ to $N$ do
11         Compute $sim(U, U_i)$;
12 until $N \neq 1$;
```

$$sim(U_1, U_2) = \sum_{d \in \mathcal{D}} sim^d(U_1, U_2) \tag{2.1}$$

Individual users' and clusters' preference relations on attributes are strict partial orders. No prior work studied clustering approaches or similarity measures for partial orders. Similarity measures commonly used in clustering algorithms assume numeric or categorical attributes. Kamishima et al. [24, 23] and Ukkonen et al. [37] cluster total orders but not partial orders. Given two totally ordered attributes, these works use the comparative ranks of the corresponding values to measure similarity. Clearly, such similarity measures are not applicable for partially ordered attributes.

In this section, we propose four different similarity functions for defining $sim^d(U_1, U_2)$.

1) Intersection size This is simply the size of the intersection of $\succ^d_{U_1}$ and $\succ^d_{U_2}$, i.e., the number of common preference tuples of all users in the two clusters U_1 and U_2. It is defined as

$$sim^d_i(U_1, U_2) = |\succ^d_{U_1} \cap \succ^d_{U_2}| \tag{2.2}$$

Example 6. Table 2.3 shows three clusters U_1 ($\{c_1,\ c_2\}$), U_2 ($\{c_3,\ c_4\}$), and U_3 ($\{c_5,\ c_6\}$) and the common preference relation associated with each cluster on attribute brand. U_1 and U_2 do not share any preference tuple and thus $sim^{\mathsf{brand}}_i(U_1, U_2) = 0$. U_1 and U_3 have (*Apple*, *Samsung*) and (*Lenovo*, *Samsung*) as common preference tuples, i.e., $sim^{\mathsf{brand}}_i(U_1, U_3) = 2$. Similarly, U_2 and U_3 share (*Lenovo*, *Apple*) and (*Lenovo*, *Toshiba*), i.e., $sim^{\mathsf{brand}}_i(U_2, U_3) = 2$. △

2) Jaccard similarity The measure sim_i captures the absolute size of the intersection of two preference relations. It does not take into account their differences. Consider three clusters U_1, U_2 and U_3 such that $sim^d_i(U_1, U_2) = sim^d_i(U_1, U_3)$ (i.e., $|\succ^d_{U_1} \cap \succ^d_{U_2}| = |\succ^d_{U_1} \cap \succ^d_{U_3}|$) and $|\succ^d_{U_1} \cup \succ^d_{U_2}| < |\succ^d_{U_1} \cup \succ^d_{U_3}|$. We can argue that the similarity between U_1 and U_2 should be higher than (instead of equal to) that between U_1 and U_3, because U_1 and U_2 have a larger percentage of common preference tuples than U_1 and U_3. To address this limitation of sim_i, we define the *Jaccard similarity* between two preference relations as their intersection size over their union size, i.e., the ratio of common preference tuples to all preference tuples in the two preference relations. Formally,

$$sim^d_j(U_1, U_2) = \frac{|\succ^d_{U_1} \cap \succ^d_{U_2}|}{|\succ^d_{U_1} \cup \succ^d_{U_2}|} = \frac{sim^d_i(U_1, U_2)}{|\succ^d_{U_1} \cup \succ^d_{U_2}|} \tag{2.3}$$

Example 7. Continue Example 6. $\succ^{\mathsf{brand}}_{U_1}$ and $\succ^{\mathsf{brand}}_{U_3}$ have 6 preference tuples in total while $\succ^{\mathsf{brand}}_{U_2}$ and $\succ^{\mathsf{brand}}_{U_3}$ have 7. Thus, $sim^{\mathsf{brand}}_j(U_1, U_3) = 2/6$ and $sim^{\mathsf{brand}}_j(U_2, U_3) = 2/7$. △

3) Weighted intersection size Intersection size and Jaccard similarity are based on the cardinalities of intersection and union sets of preference relations. In counting the cardinalities, they both treat all preference tuples equal. We argue that this is counter-intuitive.

Values at the top of a partial order matter more than those at the bottom, in terms of their impact on which objects belong to the Pareto frontier. Accordingly we introduce *weighted intersection size*, a modified version of intersection size sim_i. In counting the common preference tuples of two preference relations, it assigns a weight to each preference tuple. Formally,

$$sim^d_{wi}(U_1, U_2) = \sum_{(v,v') \in \succ^d_{U_1} \cap \succ^d_{U_2}} \frac{1}{2} \times \left(\frac{1}{\min\limits_{s \in S^d_{U_1}} D(s,v)+1} + \frac{1}{\min\limits_{s \in S^d_{U_2}} D(s,v)+1}\right) \tag{2.4}$$

In the above equation, with regard to an attribute d, the similarity between two clusters' preference relations is a summation over their common preference tuples. For each common preference tuple (v, v'), it computes the average weight of the better value v with respect to U_1 and U_2, respectively. Given a cluster U, S^d_U is the set of *maximal values* in the partial order $\succ^d_U$ and $D(s, v)$ for each $s \in S^d_U$ is the shortest distance from s to v in $\succ^d_U$. The weight of v in U is the inverse of the minimal distance from any maximal value to v (plus 1, to avoid division by zero). The concept of maximal value is defined as follows.

Definition 7 (Maximal Value)**.** With regard to $\succ^d_U$, value $x \in dom(d)$ is a *maximal value* if no other value in $dom(d)$ is preferred over x. The set of maximal values for $\succ^d_U$ is denoted S^d_U. Formally, $S^d_U = \{x \in dom(d) \mid \nexists y \in dom(d)\ s.t.\ (y, x) \in \succ^d_U\}$. △

Example 8. Continue Example 6. The maximal values in $\succ^{\text{brand}}_{U_1}$, $\succ^{\text{brand}}_{U_2}$ and $\succ^{\text{brand}}_{U_3}$ are $S^{\text{brand}}_{U_1}$=\{*Apple*, *Toshiba*\}, $S^{\text{brand}}_{U_2}$ = \{*Samsung*\} and $S^{\text{brand}}_{U_3}$=\{*Lenovo*\}, respectively. In the partial order corresponding to $\succ^{\text{brand}}_{U_1}$, the minimal shortest distances to *Apple*, *Lenovo*, *Samsung*, and *Toshiba* from the maximal values \{*Apple*, *Toshiba*\} are 0, 1, 1 and 0, respectively. The corresponding weights are 1, $1/2$, $1/2$ and 1. Similarly, in $\succ^{\text{brand}}_{U_2}$, the weights of *Apple*, *Lenovo*, *Samsung* and *Toshiba* are $1/3$, $1/2$, 1 and $1/3$, respectively. In $\succ^{\text{brand}}_{U_3}$, the corresponding weights are $1/2$, 1, $1/3$ and $1/2$, respectively.

U_1 and U_3 have (*Apple*, *Samsung*) and (*Lenovo*, *Samsung*) as common preference tuples. For the two better-values in these preference tuples—*Apple* and *Lenovo*, the average weights are both $3/4$. The similarity $sim_{wi}^{\mathsf{brand}}(U_1, U_3) = \frac{1+\frac{1}{2}}{2} + \frac{\frac{1}{2}+1}{2} = \frac{3}{2}$. Similarly, U_2 and U_3 have (*Lenovo*, *Apple*) and (*Lenovo*, *Toshiba*) as common preference tuples. In U_2 and U_3, the average weight of *Lenovo*—the better-value in both common preference tuples—is $3/4$. The similarity $sim_{wi}^{\mathsf{brand}}(U_2, U_3) = \frac{\frac{1}{2}+1}{2} + \frac{\frac{1}{2}+1}{2} = \frac{3}{2}$. $\triangle$

4) Weighted Jaccard similarity This measure is a combination of the last two ideas—Jaccard similarity and weighted intersection size. As in Jaccard similarity, *weighted Jaccard similarity* computes the ratio of intersection size to union size. Similar to weighted intersection size, the values in a preference relation are assigned weights corresponding to their minimal shortest distances to the preference relation's maximal values. The measure's definition is as follows.

$$\begin{aligned}
sim_{wj}^{d}(U_1,U_2) = & \sum_{(v,v')\in \succ_{U_1}^{d} \cap \succ_{U_2}^{d}} \frac{1}{2} \times \Big(\frac{1}{\min\limits_{s\in S_{U_1}^{d}} D(s,v)+1} + \frac{1}{\min\limits_{s\in S_{U_2}^{d}} D(s,v)+1}\Big) \\
& \Big/ \sum_{(v,v')\in \succ_{U_1}^{d} \cup \succ_{U_2}^{d}} \frac{1}{2} \times \Big(\frac{1}{\min\limits_{s\in S_{U_1}^{d}} D(s,v)+1} + \frac{1}{\min\limits_{s\in S_{U_2}^{d}} D(s,v)+1}\Big) \\
= & \; sim_{wi}^{d}(U_1,U_2) \Big/ \Big[sim_{wi}^{d}(U_1,U_2) + \sum_{(v,v')\in \succ_{U_1}^{d} - \succ_{U_2}^{d}} \frac{1}{\min\limits_{s\in S_{U_1}^{d}} D(s,v)+1} \\
& + \sum_{(v,v')\in \succ_{U_2}^{d} - \succ_{U_1}^{d}} \frac{1}{\min\limits_{s\in S_{U_2}^{d}} D(s,v)+1} \Big]
\end{aligned} \tag{2.5}$$

Example 9. Continue Example 8. Now $sim_{wj}^{\mathsf{brand}}(U_1,\, U_3) = \frac{\frac{3}{2}}{(1+1)+(1+1)+\frac{3}{2}} = \frac{3}{11}$, since $\succ_{U_1}^{d} - \succ_{U_3}^{d} = \{$(*Apple*, *Lenovo*), (*Toshiba*, *Samsung*)$\}$ and $\succ_{U_3}^{d} - \succ_{U_1}^{d} = \{$(*Lenovo*, *Apple*), (*Lenovo*, *Toshiba*)$\}$. Similarly, $sim_{wj}^{\mathsf{brand}}(U_2, U_3) = \frac{\frac{3}{2}}{(1+1+1)+(1+\frac{1}{2})+\frac{3}{2}} = \frac{3}{12}$, as $\succ_{U_2}^{d} - \succ_{U_3}^{d} =$ $\{$(*Samsung*, *Lenovo*), (*Samsung*, *Apple*), (*Samsung*, *Toshiba*)$\}$ and $\succ_{U_3}^{d} - \succ_{U_2}^{d} = \{$(*Lenovo*, *Samsung*), (*Apple*, *Samsung*)}. Note that $sim_{wj}^{\mathsf{brand}}(U_1, U_3) > sim_{wj}^{\mathsf{brand}}(U_2, U_3)$ although $sim_{wi}^{\mathsf{brand}}(U_1, U_3) = sim_{wi}^{\mathsf{brand}}(U_2, U_3)$. $\triangle$

2.6 Approximate User Preferences

Two conflicting factors have crucial impacts on the effectiveness of FilterThenVerify. One is the size of the common preference relations. The other is the size of the clusters. Specifically, the more preference tuples a cluster's users share, the more objects can be filtered out and thus the less verifications need to be done for individual users. On the contrary, the more users a cluster contains, the more repeated comparisons are avoided for these individual users. There is a clear tradeoff between these two factors, since larger clusters (i.e., more users in each cluster) naturally leads to smaller common preference relations.

Our approach to this challenge is *approximation*. As discussed in Section 2.1, it suffices for many applications to approximately identify target users. In this section, we show that we can find such approximation through a relaxed notion of common preference tuple, namely *approximate common preference tuple*. For a set of users, it allows a preference tuple to be absent from a tolerably small subset. If a sizable subset of the users agree with the preference tuple, it is considered an approximate common preference tuple. This relaxation addresses the aforementioned concern, since more approximate common preferences lead to larger clusters.

2.6.1 Approximate Common Preference Tuples and Relations

Based on the aforementioned objective, we procedurally construct approximate common preference relations. Before we provide its formal definition, we explain the intuition, as follows. Given a cluster of users, the resulting approximate common preference relation always includes the common preference tuples. The remaining possible preference tuples are considered in descending order of their frequencies, since preference tuples with higher frequencies are shared by more users. A preference tuple is included into the approximate common preference relation only if its reverse tuple is not included. This guarantees asymmetry. Furthermore, when a preference tuple is included, the transitive closure of the updated approximate common preference relation is also included. This guarantees transitivity. Irreflexivity is guaranteed too since this procedure never considers preference tuples in the form of (x, x). These altogether assure the constructed preference relation is a strict partial order. Given an append-only database of objects, a strict partial order ensures that preference query results are independent of the order by which objects are appended to the database. We denote the approximate common preference relation by $\widehat{\succ}_{U}^{d}$. It can be viewed as the preference of a virtual user (denoted $\widehat{U}$) on attribute d. Moreover, we denote the Pareto frontier of $\mathcal{O}$ for $\widehat{U}$ as U.

Definition 8 (Approximate Common Preference Tuple and Relation)**.** Given a set of users $U{\subseteq}\mathcal{C}$, an attribute $d{\in}\mathcal{D}$ of which $|dom(d)|{=}m$, consider $A_{1\ldots m2}$ which is an ordered permutation of all possible preference tuples $\{(x, y) \in dom(d) \times dom(d) \mid x \neq y\}$ such that $freq(A_i){\geq}freq(A_{i+1})$ for $i \in [1, m2 - 1]$, in which $freq(A_i)$ denotes the percentage of users in U whose preference relations contain preference tuple A_i. The *approximate common preference relation* $\widehat{\succ}_{U}^{d}$ is defined as R_j in which j is the largest index $i \in [1, m2]$ that satisfies the condition $(|R_i| < \theta_1 \wedge freq(A_i) > \theta_2) \vee freq(A_i) = 1$ where R_i is defined as

$$R_i = \begin{cases} \{A_1\} & \text{if } i = 1 \\ (R_{i-1} \cup \{A_i\})^{+} & \text{if } R_{i-1} \cup \{A_i\} \text{ is a strict partial order} \\ R_{i-1} & \text{otherwise} \end{cases}$$

and θ_1 and θ_2 are two given thresholds. θ_1 limits the size of the resulting $\widehat{\succ}_U^d$ while θ_2 excludes infrequent preference tuples from $\widehat{\succ}_U^d$. $\triangle$

θ_1 and θ_2 regulate the size of $\widehat{\succ}_U^d$. A pair of large θ_1 and small θ_2 allows $\widehat{\succ}_U^d$ to include infrequent preference tuples. In such a case the approximate common preference relation becomes ineffective, since Procedure updateParetoFrontierU in Algorithm2 may retain a large number of candidates that must be verified for each $c \in U$. On the other hand, a pair of small θ_1 and large θ_2 may limit $\widehat{\succ}_U^d$ to contain only $\succ_U^d$, in which case the concern regarding small common preference relation remains.

As Definition 8 itself is procedural, it naturally corresponds to a greedy algorithm for constructing approximate preference relation $\widehat{\succ}_U^d$. The pseudo code GetApproxPreference-Tuples is in Algorithm 4. First, all the common preference tuples are included (Lines 2-3). After that, preference tuples are considered in the order of frequency, as long as the two thresholds are satisfied (Line 4). For each preference tuple in consideration, if it together with all chosen tuples hitherto do not violate the properties of a strict partial order, their transitive closure is included into the approximate preference relation (Lines 6-7).

Example 10. We use Figure 2.1 to explain the execution of GetApproxPreferenceTuples. Figure 2.1a depicts three users' preference relations on brand. Suppose together these three users form a cluster. Assume $\theta_1 = 7$ and $\theta_2 = 60\%$.

Table 2.5 shows the frequencies of all possible preference tuples after sorting. For instance, since all users prefer *Apple* to *Toshiba*, the corresponding frequency is $3/3$; the frequency of (*Apple*, *Samsung*) is $2/3$ as two of these three users prefer *Apple* to *Samsung*. At first GetApproxPreferenceTuples includes the common preference tuple (*Apple*, *Toshiba*) into $\widehat{\succ}_U^d$. It then includes (*Apple*, *Samsung*), (*Lenovo*, *Toshiba*), and (*Toshiba*, *Samsung*) as approximate preference tuples too. Furthermore, upon the addition of (*Toshiba*, *Samsung*), GetApproxPreferenceTuples includes (*Lenovo*, *Samsung*) as well since (*Lenovo*, *Toshiba*)

Algorithm 4: GetApproxPreferenceTuples

```
Input: $A_i$: ordered permutation of all possible preference tuples, defined on $dom(d)$, in
       descending order of their frequencies among users $U$, $\theta_1$ and $\theta_2$: thresholds
Output: $\widehat{\succ}_U^d$: approximate common preference relation of $U$ on attribute $d$

1 for $i = 1$ to $|dom(d)|2$ do
2     if freq($A_i$) $= 1$ then
3         $\widehat{\succ}_U^d \leftarrow \widehat{\succ}_U^d \cup \{A_i\}$; continue;
4     if $|\widehat{\succ}_U^d| \geq \theta_1$ or freq($A_i$) $\leq \theta_2$ then
5         break;
6     if $(\widehat{\succ}_U^d \cup \{A_i\})$ is a strict partial order then
7         $\widehat{\succ}_U^d \leftarrow (\widehat{\succ}_U^d \cup \{A_i\})^+$;
8 return $\widehat{\succ}_U^d$;
```

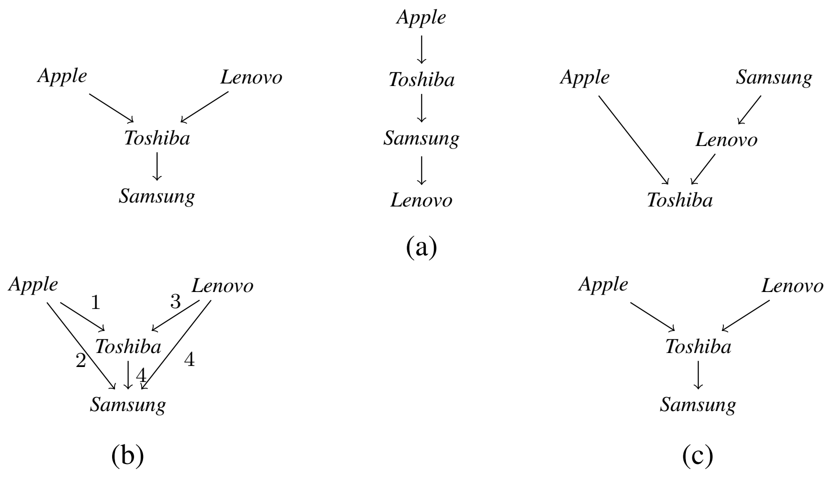


Figure 2.1: Execution of GetApproxPreferenceTuples. **a)** Input: the preferences of 3 users w.r.t. brand. **b)** The sequence of included approximate preference tuples. **c)** Output: the final Hasse diagram representation of the partial order.

and (*Toshiba*, *Samsung*) transitively induce it. The algorithm then considers (*Samsung*, *Lenovo*), which is disqualified since its reverse tuple (*Lenovo*, *Samsung*) is already included. Otherwise the tuples will not form a strict partial order. The algorithm stops at (*Apple*, *Lenovo*) because its frequency is below the threshold 60%. Figure2.1b illustrates

(A, T)	(A, S)	(L, T)	(T, S)	(S, L)	(A, L)	(L, S)	(T, L)	(S, T)	(L, A)	(T, A)	(S, A)
3/3	2/3	2/3	2/3	2/3	1/3	1/3	1/3	1/3	0/3	0/3	0/3

Table 2.5: All possible preference tuples in order of frequency. (*A*, *L*, *S* and *T* stand for *Apple*, *Lenovo*, *Samsung* and *Toshiba*.)

the sequence of the included tuples and Figure2.1c depicts the output approximate preference relation in the form of a Hasse diagram. △

2.6.2 False Positives and False Negatives due to Approximation

FilterThenVerify (Algorithm2) is extended to use approximate preference tuples and thus we rename it FilterThenVerifyApprox. The algorithm itself remains the same. Procedure updateParetoFrontierU maintains U as the candidate Pareto frontier. The algorithm eventually returns c for each user $c \in U$, in which $c = \{o \in U | \nexists o' \in U$ s.t. $o' \succ_c o\}$, i.e., $U \supseteq c$. Thus, $\widehat{\mathcal{C}}_o = \{c \in \mathcal{C} | o \in c\}$. We use the example below to explain its execution over approximate preference relations.

Example 11. Reconsider Example 5, but use the approximate preference relations associated with virtual user $\widehat{U}$ in Table 2.2. Upon the arrival of o_{15}, it is compared with the elements in $U = \{o_2, o_7\}$. U becomes $\{o_2, o_{15}\}$ since o_{15} dominates o_7. o_7 is then also removed from c_2. o_{15} is further compared with $c_1 = \{o_2\}$ and $c_2 = \{o_2\}$, which does not lead to any further change. Overall, $\widehat{\mathcal{C}}_{o_{15}} = \{c_2\}$. The target users using approximate preference relations remain identical to the exact ones, i.e., no loss of accuracy in this case. △

The rest of this section focuses on the accuracy of FilterThenVerifyApprox. It produces *false positives* if there exists such an o that $o \in c$ but $o \notin \mathcal{P}_c$. It produces *false negatives* if there exists such an o that $o \notin c$ but $o \in \mathcal{P}_c$. Below we present Theorems 3 and 4 to analyze how U and c relate to $\mathcal{P}_U$ and $\mathcal{P}_c$.

Lemma 2. *Given a set of users U and an attribute d, the common preference relation $\succ^d_U$ and an approximate common preference relation $\widehat{\succ}^d_U$ satisfy the following properties:*

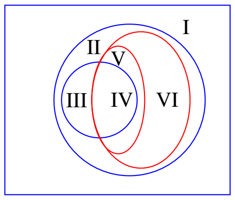


Figure 2.2: Venn diagram depicting $\mathcal{O}$, $\mathcal{P}_U$, U, $\mathcal{P}_c$ and c.

Set	Area Covered
$\mathcal{O}$	I,II,III,IV,V,VI
$\mathcal{P}_U$	II,III,IV,V,VI
U	IV,V,VI
$\mathcal{P}_c$	III,IV
c	IV,V

Table 2.6: Areas covered by $\mathcal{O}$, $\mathcal{P}_U$, U, $\mathcal{P}_c$ and c in Figure2.2.

	Exact		
Approx.		**Pareto frontier**	**Non Pareto frontier**
	Pareto frontier	IV	V
	Non Pareto frontier	III	I,II,VI

Table 2.7: Confusion matrix w.r.t. c.

1) *The approximate preference tuples are a superset of the common preference tuples, i.e.,* $\widehat{\succ}^d_U \supseteq \succ^d_U$.

2) *If any preference tuple along with its reverse tuple do not belong to the approximate common preference relation, neither of them belongs to the common preference relation either, i.e.,* $(x,y) \notin \widehat{\succ}^d_U \wedge (y,x) \notin \widehat{\succ}^d_U \Rightarrow (x,y) \notin \succ^d_U \wedge (y,x) \notin \succ^d_U$. △

Theorem 3. *Given objects $\mathcal{O}$ and users U, the Pareto frontier with regard to approximate common preference relations is a subset of the Pareto frontier with regard to common preference relations, i.e.,* $U \subseteq \mathcal{P}_U$. △

Proof: We prove by contradiction. Suppose $U \nsubseteq \mathcal{P}_U$, which would mean there exists $o \in \mathcal{O}$ such that $o \in U$ and $o \notin \mathcal{P}_U$. That leads to the existence of an o' such that $o' \succ_U o$ and $o' \not\succ_{\widehat{U}} o$. However, $o' \succ_U o$ implies $o' \succ_{\widehat{U}} o$ because $\widehat{\succ}^d_U \supseteq \succ^d_U$ for every d (Lemma 2). Therefore, the existence of o' is impossible. This contradiction proves that $U \subseteq \mathcal{P}_U$. ■

Lemma 3. *Given any set of users U, for all user $c \in U$, $U \supseteq c$.* △

Theorem 4. *Given any set of users U, for all user $c \in U$, $U \cap \mathcal{P}_c \subseteq c$.* △

Proof: We prove by contradiction. Suppose $U \cap \mathcal{P}_c \nsubseteq c$, which would mean there exists $o \in \mathcal{O}$ such that $o \in U \cap \mathcal{P}_c$ and $o \notin c$. $o \notin c$ implies the existence of an $o' \in \mathcal{O}$ such that $o' \in c$ and $o' \succ_c o$ (since $o \in U \cap \mathcal{P}_c$ and thus $o \in U$ which means $o' \not\succ_{\widehat{U}} o$). Since $o' \succ_c o$, $o \notin \mathcal{P}_c$ (Definition 3) and thus $o \notin U \cap \mathcal{P}_c$. In other words, the existence of o' is impossible. This contradiction proves that $U \cap \mathcal{P}_c \subseteq c$. ■

Consider a cluster U and a user $c \in U$. The Venn diagram in Figure 2.2 shows the effect of approximation through depicting $\mathcal{O}$ (rectangle), $\mathcal{P}_U$ (outer blue circle), U (outer red ellipse), $\mathcal{P}_c$ (inner blue circle), and c (inner red ellipse). Besides, Table 2.6 elaborates the area covered by these sets while Table 2.7 shows the confusion matrix for c. Note that using approximate common preference relations results in false negatives (III). Mistakenly declaring III as not Pareto-optimal further allows false positives (V) to sneak in.

With these notations in place, we are ready to quantify the accuracy of FilterThenVerifyApprox using standard evaluation measures in information retrieval. Specifically, *precision* is the fraction of objects found by FilterThenVerifyApprox that are truly Pareto-optimal, i.e., $\frac{\sum_{c \in \mathcal{C}} c \cap \mathcal{P}_c}{\sum_{c \in \mathcal{C}} c}$. *Recall* is the fraction of Pareto-optimal objects that are correctly found by FilterThenVerifyApprox, i.e., $\frac{\sum_{c \in \mathcal{C}} c \cap \mathcal{P}_c}{\sum_{c \in \mathcal{C}} \mathcal{P}_c}$. With regard to a specific user c, the algorithm's precision, recall and accuracy can be represented using the areas in Figure 2.2, as follows.

$$precision = \frac{|\,\text{IV}\,|}{|\,\text{IV} \cup \text{V}\,|} \tag{2.6}$$

$$recall = \frac{|\,\text{IV}|}{|\,\text{III} \cup \text{IV}\,|} \tag{2.7}$$

$$accuracy = \frac{|\,\text{I} \cup \text{II} \cup \text{IV} \cup \text{VI}\,|}{|\,\text{I} \cup \text{II} \cup \text{III} \cup \text{IV} \cup \text{V} \cup \text{VI}\,|} \tag{2.8}$$

2.6.3 Similarity Functions

To make the clustering solution in Section 2.5 compatible with approximate preference relations, we extend the similarity measures, using ideas inspired by the Jaccard similarity for non-negative multidimensional real vectors [11].

1) Jaccard Similarity Consider an attribute d with $|dom(d)| = m$. For each cluster U, construct a vector $\mathbf{U} = (\mathbf{U}(1), \mathbf{U}(2), \ldots, \mathbf{U}(m2))$. For $i \in [1, m2]$, $\mathbf{U}(i)$ represents the frequency of A_i (Definition 8) in U. Given two clusters U and V, their *Jaccard similarity* on attribute d is

$$sim_j^d(U, V) = \frac{\sum_i \min(\mathbf{U}(i), \mathbf{V}(i))}{\sum_i \max(\mathbf{U}(i), \mathbf{V}(i))} \tag{2.9}$$

Example 12. Consider U_1 and U_3 in Table 2.3. Suppose $A(i)$ for $i \in [1, m2]$ are ((*Apple*, *Lenovo*), (*Apple*, *Samsumg*), (*Apple*, *Toshiba*), (*Lenovo*, *Apple*), (*Lenovo*, *Samsung*), (*Lenovo*, *Toshiba*), (*Toshiba*, *Apple*), (*Toshiba*, *Lenovo*), (*Toshiba*, *Samsung*), (*Samsung*, *Apple*), (*Samsung*, *Lenovo*), (*Samsung*, *Toshiba*)). The two vectors are $\mathbf{U_1} = (2/2, 2/2, 0/2, 0/2, 2/2, 0/2, 0/2, 1/2, 2/2, 0/2, 0/2, 0/2)$ and $\mathbf{U_3} = (0/2, 2/2, 1/2, 2/2, 2/2, 2/2, 0/2, 0/2, 1/2, 0/2, 0/2, 0/2)$. For instance, $\mathbf{U_1}$ has $1/2$ on the 8^{th}-dimension since only one of the two users' preference relations contains (*Toshiba*, *Lenovo*). Hence, $sim_j^{\text{brand}}(U_1,U_3) = 0.36$. △

2) Weighted Jaccard Similarity This measure, denoted as sim_{wj}^d, extends the namesake measure in Section 2.5 with the idea above. Its definition is the same as Eq. 2.9 except that a value $\mathbf{U}(i)$ in a vector represents the frequency of A_i in U that takes into consideration the weights explained in Section 2.5. Consider A_i as the preference tuple $(A_i(x), A_i(y))$. This similarity measure is defined as follows.

$$\begin{aligned} sim^{d}_{wj}(U,V) = \sum_{i}(\min(\frac{1}{|U|} \times \sum_{c \in U} \frac{1}{\min\limits_{s \in S^{d}_{c}} D(s, A_i(x))+1}, \\ \frac{1}{|V|} \times \sum_{c \in V} \frac{1}{\min\limits_{s \in S^{d}_{c}} D(s, A_i(x))+1})) \\ \Big/ \sum_{i}(\max(\frac{1}{|U|} \times \sum_{c \in U} \frac{1}{\min\limits_{s \in S^{d}_{c}} D(s, A_i(x))+1}, \\ \frac{1}{|V|} \times \sum_{c \in V} \frac{1}{\min\limits_{s \in S^{d}_{c}} D(s, A_i(x))+1})) \end{aligned} \tag{2.10}$$

Example 13. In Table 2.3, in the partial order depicting $\succ^{\mathsf{brand}}_{c_6}$, the distance to *Apple* from the maximal value *Lenovo* is 1, i.e., the weight of *Apple* is $1/2$. Since only one of the two users in U_3 has (*Apple*, *Toshiba*) in their preference relation, $\mathbf{U_3}$ has $\frac{\frac{1}{2}+0}{2} = \frac{1}{4}$ on the 3^{rd}-dimension. In this way, we get $\mathbf{U_1} = (2/2,\ 2/2,\ 0/2,\ 0/2,\ 1/2,\ 0/2,\ 0/2,\ 1/2,\ 2/2,\ 0/2,\ 0/2,\ 0/2)$ and $\mathbf{U_3} = (0/2,\ 1/2,\ 1/4,\ 2/2,\ 2/2,\ 2/2,\ 0/2,\ 0/2,\ 1/4,\ 0/2,\ 0/2,\ 0/2)$. Therefore, $sim^{\mathsf{brand}}_{wj}(U_1,U_3) = 0.19$. △

2.7 Alive Object Dissemination

In Section 2.1, we discussed motivating applications such as social network content dissemination, news delivery and product recommendation. The significance of a particular social network content (e.g. a post in Facebook) or a piece of news diminishes eventually. Similarly, in any inventory, products are consumed and perishable products expire over time. In other words, objects can have limited lifetime. Thus, upon the arrival of a new object, it needs to compete only with the alive objects. To meet this requirement, we extend our problem as continuous monitoring of Pareto frontiers over *alive objects* for many users and formalize it as finding Pareto frontiers over *sliding window*.

Suppose $\mathcal{O} = \{o_1,\ o_2,\ \ldots,\ o_N\}$ is a stream of objects, in which the subscript of each object is its timestamp. We consider a sliding window as a sequence of W recent objects. Upon the arrival of an incoming object o_{in}, an object o_{out} expires if $in - out = W$. Specifically, the sliding window contains objects whose timestamps are in $(out, in]$, i.e., an object $o_i \in \mathcal{O}$ is alive during $(out, in]$ if $i \in (out, in]$. Given the concept of sliding window, we extend the definition of Pareto frontier in Definition 3 and the problem statement in Section 2.3.

Definition 9 (Pareto Frontier)**.** An alive object o is Pareto-optimal with respect to c, if no other alive object dominates it. $\mathcal{P}_c = \{o_i \in \mathcal{O} | \nexists o_j \in \mathcal{O} \text{ s.t. } o_j \succ_c o_i \wedge i, j \in (out, in]\}$. The target users of o_{in} is $\mathcal{C}_{o_{in}} = \{c \in \mathcal{C} | o_{in} \in \mathcal{P}_c\}$ (Definition 4). △

Problem Statement The problem of continuous monitoring of Pareto frontiers over sliding window is, given a set of users $\mathcal{C}$, their preference relations on attributes $\mathcal{D}$, and a stream of objects $\mathcal{O}$ with the incoming object o_{in} as well as the outgoing object o_{out}, find $\mathcal{C}_{o_{in}}$—the target users of o_{in}.

Algorithms BaselineSW and FilterThenVerifySW We extend Baseline and FilterThenVerify to BaselineSW and FilterThenVerifySW, respectively, to accommodate sliding window. The pseudo codes are described in Algorithm 5 and Algorithm 6, respectively. We note that no prior work studied Pareto frontier maintenance with regard to strict partial orders over sliding window. [28, 36, 29] studied skyline maintenance over sliding window, assuming numeric attributes. [32] considered total orders (with ties) on categorical attributes instead of general partial orders. There is no clear way to extend these works for partially ordered attributes.

Under the constraint of having a sliding window, an object can be excluded from Pareto frontier forever if it is dominated by any succeeding object. This observation is formalized as Theorem 5.

Theorem 5. *Consider a user $c \in \mathcal{C}$ and two objects $o_i, o_j \in \mathcal{O}$ such that $o_i \prec_c o_j$ and $i < j$. After the arrival of o_j, o_i can never be part of $\mathcal{P}_c$ in its remaining lifetime.* △

Algorithm 5: BaselineSW

```
Input: $\mathcal{C}$: all users; $\mathcal{P}$: Pareto frontier; $\mathcal{PB}$: Pareto frontier buffer; $o_{in}$: incoming object; $o_{out}$:
        outgoing object
Output: $\mathcal{C}_{o_{in}}$: target users of $o_{in}$
1  foreach $c \in \mathcal{C}$ do
2      if $o_{out} \in \mathcal{P}_c$ then
3          foreach $o \in \mathcal{PB}_c$ do
4              if $o_{out} \succ^c o$ then
5                  mendParetoFrontierSW$(c, o)$;
6      $\mathcal{PB}_c \leftarrow \mathcal{PB}_c - \{o_{out}\}$;
7      if $o_{in}$ not dominated by $\mathcal{P}_c$ then
8          updateParetoFrontierSW$(c, o)$;
9      refreshParetoBufferSW$(c, o_{in})$;
10 return $\mathcal{C}_{o_{in}}$;

Procedure: mendParetoFrontierSW $(c, o)$
1  $isPareto \leftarrow$ true;
2  foreach $o' \in \mathcal{P}_c$ do
3      if $o' \succ^c o$ then $isPareto \leftarrow$ false; break ;
4  if $isPareto$ then $\mathcal{P}_c \leftarrow \mathcal{P}_c \cup \{o\}; \mathcal{C}_o \leftarrow \mathcal{C}_o \cup \{c\}$ ;

Procedure: updateParetoFrontierSW $(c, o_{in})$
1  $\mathcal{P}_c \leftarrow \mathcal{P}_c \cup \{o_{in}\}; \mathcal{C}_{o_{in}} \leftarrow \mathcal{C}_{o_{in}} \cup \{c\}$;
2  foreach $o \in \mathcal{P}_c$ do
3      if $o_{in} \succ^c o$ then
4          $\mathcal{P}_c \leftarrow \mathcal{P}_c - \{o\}; \mathcal{C}_o \leftarrow \mathcal{C}_o - \{c\}$

Procedure: refreshParetoBufferSW $(c, o_{in})$
1  $\mathcal{PB}_c \leftarrow \mathcal{PB}_c \cup \{o_{in}\}$;
2  foreach $o \in \mathcal{PB}_c$ do
3      if $o_{in} \succ^c o$ then $\mathcal{PB}_c \leftarrow \mathcal{PB}_c - \{o\}$ ;
```

Proof: Since $i < j$, o_i expires before o_j and the sliding window always includes o_j if it includes o. Since o_j dominates o_i, o_i will never get into $\mathcal{P}_c$ after the arrival of o_j. ■

Example 14. Consider Table 2.1 and Table 2.2. Consider W, in and out as 5, 10 and 5, respectively. Upon the arrival of $o_{10} = \langle 9.5$, *Lenovo*, *triple*$\rangle$ and the expiration of $o_5 = \langle 9$, *Samsung*, *quad*$\rangle$, we get $\mathcal{P}_{c_1} = \{o_8\}$ and $\mathcal{P}_{c_2} = \{o_7, o_8\}$. △

Algorithm 6: FilterThenVerifySW

```
Input: $\{U_1, U_2, ..., U_n\}$: all clusters; $\mathcal{P}$: Pareto frontier; $\mathcal{PB}$: Pareto frontier buffer; $o_{in}$:
        incoming object; $o_{out}$: outgoing object
Output: $\mathcal{C}_{o_{in}}$: target users of $o_{in}$
1  foreach $i = 1$ to $n$ do
2      if $o_{out} \in \mathcal{P}_{U_i}$ then
3          foreach $o \in \mathcal{PB}_{U_i}$ do
4              if $o_{out} \succ o$ then
5                  $isPareto \leftarrow$ mendParetoFrontierUSW$(U_i, o)$;
6                  if $isPareto$ then
7                      foreach $c \in U_i$ do
8                          mendParetoFrontierSW$(c, o)$
9      $\mathcal{PB}_{U_i} \leftarrow \mathcal{PB}_{U_i} - \{o_{out}\}$;
10     if $o_{in}$ not dominated by $\mathcal{P}_{U_i}$ then
11         updateParetoFrontierUSW$(U_i, o)$;
12         foreach $c \in U_i$ do
13             if $o_{in}$ not dominated by $\mathcal{P}_c$ then
14                 updateParetoFrontierSW$(c, o)$;//Algorithm 5
15     refreshParetoBufferSW$(U_i, o_{in})$; //Algorithm 5
16 return $\mathcal{C}_{o_{in}}$;
```

```
Procedure: mendParetoFrontierUSW $(U, o)$
1  $isPareto \leftarrow$ true;
2  foreach $o' \in \mathcal{P}_U$ do
3      if $o' \succ^U o$ then return false ;
4  if $isPareto$ then $\mathcal{P}_U \leftarrow \mathcal{P}_U \cup \{o\}$ ;
5  return $isPareto$;
```

```
Procedure: updateParetoFrontierUSW
        $(U, o_{in})$
1  $\mathcal{P}_U \leftarrow \mathcal{P}_U \cup \{o_{in}\}$;
2  foreach $o \in \mathcal{P}_U$ do
3      if $o_{in} \succ^U o$ then $\mathcal{P}_c \leftarrow \mathcal{P}_c - \{o\}$ ;
```

By Theorem 5, we extend our algorithms to maintain a *Pareto frontier buffer* which stores at most W recent objects that are not dominated by any succeeding object. Clearly, o_{in} is part of the Pareto frontier buffer.

Definition 10 (Pareto Frontier Buffer)**.** With regard to user c and the sliding window $(out, in]$, an alive object o belongs to the Pareto frontier buffer if it is not dominated by any succeeding object. The Pareto frontier buffer is $\mathcal{PB}_c = \{o_i \in \mathcal{O} | \nexists o_j \in \mathcal{O}$ s.t. $o_j \succ_c o_i \wedge i, j \in (out, in] \wedge i < j\}$. By definition, $\mathcal{PB}_c \supseteq \mathcal{P}_c$ (Definition 9). △

Theorem 6. *Given a set of users U, for all $c \in U$, i) $\mathcal{PB}_U \supseteq \mathcal{P}_U$ and ii) $\mathcal{PB}_U \supseteq \mathcal{PB}_c$.* △

Proof: i) Together Definition 9 and 10 imply that $\mathcal{PB}_U \supseteq \mathcal{P}_U$.

ii) We prove by contradiction. Suppose that there exists $c \in U$ such that $\mathcal{PB}_U \not\supseteq \mathcal{PB}_c$, which would mean there exists $o \in \mathcal{O}$ such that $o \in \mathcal{PB}_c$ and $o \notin \mathcal{PB}_U$. That implies the existence of an $o' \in \mathcal{O}$ such that $o' \succ_U o$ and $o' \not\succ_c o$. However, by Definition 5, $o' \succ_U o$ implies $o' \succ_c o$. Therefore, the existence of o' is impossible. In conclusion, $\mathcal{PB}_U \supseteq \mathcal{PB}_c$. ■

Note that, BaselineSW needs to maintain an exclusive Pareto frontier buffer for each user ($\mathcal{PB}_c$) while a Pareto frontier buffer per cluster ($\mathcal{PB}_U$) is sufficient for FilterThenVerifySW.

Example 15. Continue Example 14. We get $\mathcal{PB}_{c_1}$=$\{o_8, o_9, o_{10}\}$. In this case, o_8 is the only element of $\mathcal{P}_{c_1}$. Since o_6 or o_7 could never been qualified in Pareto-optimality as they arrive before o_8, we do not need to store them. Nevertheless, upon the expiration of o_8, either o_9 or o_{10} could attain Pareto-optimality during their lifetime if they are not dominated by any following object. Therefore, o_9 and o_{10} are stored in $\mathcal{PB}_{c_1}$. For instance, o_{10} acquires Pareto-optimality during $(8, 13]$ as it does not dominated by any following object. △

In our sliding window framework, upon the expiration of an outgoing object o_{out}, for all $c \in \mathcal{C}$, at first BaselineSW calls Procedure mendParetoFrontierSW to mend $\mathcal{P}_c$. Because at this point, the alive objects those are exclusively dominated by o_{out}, acquire Pareto-

	display	brand	CPU
o_1	17	*Lenovo*	*dual*
o_2	9.5	*Sony*	*single*
o_3	12	*Apple*	*dual*
o_4	16	*Lenovo*	*quad*
o_5	19	*Toshiba*	*single*
o_6	12.5	*Samsung*	*quad*
o_7	14	*Apple*	*dual*

Table 2.8: Product Table

W	$\mathcal{P}_{c_1}$	$\mathcal{P}_{c_2}$	$\mathcal{PB}_{c_1}$	$\mathcal{PB}_{c_2}$
$[1, 6]$	$\{o_1, o_3\}$	$\{o_3, o_4\}$	$\{o_1, o_3, o_4, o_6\}$	$\{o_3, o_4, o_5, o_6\}$
$(1, 6]$	$\{o_3\}$	$\{o_3, o_4\}$	$\{o_3, o_4, o_6\}$	$\{o_3, o_4, o_5, o_6\}$
$(1, 7]$	$\{o_7\}$	$\{o_4, o_7\}$	$\{o_4, o_7\}$	$\{o_4, o_7\}$

Table 2.9: Content of Pareto Frontiers and Pareto Buffers During 3 Different Phases of Window of BaselineSW

W	$\mathcal{P}_U$	$\mathcal{P}_{c_1}$	$\mathcal{P}_{c_2}$	$\mathcal{PB}_U$
$[1, 6]$	$\{o_1, o_3, o_4\}$	$\{o_1, o_3\}$	$\{o_3, o_4\}$	$\{o_1, o_3, o_4, o_5, o_6\}$
$(1, 6]$	$\{o_3, o_4\}$	$\{o_3\}$	$\{o_3, o_4\}$	$\{o_3, o_4, o_5, o_6\}$
$(1, 7]$	$\{o_4, o_7\}$	$\{o_7\}$	$\{o_4, o_7\}$	$\{o_4, o_7\}$

Table 2.10: Content of Pareto Frontiers and Pareto Buffers During 3 Different Phases of Window of FilterThenVerifySW

optimality. While o_{in} arrives, if o_{in} belongs to $\mathcal{P}_c$, then Procedure updateParetoFrontierSW in BaselineSW discards objects that are dominated by o_{in}, thereby updates $\mathcal{P}_c$ (Line 8). After that, Procedure refreshParetoBufferSW in BaselineSW repairs $\mathcal{PB}_c$. Specifically, o_{in} replaces the alive objects from $\mathcal{PB}_c$ that it dominates (Line 9). Thus $\mathcal{PB}_c$ remains concurrent with Definition 10.

On the contrary, in case of FilterThenVerifySW, upon the expiration of an outgoing object o_{out}, Procedures mendParetoFrontierUSW and mendParetoFrontierSW together mend $\mathcal{P}_U$ and $\mathcal{P}_c$ for all $U \subseteq \mathcal{C}$, for all $c \in U$. While o_{in} arrives, if o_{in} belongs to $\mathcal{P}_U$, then Procedure updateParetoFrontierUSW discards objects from $\mathcal{P}_U$ that are dominated by

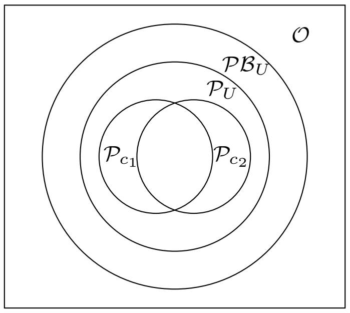


Figure 2.3: Venn diagram depicting $\mathcal{P}_{c_1}$, $\mathcal{P}_{c_2}$, $\mathcal{P}_U$, $\mathcal{PB}_U$ and $\mathcal{O}$

o_{in} (Line 11). Now for all $c \in U$, if c approves o_{in} as a Pareto-optimal object, then Procedure updateParetoFrontierSW finds out the objects in $\mathcal{P}_c$ dominated by o_{in} and thereby removes them (Line 14). Lastly, Procedure refreshParetoBufferUSW repairs $\mathcal{PB}_U$ so that it includes only the objects which have the potentiality to acquire Pareto-optimality over time with regard to U (Definition 10 and Theorem 6) (Line 15).

Example 16. The executions of BaselineSW and FilterThenVerifySW on Table 2.8 and Table 2.2 are briefly explained here. Consider W, in and out as 6, 7 and 1, respectively.

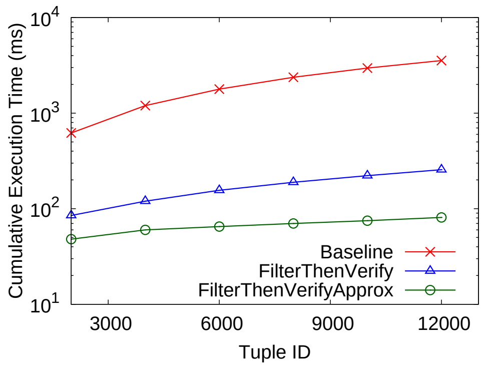
While the sliding window is at [1, 6], $\mathcal{P}_{c_1}$ = $\{o_1, o_3\}$, $\mathcal{P}_{c_2} = \{o_3, o_4\}$, $\mathcal{PB}_{c_1}$ = $\{o_1, o_3, o_4, o_6\}$ and $\mathcal{PB}_{c_2} = \{o_3, o_4, o_5, o_6\}$. Upon the expiration of $o_1 = \langle 17, \textit{Lenovo}, \textit{dual}\rangle$, the window is at $(1, 6]$. Now BaselineSW checks whether o_1 belongs to $\mathcal{P}_{c_1}$ and $\mathcal{P}_{c_2}$. Since o_1 belongs to $\mathcal{P}_{c_1}$, $\mathcal{P}_{c_1}$ is mended to $\{o_3\}$. Upon the arrival of $o_7 = \langle 14, \textit{Apple}, \textit{dual}\rangle$, the window includes objects correspond to $(1, 7]$. At this point BaselineSW starts checking whether o_7 is qualified as an element of $\mathcal{P}_{c_1}$ and $\mathcal{P}_{c_2}$, sequentially. In both $\mathcal{P}_{c_1}$ and $\mathcal{P}_{c_2}$, o_7 takes the place of o_3 (Line 8). After that, o_7 is stored to $\mathcal{PB}_{c_1}$ and $\mathcal{PB}_{c_2}$. Furthermore, for both $\mathcal{PB}_{c_1}$ and $\mathcal{PB}_{c_2}$, BaselineSW finds out the objects dominated by o_7 and discards them, i.e., $\{o_3, o_6\}$ and $\{o_3, o_5, o_6\}$, respectively. As these dominated objects arrives before o_7, they could never acquire Pareto-optimality. Now $\mathcal{PB}_{c_1} = \{o_4, o_7\}$ and $\mathcal{PB}_{c_2} = \{o_4, o_7\}$ (Line 9) (Definition 10). We get that $\mathcal{C}_{o_7} = \{c_1, c_2\}$. The content of Pareto frontiers and Pareto buffers at 3 phases of window of BaselineSW is shown in Table 2.9.

In case of FilterThenVerifySW, while the sliding window is at $[1, 6]$, $\mathcal{P}_{c_1} = \{o_1, o_3\}$, $\mathcal{P}_{c_2} = \{o_3, o_4\}$, $\mathcal{P}_U = \{o_1, o_3, o_4\}$ and $\mathcal{PB}_U = \{o_1, o_3, o_4, o_5, o_6\}$. Upon the expiration of o_1, the algorithm checks whether o_1 belongs to $\mathcal{P}_U$. Therefore, $\mathcal{P}_U$ becomes $\{o_3, o_4\}$ while the window is at $(1, 6]$. Upon the arrival of o_7, the window includes objects correspond to $(1, 7]$. Now FilterThenVerifySW starts checking whether o_7 can occupy $\mathcal{P}_U$. With respect to U, o_7 dominates o_3, i.e., o_7 replaces of o_3 in $\mathcal{P}_U$ (Line 11) as well as in both $\mathcal{P}_{c_1}$ and $\mathcal{P}_{c_2}$ (Line 14). After that, o_7 is stored in $\mathcal{PB}_U$. Moreover, o_7 dominates o_3, o_5 and o_6 in $\mathcal{PB}_U$. Since each of these dominated objects in $\mathcal{PB}_U$ arrives before o_7, they could never been qualified in Pareto-optimality. Therefore, FilterThenVerifySW discards them from $\mathcal{PB}_U$, i.e., $\mathcal{PB}_U = \{o_4, o_7\}$ (Line 15) (Definition 10). Finally we get $\mathcal{C}_{o_7} = \{c_1, c_2\}$. The content of Pareto frontiers and Pareto buffers at 3 phases of window of FilterThenVerifySW are shown in Table 2.10. The Venn diagram in Figure2.3 depicts $\mathcal{P}_{c_1}$, $\mathcal{P}_{c_2}$, $\mathcal{P}_U$ and $\mathcal{PB}_U$. Note that, while BaselineSW needs to maintain individual Pareto frontier buffer per user ($\mathcal{PB}_{c_1}$ and $\mathcal{PB}_{c_2}$), a shared Pareto frontier buffer per cluster ($\mathcal{PB}_U$) suffices for FilterThenVerifySW. In conclusion, along with Pareto frontier maintenance, FilterThenVerifySW prunes comparisons in terms of Pareto buffer maintenance. $\triangle$

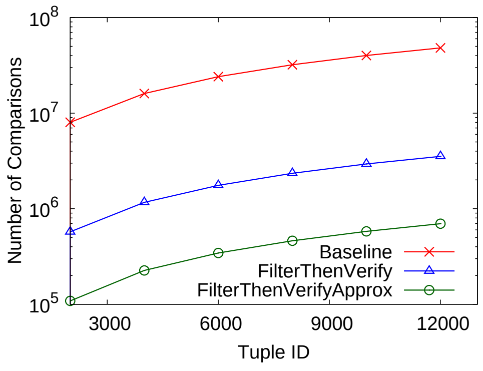
2.8 Experiments

Dataset	$\lvert\mathcal{O}\rvert$	$h = 0.70$			$h = 0.65$			$h = 0.60$			$h = 0.55$		
		P	R	F	P	R	F	P	R	F	P	R	F
Movie	$12,749$	100	95.43	97.67	100	93.93	96.87	99.99	93.28	96.52	99.99	90.46	94.99
Publication	$17,598$	100	96.59	98.27	100	95.85	97.88	100	95.54	97.72	100	95.13	97.51

Table 2.11: The precision, recall and F-measure (in percentage) of FilterThenVerifyApprox. Varying h, d=4. (P, R, and F stand for Precision, Recall, and F-measure.)

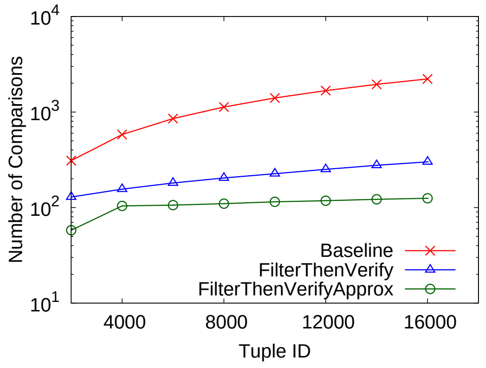


(a) Execution time

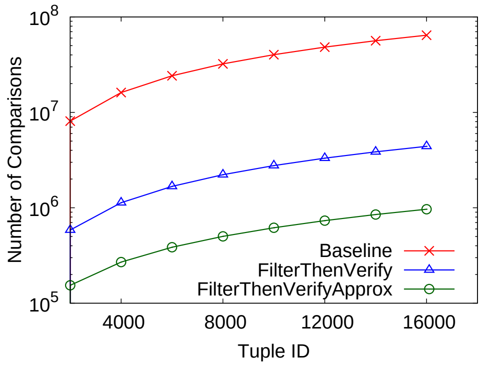


(b) Object comparisons

Figure 2.4: Comparison of Baseline, FilterThenVerify and FilterThenVerifyApprox on the movie dataset. Varying $|\mathcal{O}|$, $h = 0.55$, $d = 4$.



(a) Execution time



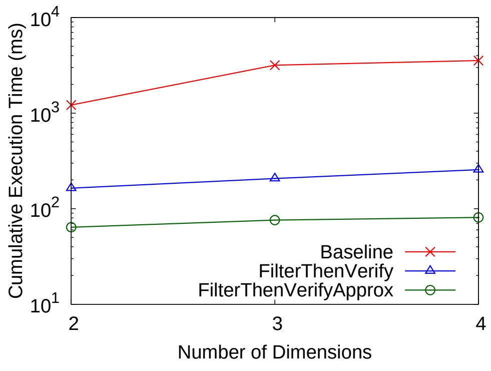
(b) Object comparisons

Figure 2.5: Comparison of Baseline, FilterThenVerify and FilterThenVerifyApprox on the publication dataset. Varying $|\mathcal{O}|$, $h = 0.55$, $d = 4$.

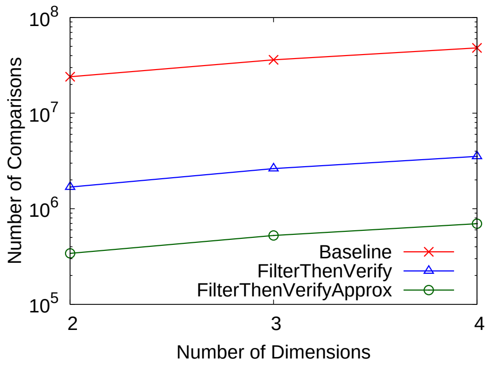
2.8.1 Experiment Setup

The algorithms were implemented in Java. The maximal heap size of Java Virtual Machine (JVM) was set to 16 GB. The experiments were conducted on a computer with 2.0 GHz Quad Core 2 Duo Xeon CPU running Ubontu 8.10.

Datasets Currently there exists no publicly available dataset that captures real users' preferences in partial orders. We thus simulated such partial orders using two real datasets of users' preferences.

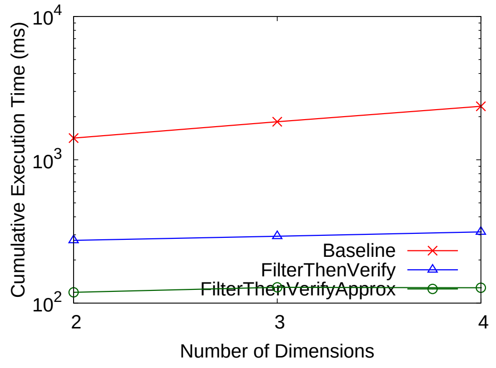


(a) Execution time

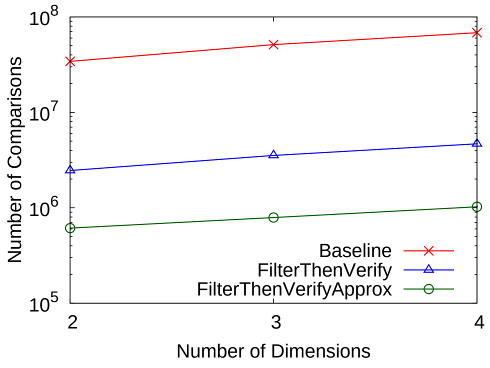


(b) Object comparisons

Figure 2.6: Comparison of Baseline, FilterThenVerify and FilterThenVerifyApprox on the movie dataset. Varying d, $|\mathcal{O}| = 12,749$, $h = 0.55$.



(a) Execution time



(b) Object comparisons

Figure 2.7: Comparison of Baseline, FilterThenVerify and FilterThenVerifyApprox on the publication dataset. Varying d, $|\mathcal{O}| = 17,598$, $h = 0.55$.

Movie Dataset We joined the Netflix dataset (`netflixprize.com`) with data from IMDB (`imdb.com`). The Netflix dataset contains the ratings (ranging from 0 to 5) given by users to movies. From IMDB we fetched the movies' attribute values, including actors, directors, genres, and writers. In this way, we found the attributes of $12,749$ Netflix movies. The goal is to, for each particular movie, identify users who may like it according to their preferences on those attributes. The mapping from our problem formulation to this dataset is the following: (i) $\mathcal{O}$ is the set of $12,749$ movies. (ii) $\mathcal{C}$ is the set of users. It includes the $1,000$ most active users based on how many movies they have rated. (iii) $\mathcal{D} = \{$actor, director, genre, writer$\}$. (iv) Given the lack of user preference data, for each attribute, the partial order corresponding to a user's preferences is simulated as follows. For two attribute values, the user's preference is based on the *average rating* and the *count* of movies satisfying these attribute values. More specifically, consider a user c who has rated m movies featuring actor a. Suppose the ratings of these movies are $r_1, r_2, \ldots, r_m$. Given c and a, the average rating is $R_a = \frac{\sum_i r_i}{m}$ and the count is $M_a = m$. Consider another actor b. If $(R_a > R_b \wedge M_a \geq M_b) \vee (R_a \geq R_b \wedge M_a > M_b)$, then $(a, b) \in \succ_c^{\mathsf{actor}}$. Intuitively, if user c watches more movies featuring a than b and gives them higher ratings, our simulation assumes the user prefers a to b.

Publication Dataset We collected from the ACM Digital Library (`dl.acm.org`) $17,598$ publications and their attributes, including affiliations, authors, conferences and topic keywords. The users are the authors themselves. The goal is to notify them about newly published articles. The recommendations are based on the users' preference relations on the attributes. The mapping from our problem formulation to this dataset is the following: (i) $\mathcal{O}$ is the set of papers. (ii) $\mathcal{C}$ is the set of authors. It includes the $1,000$ most prolific authors based on how many publications they have, similar to the $1,000$ most active users in the movie dataset. (iii) $\mathcal{D} = \{$affiliation, author, conference, keyword$\}$. The domain of attribute author is the same $1,000$ authors in $\mathcal{C}$. (iv) Given a user, the partial order on each attribute is simulated based on their preferences on the attribute values. The preference between two values on affiliation (and similarly author) is based on the *number of collaborations* between the user and the affiliation/author and the *number of citations*. For conference and keyword, the preference between two values is based on *number of publications* and *number of citations*. More specifically, consider a user c and an affiliation (or similarly another author) a. Suppose c has p_a collaborations with a and has cited articles from a q_a times. If $(p_a > p_b \wedge q_a \geq q_b) \vee (p_a \geq p_b \wedge q_a > q_b)$, then $(a, b) \in \succ_c^{\mathsf{affiliation}}$ (or $(a, b) \in \succ_c^{\mathsf{author}}$). With regard to a conference (keyword) x, suppose c has r_x publications associated with x and has cited publications associated with x s_x times. If $(r_x > r_y \wedge s_x \geq s_y) \vee (r_x \geq r_y \wedge s_x > s_y)$, then $(x, y) \in \succ_c^{\textsf{conference}}$ (or $(x, y) \in \succ_c^{\textsf{keyword}}$).

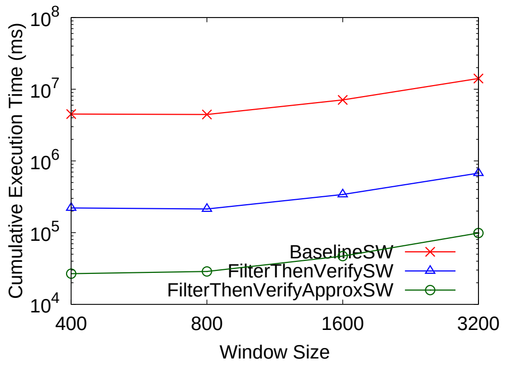
2.8.2 Baseline, FilterThenVerify, and FilterThenVerifyApprox

We conducted experiments to compare the performance of Baseline, FilterThenVerify and FilterThenVerifyApprox. For FilterThenVerify (resp. FilterThenVerifyApprox), users are clustered by the conventional hierarchical agglomerative clustering algorithm [19] using the similarity functions in Section 2.5 (resp. Section 2.6.3) and, for each cluster, it extracts the common preference relation (resp. approximate common preference relation). The experiments use three parameters which are number of objects ($|\mathcal{O}|$), number of attributes (d), and branch cut (h). In hierarchical clustering, the *branch cut* h is a threshold that controls the number of clusters by governing the minimum pairwise similarity that two clusters must satisfy in order to be merged into one cluster. The sequential order of merging clusters is depicted as a tree called *dendrogram*. The branch cut thus controls where to cut the dendrogram. In Example 9, the set of clusters are $\{\{c_1, c_2, c_5, c_6\}, \{c_3, c_4\}\}$ for $h \in (0, \frac{3}{11}]$. This is because $sim(U_4,U_2)$=0 where U_2=$\{c_3,c_4\}$ and U_4 is the cluster composed of c_1, c_2, c_5, and c_6.

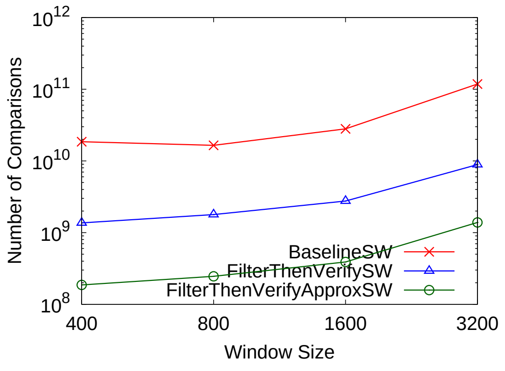
Figure 2.4a shows, for each of the three methods on the movie dataset, how its cumulative execution time (by milliseconds, in logarithmic scale) increases while the objects (i.e., movies) are sequentially processed. Figure 2.5a depicts similar behaviours of these methods on the publication dataset. Figure 2.4b and Figure2.5b, for the two datasets separately, further present the amount of work done by these methods, in terms of number of pairwise object comparisons (in logarithmic scale) for maintaining Pareto frontiers. The figures show that FilterThenVerify and FilterThenVerifyApprox beat Baseline by 1 to 2 orders of magnitude. The reason is as follows. With regard to a user c, Baseline considers all objects as candidate Pareto-optimal objects and compares all pairs. On the contrary, FilterThenVerify eliminates an object o if the corresponding common preference tuples disqualify o. FilterThenVerifyApprox incurs even less comparisons by benefiting from shared computations for clusters of users.

Figure 2.6a (Figure 2.7a) shows that the execution time of all these methods increased super-linearly by number of attributes (d). Figure 2.6b (Figure 2.7b) further reveals that the number of object comparisons also increases similarly. This is not surprising because more attributes result in larger Pareto frontiers, which makes it necessary for objects to be compared with more existing Pareto-optimal objects.

Table 2.11 reports the precision, recall and F-measure of FilterThenVerifyApprox on varying h. We can observe that, when h got smaller, the recall slowly decreased. This is expected because smaller h results in larger clusters and potentially more approximate common preference tuples for each cluster. Those approximate common preference tuples cause false negatives—the domination and elimination of objects that are instead in the Pareto frontier under the true common preference tuples, which are a subset of the approximate common preference tuples. What can be more surprising is the almost perfect precision under the various h values in Table 2.11, i.e., almost no false positives were introduced into the results. For a user c, an object o becomes a false positive if every single Pareto optimal object that dominates o becomes a false negative. As long as one of its dominating objects is not mistakenly filtered out, o will not be mistakenly introduced into the Pareto frontier. Therefore, an object is much less likely to become a false positive than a false negative. Overall, under the h values in Table 2.11, both precision and recall remain high. This may suggest that the thresholds θ_1 and θ_2 (Section 8) effectively ensure that the approximate common preference relation only includes frequent preference tuples and does not overgrow in size.

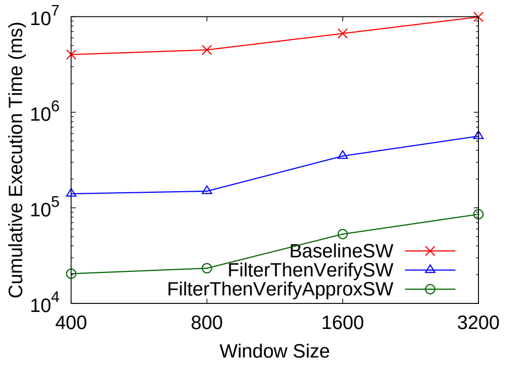


(a) Execution time

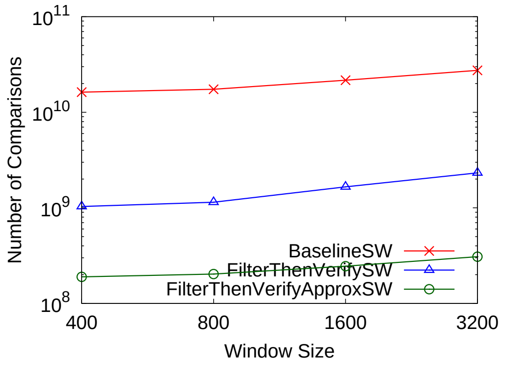


(b) Object comparisons

Figure 2.8: Effect of window size on the movie dataset. Varying W, $|\mathcal{O}| = 1$M, $h = 0.55$, $d = 4$.



(a) Execution time



(b) Object comparisons

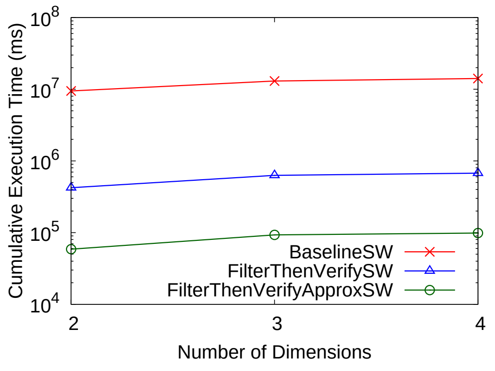
Figure 2.9: Effect of window size on the publication dataset. Varying W, $|\mathcal{O}| = 1$M, $h = 0.55$, $d = 4$.

Data stream	W	$h = 0.70$			$h = 0.65$			$h = 0.60$			$h = 0.55$		
		P	R	F	P	R	F	P	R	F	P	R	F
Movie	400	100	89.36	94.38	100	87.33	93.24	100	85.94	92.44	100	81.95	90.08
	800	100	87.87	93.54	100	85.78	92.34	100	84.04	91.33	100	80.10	88.95
	1600	100	88.65	93.98	100	86.58	92.81	100	85.01	91.90	100	81.10	89.56
	3200	99.99	94.80	97.33	100	93.08	96.41	100	92.29	95.99	100	88.99	94.17
Publication	400	100	94.58	97.21	100	93.57	96.68	100	92.98	96.36	100	92.06	95.87
	800	100	94.79	97.32	100	93.60	96.70	100	93.01	96.38	100	91.98	95.82
	1600	100	94.62	97.24	100	93.44	96.61	100	92.85	96.29	100	91.81	95.73
	3200	100	96.71	98.33	100	95.98	97.95	100	95.67	97.79	100	95.27	97.58

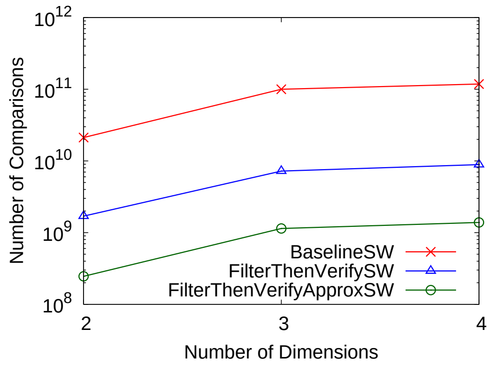
Table 2.12: The precision, recall and F-measure (in percentage) of FilterThenVerifyApproxSW. Varying W and h, $|\mathcal{O}|$=1M, d=4. (P, R, and F stand for Precision, Recall, and F-measure.)

2.8.3 BaselineSW, FilterThenVerifySW, and FilterThenVerifyApproxSW

We further compare the performance of FilterThenVerifySW and FilterThenVerifyApproxSW with BaselineSW. In this regard, we simulated two data streams—movie and pub-

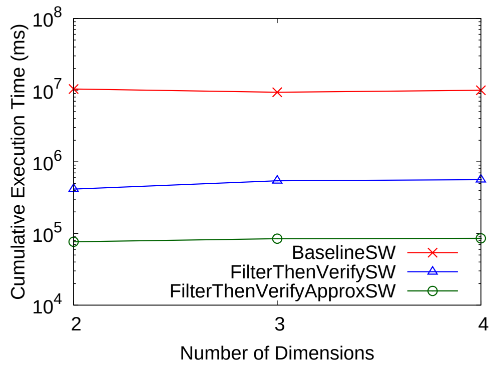


(a) Execution time

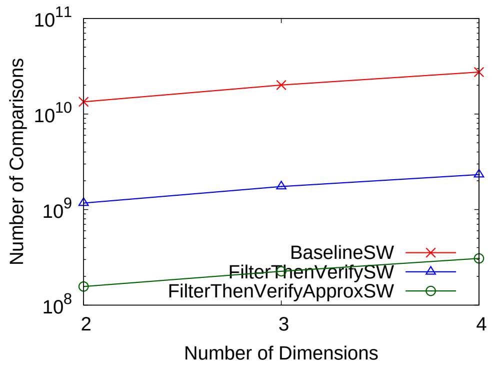


(b) Object comparisons

Figure 2.10: Comparison of BaselineSW, FilterThenVerifySW and FilterThenVerifyApproxSW on the movie dataset. Varying d, W = 3,200, $|\mathcal{O}|$ = 1M, h = 0.55.



(a) Execution time



(b) Object comparisons

Figure 2.11: Comparison of BaselineSW, FilterThenVerifySW and FilterThenVerifyApproxSW on the publication dataset. Varying d, W = 3,200, $|\mathcal{O}|$ = 1M, h = 0.55.

lication where $\mathcal{O}$ is composed of duplicated sequence of the corresponding dataset such that $|\mathcal{O}|$=1 million. Following [36], we experimented with windows of size 400, 800, $1{,}600$, and $3{,}200$, as well as report the cumulative execution times in milliseconds. In this direction, Figure2.8a demonstrates the cumulative execution times (by milliseconds, in logarithmic scale) of the aforementioned methods on the movie stream. Figure2.8a shows that the cumulative execution times increase super-linearly by W as wider window broadens the size of Pareo frontiers. These figures illustrate that both FilterThenVerifySW and FilterThenVerifyApproxSW outperformed BaselineSW by 1 to 2 orders of magnitude, which concurs with the comparative behaviours of FilterThenVerify, FilterThenVerifyApprox and Baseline. This concurrence is also applicable for the publication stream (Figure 2.9a).

Figure 2.8b (Figure 2.9b) further reveals the amount of work done by these solutions, in aspect of compared objects (in logarithmic scale) to maintain Pareto frontiers over sliding window. Moreover, Figure 2.10a (Figure 2.11a) depicts the effectiveness of FilterThenVerifySW (FilterThenVerifyApproxSW) on varying d. Figure 2.10b (Figure 2.11b) clarifies Figure 2.10a (Figure 2.11a) through illustrating the number of compared objects. The reason behind the comparative behaviour of Baseline, FilterThenVerify and FilterThenVerifyApprox is also applicable in this case. In addition, BaselineSW maintains exclusive Pareto buffer for each user ($\mathcal{PB}_c$) while FilterThenVerifySW shares a Pareto buffer across users in a cluster ($\mathcal{PB}_U$). Therefore, in sliding window protocol, the filter-then-verify approach attains the benefit of clustering in greater extent.

Table 2.12 demonstrates the precision, recall and F-measure of FilterThenVerifyApproxSW on varying W and h. We can observe that the recall declines slowly by h. Yet h does not have significant impact on the efficacy of FilterThenVerifyApproxSW. Besides, the loss of accuracy is due to false negatives rather than false positives. These behaviors concur with FilterThenVerifyApprox and the reasons behind are same as before. In addition, Table 2.12 reveals that W does not have noticeable impact on efficacy and FilterThenVerifyApprox remains effective on varying W.

2.9 Summary

We studied the problem of continuous object dissemination, which is formalized as finding the users who approve a new object in Pareto-optimality. We designed algorithm for efficient finding of target users based on sharing computation across similar preferences. To recognize users of similar preferences, we studied the novel problem of clustering users where each user's preferences are described as strict partial orders. We also presented an approximate solution of the problem of finding target users, further improving efficiency with tolerable loss of accuracy. Experimental evaluation validated the efficiency and effectiveness of our proposed solutions.

CHAPTER 3

Prominent Situational Facts

3.1 Overview

Computational journalism emerged recently as a young interdisciplinary field [13] that brings together experts in journalism, social sciences and computer science, and advances journalism by innovations in computational techniques. Database and data mining researchers have also started to push the frontiers of this field [14, 21, 46, 50, 34, 20, 40, 47, 18]. One of the goals in computational journalism is *newsworthy fact discovery*. Reporters always try hard to bring out attention-seizing factual statements backed by data, which may lead to news stories and investigation. While such statements take many different forms, we consider a common form exemplified by the following excerpts from real-world news media:

- "Paul George had 21 points, 11 rebounds and 5 assists to become the first Pacers player with a 20/10/5 (points/rebounds /assists) game against the Bulls since Detlef Schrempf in December 1992." (`http://espn.go.com/espn/elias?date=20130205`)
- "The social world's most viral photo ever generated 3.5 million likes, 170,000 comments and 460,000 shares by Wednesday afternoon." (`http://cnbc.com/id/49728455/President_Obama_Sets_New_Social_Media_Record`)

What is common in the above two statements is a prominent fact with regard to a context and several measures. In the first statement, the context includes the performance of Pacers players in games against the Bulls since December 1992 and the measures are points, rebounds, assists. By these measures, no performance in the context is better than the mentioned performance of Paul George. For the second statement, the measures

tuple id	player	day	month	season	team	opp_team	points	assists	rebounds
t_1	Bogues	11	Feb.	1991-92	Hornets	Hawks	4	12	5
t_2	Seikaly	13	Feb.	1991-92	Heat	Hawks	24	5	15
t_3	Sherman	7	Dec.	1993-94	Celtics	Nets	13	13	5
t_4	Wesley	4	Feb.	1994-95	Celtics	Nets	2	5	2
t_5	Wesley	5	Feb.	1994-95	Celtics	Timberwolves	3	5	3
t_6	Strickland	3	Jan.	1995-96	Blazers	Celtics	27	18	8
t_7	Wesley	25	Feb.	1995-96	Celtics	Nets	12	13	5

* Attribute opp_team is the short form of opposition team.

Table 3.1: A Mini-world of Basketball Gamelogs

are likes, comments, shares and the context includes all photos posted to Facebook. The story is that no photo in the context attracted more attention than the mentioned photo of President Barack Obama, by the three measures. In general, facts can be put in many contexts, such as photos posted in 2012, photos posted by political campaigns, and so on.

Similar facts can be stated on data from domains outside of sports and social media, including stock data, weather data, and criminal records. For example: 1) "Stock A becomes the first stock in history with price over \$300 and market cap over \$400 billion." 2) "Today's measures of wind speed and humidity are x and y, respectively. City B has never encountered such high wind speed and humidity in March." 3) "There were 35 DUI arrests and 20 collisions in city C yesterday, the first time in 2013." Some of these facts are not only interesting to reporters but also useful to financial analysts, scientists, and citizens.

In technical terms, a fact considered in this chapter is a *contextual skyline* object that stands out against other objects in a context with regard to a set of measures. Consider a table R whose schema includes a set of measure attributes $\mathcal{M}$ and a set of dimension attributes $\mathcal{D}$. A context is a subset of R, resulting from a conjunctive constraint defined on a subset of the dimension attributes $D \subseteq \mathcal{D}$. A measure subspace is defined by a subset of the measure attributes $M \subseteq \mathcal{M}$. A tuple t is a contextual skyline tuple if no other tuple in the context dominates t. A tuple t' dominates t if t' is better than or equal to t on every attribute in M and better than t on at least one of the attributes. Such is the standard notion of dominance relation adopted in *skyline analysis* [9].

We study how to find *situational facts* pertinent to new tuples in an ever-growing database, where the tuples capture real-world events. We propose algorithms that, whenever a new tuple t enters an append-only table R, discover constraint-measure pairs that qualify t as a contextual skyline tuple. Each such pair constitutes a situational fact pertinent to t's arrival.

Example 17. Consider the mini-world of basketball gamelogs R in Table 3.1, where $\mathcal{D}$ = {player, month, season, team, opp_team} and $\mathcal{M}$={points, assists, rebounds}. The existing tuples are t_1 to t_6 and the new tuple is t_7. If the context is the whole table (i.e., no constraint) and the measure subspace M=$\mathcal{M}$, t_7 is not a skyline tuple since it is dominated by t_3 and t_6. However, with regard to context $\sigma_{\mathsf{month}=\textit{Feb.}}(R)$ (corresponding to constraint month=*Feb.*) and the same measure subspace M, t_7 is in the skyline along with t_2. In yet another context $\sigma_{\mathsf{team}=\textit{Celtics}\wedge\ \mathsf{opp\_team}=\textit{Nets}}(R)$ under measure subspace M={assists, rebounds}, t_7 is in the skyline along with t_3. Tuple t_7 is also a contextual skyline tuple for other constraint-measure pairs, which we do not further enumerate. ■

Discovering situational facts is challenging as timely discovery of such facts is expected. In finding news leads centered around situational facts, the value of a news piece diminishes rapidly after the event takes place. Consider NBA games again. Sports media need to identify and discuss sensational records quickly as they emerge; any delay makes fans less interested in the records and risks losing them to rival media. Timely identification of situational facts is also critical in areas beyond journalism. To make informed investment decisions, investors want to know facts related to stock trading as soon as possible. Facts discovered from weather data can assist scientists in identifying extreme weather conditions and help government and the public in coping with the weather.

Simple situational facts on a single measure and a complete table, e.g., the all-time NBA scoring record, can be conveniently detected by database triggers. However, general and complex facts involving multiple dimension and measure attributes are much harder to discover. Exhaustively using triggers leads to an exponential explosion of constraint-measure pairs to check for each new tuple. In reality, news media relies on instincts and experiences of domain experts on this endeavor. The experts, impressed by an event such as the outstanding performance of a player in a game, hypothesize a fact and manually craft a database query to check it. This is how Elias Sports Bureau tackles the task and provides sports records (such as the aforementioned one by Paul George) to many sports media [2]. With ever-growing data and limited human resources, such manual checking is time-consuming and error-prune. Its low efficiency not only leads to delayed and missing facts, but also ties up precious human expertise that could be otherwise devoted to more important journalistic activities.

The technical focus of this chapter is thus on efficient automatic approach to discovering situational facts, i.e., finding constraint-measure pairs that qualify a new tuple t as a contextual skyline tuple. A straightforward brute-force approach would compare t with every historical tuple to determine if t is dominated, repeatedly for every conjunctive constraint satisfied by t under every possible measure subspace. The obvious low-efficiency of this approach has three culprits—exhaustive comparison with *every tuple*, under *every constraint*, and over *every measure subspace*. We thus design algorithms to counter these issues by three corresponding ideas, as follows:

1) Tuple reduction Instead of comparing t with every previous tuple, it is sufficient to only compare with current skyline tuples. This is based on the simple property that, if any tuple dominates t, then there must exist a skyline tuple that also dominates t. For example, in Table 3.1, under constraint month=*Feb.* and the full measure space $\mathcal{M}$, the corresponding context contains t_1, t_2, t_4 and t_5, and the contextual skyline has two tuples—

t_1 and t_2. When the new tuple t_7 comes, with regard to the same constraint-measure pair, it suffices to compare t_7 with t_1 and t_2, not the remaining tuples.

2) Constraint pruning If t is dominated by t' in a particular measure subspace M, then t does not belong to the contextual skyline of constraint-measure pair (C, M) for any C satisfied by both t and t'. For example, since t_7 is dominated by t_3 in the full measure space $\mathcal{M}$, it is not in the contextual skylines for (team=*Celtics* $\wedge$ opp_team=*Nets*, $\mathcal{M}$), (team=*Celtics*, $\mathcal{M}$), (opp_team=*Nets*, $\mathcal{M}$) and (no constraint, $\mathcal{M}$). Furthermore, since t_7 is dominated by t_6 in $\mathcal{M}$, it does not belong to the contextual skylines for (season = *1995-96*, $\mathcal{M}$) and (no constraint, $\mathcal{M}$). Based on this, we examine the constraints satisfied by t in a certain order, such that comparisons of t with skyline tuples associated with already examined constraints are used to prune remaining constraints from consideration.

3) Sharing computation across measure subspaces As repeatedly visiting the constraints satisfied by t for every measure subspace is wasteful, we pursue sharing computation across different subspaces. The challenge in such sharing lies in the anti-monotonicity of dominance relation—a skyline tuple in space M may or may not be in the skyline of a superspace or subspace M' [30]. Nonetheless, we can first consider the full space $\mathcal{M}$ and prune various constraints from consideration for smaller subspaces. For instance, after comparing t_7 with t_2 in $\mathcal{M}$, the algorithms realize that t_7 has smaller values on points and rebounds. It is dominated by t_2 in three subspaces—{points, rebounds}, {points} and {rebounds}. When considering these subspaces, we can skip two contexts—corresponding to constraint month=*Feb.* and empty constraint, respectively—as t_2 and t_7 are in both contexts.

It is crucial to report truly *prominent* situational facts. A newly arrived tuple t may be in the contextual skylines for many constraint-measure pairs. Reporting all of them will overwhelm users and make important facts harder to spot. We measure the *prominence* of a constraint-measure pair by the cardinality ratio of all tuples to skyline tuples in the corresponding context. The intuition is that, if t is one of the very few skyline tuples in a context containing many tuples under a measure subspace, then the corresponding constraint-measure pair brings out a prominent fact. We thus rank all situational facts pertinent to t in descending order of prominence. Reporters and experts can choose to investigate top-*k* facts or the facts with prominence values above a threshold.

Any measure value of a tuple can be the accumulation of the corresponding measure within a time frame. For instance, while Table 3.1 demonstrates a mini-world of basketball gamelogs, each of its measure value is the cumulative measure with regard to each instant of time in the corresponding match. In order to discover the newsworthy facts synchronously with a live event or stream live news, we further consider *instant* dataset and incorporate the algorithms with this framework.

The contributions of this chapter are summarized as follows:

- We study the novel problem of finding situational facts and formalize it as discovering constraint-measure pairs that qualify a tuple as a contextual skyline tuple (Section 3.3).
- We devise efficient algorithms based on three main ideas—tuple reduction, constraint pruning and sharing computation across measure subspaces (Section 3.5).
- We further extend the algorithms to consider an instant dataset as input and thus allow live news streaming (Section 3.6).
- We use a simple prominence measure for ranking situational facts and discovering prominent situational facts (Section 3.7).
- We conduct extensive experiments on three real datasets (two NBA datasets and weather dataset) to investigate their prominent situational facts and to study the efficiency of various proposed algorithms and their tradeoffs (Section 3.8).

3.2 Related Work

Pioneers in data journalism have considerable success in using computer programs to write stories about sports games and stock earnings (e.g., StatSheet `http://statsheet.com` and Narrative Science `http://narrativescience.com`). The stories follow writing patterns to narrate box scores and play-by-play data and a company's earnings data. They focus on capturing what happened in the game or what the earnings numbers indicate. They do not find situational facts pertinent to a game or an earnings report in the context of historical data.

Skyline query is an optimization problem extensively investigated in recent years [22], since Börzsönyi et al. [9] brought the concept to the database field. In [9] and the studies afterwards, it is assumed both the context of tuples in comparison and the measure space are given as query conditions. A high-level perspective on what distincts our work is—while prior studies *find answers* (i.e., skyline points) for a given query (i.e., a context, a measure space, or their combination), we study the reverse problem of *finding queries* (i.e., constraint-measure pairs that qualify a tuple as a contextual skyline tuple, among all possible pairs) for a particular answer (i.e., a new tuple). In other words, we find ways to make a particular tuple stand out.

From a technical perspective, Table 3.2 summarizes the differences among the more relevant previous studies and this chapter, along three aspects—whether they consider all possible contexts defined on dimension attributes, all measure subspaces, and incremental computation on dynamic data. With regard to context, Zhang et al. [51] integrate the evaluation of a constraint with finding skyline tuples in the corresponding context in a given measure space. With regard to measure, Pei et al. [30] compute on static data the *skycube*—skyline points in all measure subspaces. Xia et al. [48] studied how to update a compressed skycube (CSC) when data change. The CSC stores a tuple t in its *minimum subspaces*—the measure subspaces in which t is a skyline tuple and of which the subspaces do not contain

t in the skyline. They proposed an algorithm to update CSC when new tuples come and also an algorithm to use CSC to find all skyline tuples for a given measure subspace.

We can adapt [48] to find situational facts. While Section 3.8 provides experimental comparisons with the adaptation, here we analyze its shortcomings. Since [48] does not consider different contexts, the adaptation entails maintaining a separate CSC for every possible context. Upon the arrival of a new tuple t, for every context, the adaptation will update the corresponding CSC. Since a CSC only stores t in its minimum subspaces, the adaptation needs to run their query algorithm to find the skyline tuples for all measure subspaces, in order to determine if t is one of the skyline tuples. This is clearly an overkill, caused by that CSC is designed for finding all skyline tuples. Furthermore, while our algorithms can share computation across measure subspaces, there does not appear to be an effective strategy to share the computation of CSC algorithms across different contexts.

Promotion analysis by ranking [45] finds the contexts (given by conjunctive constraints on dimension attributes) in which an object is ranked high. It has two key differences from our work. (i) It ranks objects by a single score attribute, while we define object dominance relation on multiple measure attributes and consider all measure subspaces. A well-known merit of the concept of skyline, in comparison with ranking, is that it removes the burden of defining ranking functions from users. Its result is also intuitive to explain. This is particularly appealing to computational journalism. (ii) It considers one-shot computation on static data, while we focus on incremental discovery on dynamic data. Due to these distinctions, the algorithmic approaches in the two works are also fundamentally different.

Wu et al. [46] studied the *one-of-the-τ object* problem, which entails finding the largest k value and the corresponding *k-skyband objects* (objects dominated by less than k other objects) such that there are no more than τ k-skyband objects. They consider all

	all possible contexts	measure subspaces	incremental
[51]	no	no	no
[30]	no	yes	no
[48]	no	yes	yes
[45]	yes	no	no
[46]	no	yes	no
[4]	no	no	yes
this work	yes	yes	yes

Table 3.2: Comparing Related Work on Three Modeling Aspects

measure subspaces but not different contexts formed by constraints. Similar to [45], it focuses on static data.

Alvanaki et al. [4] worked on detecting interesting events through monitoring changes in ranking, by using materialized view maintenance techniques. The work focuses on top-k queries on single ranking attribute rather than skyline queries defined on multiple measure attributes. Their ranking contexts have at most three constraints. The work is similar to [8] which studied how to predict significant events based on historical data and correspondingly perform lazy maintenance of ranking views on a database.

3.3 Problem Statement

This section provides a formal description of our data model and problem statement. Table 3.3 lists the major notations. Consider a relational schema $R(\mathcal{D};\mathcal{M})$, where the *dimension space* is a set of *dimension attributes* $\mathcal{D}$=$\{d_1,\ldots,d_n\}$ on which *constraints* are specified, and the *measure space* is a set of *measure attributes* $\mathcal{M}$=$\{m_1,\ldots,m_s\}$ on which dominance relation for skyline operation is defined. Any set of dimension attributes $D \subseteq \mathcal{D}$ defines a *dimension subspace* and any set of measure attributes $M \subseteq \mathcal{M}$ defines a *measure subspace*. In Table 3.4, $R(\mathcal{D};\mathcal{M})$ = $\{t_1,t_2,t_3,t_4,t_5\}$, $\mathcal{D}$ = $\{d_1,d_2,d_3\}$, $\mathcal{M}$=$\{m_1,m_2\}$. We will use this table as a running example.

$R(\mathcal{D};\mathcal{M})$	relation R, dimension space $\mathcal{D}$, measure space $\mathcal{M}$
$D \subseteq \mathcal{D}$	dimension subspace
$M \subseteq \mathcal{M}$	measure subspace
C	constraint
$(\mathcal{C}_{\mathcal{D}}, \trianglelefteq)$	poset of all constraints on subsumption relation $\trianglelefteq$
$C_1 \lhd (\trianglelefteq) C_2$	C_1 is subsumed by (subsumed by or equal to) C_2
$t_1 \prec (\preceq) t_2$	t_1 is dominated by (dominated by or equal to) t_2
$\sigma_C(R)$	tuples in R satisfying constraint C
$\lambda_M(R)$	skyline tuples in R on measure subspace M
$\lambda_M(\sigma_C\ (R))$	contextual skyline of R with respect to C and M
$\mu_{C,M}$	tuples stored with respect to C and M
S^t	contextual skylines for t
$\mathcal{C}^t_{\mathcal{D}}$ or $\mathcal{C}^t$	tuple-satisfied constraints of t
$\top$	the top element of lattice $(\mathcal{C}^t_{\mathcal{D}}, \trianglelefteq)$ and poset $(\mathcal{C}_{\mathcal{D}}, \trianglelefteq)$
$\bot(\mathcal{C}^t_{\mathcal{D}})$	the bottom element of lattice $(\mathcal{C}^t_{\mathcal{D}}, \trianglelefteq)$
$\mathcal{A}_C$,$\mathcal{D}_C$,$\mathcal{P}_C$,$\mathcal{CH}_C$	C's ancestors, descendants, parents, children in $\mathcal{C}_{\mathcal{D}}$
$\mathcal{A}^t_C$,$\mathcal{D}^t_C$,$\mathcal{P}^t_C$,$\mathcal{CH}^t_C$	C's ancestors, descendants, parents, children in $\mathcal{C}^t_{\mathcal{D}}$
$\mathcal{C}^{t_1,t_2}$	the intersection of $\mathcal{C}^{t_1}$ and $\mathcal{C}^{t_2}$
$\mathcal{SC}^t_M$	the skyline constraints of t in M
$\mathcal{MSC}^t_M$	the maximal skyline constraints of t in M

Table 3.3: Notations

id	d_1	d_2	d_3	m_1	m_2
t_1	a_1	b_2	c_2	10	15
t_2	a_1	b_1	c_1	15	10
t_3	a_2	b_1	c_2	17	17
t_4	a_2	b_1	c_1	20	20
t_5	a_1	b_1	c_1	11	15

Table 3.4: Running Example

Definition 11 (Constraint)**.** A *constraint* C on dimension space $\mathcal{D}$ is a conjunctive expression of the form d_1=$v_1 \wedge d_2$=$v_2 \wedge \ldots \wedge d_n$=$v_n$ (also written as $\langle v_1, v_2, \ldots, v_n \rangle$ for simplicity), where $v_i \in dom(d_i) \cup \{*\}$ and $dom(d_i)$ is the value domain of dimension attribute d_i. We use $C.d_i$ to denote the value v_i assigned to d_i in C. If $C.d_i$=$*$, we say d_i is *unbound*, i.e., no condition is specified on d_i. We denote the number of bound attributes in C as $bound(C)$.

id	d_1	d_2	d_3	m_1	m_2	$id.prev$
t_1	a_1	b_2	c_2	10	5	null
t_2	a_1	b_1	c_1	15	10	null
t_3	a_2	b_1	c_1	19	3	null
t_4	a_2	b_1	c_1	20	3	t_3
t_5	a_1	b_2	c_1	13	10	null
t_6	a_1	b_1	c_2	14	11	null
t_7	a_1	b_1	c_1	16	10	t_2
t'_7	a_1	b_1	c_1	15	11	t_2

Table 3.5: Running Example on Instant Data

The set of all possible constraints over dimension space $\mathcal{D}$ is denoted $\mathcal{C}_{\mathcal{D}}$. Clearly, $|\mathcal{C}_{\mathcal{D}}| = \prod_i (|dom(d_i)| + 1)$.

Given a constraint $C \in \mathcal{C}_{\mathcal{D}}$, $\sigma_C(R)$ is the relational algebra expression that chooses all tuples in R that satisfy C. ■

Example 18. For Table 3.4, an example constraint is $C = \langle a_1, *, c_1 \rangle$ in which d_2 is unbound. $\sigma_C(R) = \{t_2, t_5\}$. ■

Definition 12 (Skyline)**.** Given a measure subspace M and two tuples $t, t' \in R$, t *dominates* t' with respect to M, denoted by $t \succ_M t'$ or $t' \prec_M t$, if t is equal to or better than t' on all attributes in M and t is better than t' on at least one attribute in M. A tuple t is a *skyline tuple* in subspace M if it is not dominated by any other tuple in R. The set of all skyline tuples in R with respect to M is denoted by $\lambda_M(R)$, i.e., $\lambda_M(R)$=$\{t \in R | \nexists t' \in R$ s.t. $t' \succ_M t\}$. ■

We use the general term "better than" in Definition 21, which can mean either "larger than" or "smaller than" for numeric attributes and either "ordered before" or "ordered after" for ordinal attributes, depending on applications. Further, the preferred ordering of values on different attributes are allowed to be different. For example, in a basketball game, 10 points is better than 5 points, while 3 fouls is worse than 1 foul. Without loss of generality, we assume measure attributes are numeric and a larger value is better than a smaller value.

Algorithm 7: Find $\mathcal{C}^t$

```
Input: $t \in R$
Output: $\mathcal{C}^t$: constraints satisfied by $t$
1  $\mathcal{C}^t \leftarrow \emptyset$;
2  $Q \leftarrow \emptyset$; $Q.enqueue(\top)$;
3  while not $Q.empty()$ do
4      $C \leftarrow Q.dequeue()$;
5      $\mathcal{C}^t \leftarrow \mathcal{C}^t \cup \{C\}$;
6      $i \leftarrow n$;
7      while $i > 0$ and $C.d_i = *$ do
8          $C' \leftarrow C$;
9          $C'.d_i \leftarrow t.d_i$;
10         $Q.enqueue(C')$;
11         $i \leftarrow i - 1$;
12 return $\mathcal{C}^t$;
```

Definition 13 (Contextual Skyline)**.** Given a relation $R(\mathcal{D};\ \mathcal{M})$, the *contextual skyline* under constraint $C \in \mathcal{C}_\mathcal{D}$ over measure subspace $M \subseteq \mathcal{M}$, denoted $\lambda_M(\sigma_C(R))$, is the skyline of $\sigma_C(R)$ in M. ■

Example 19. For Table 3.4, if $M = \mathcal{M}$, $\lambda_M(R) = \{t_4\}$. In fact, t_4 dominates all other tuples in space M. If the constraint is $C = \langle a_1, b_1, c_1\rangle$, $\sigma_C(R) = \{t_2, t_5\}$, $\lambda_M(\sigma_C(R)) = \{t_2, t_5\}$ for $M = \mathcal{M}$, and $\lambda_M(\sigma_C(R)) = \{t_2\}$ for $M = \{m_1\}$. ■

Problem Statement Given an append-only table $R(\mathcal{D};\mathcal{M})$ and the last tuple t that was appended onto R, the *situational fact discovery problem* is to find each constraint-measure pair (C,M) such that t is in the contextual skyline. The result, denoted S^t, is $\{(C,M) | C \in \mathcal{C}_\mathcal{D},\ M \subseteq \mathcal{M},\ t \in \lambda_M(\sigma_C(R))\}$. For simplicity of notation, we call S^t "the contextual

Algorithm 8: BruteForce

```
Input: $R(\mathcal{M}, \mathcal{D})$: existing tuples; $t$: the new tuple
Output: $S^t$: the contextual skylines for $t$
1  $S^t \leftarrow \emptyset$;
2  foreach $M \subseteq \mathcal{M}$ do
3      foreach $C \in \mathcal{C}^t$ do
4          $pruned \leftarrow$ false;
5          foreach $t' \in R$ do
6              if $t \prec_M t'$ and $t' \in \sigma_C(R)$ then
7                  $pruned \leftarrow$ true;
8                  break;
9          if not $pruned$ then $S^t \leftarrow S^t \cup \{(C, M)\}$;;
10 $R \leftarrow R \cup \{t\}$;
11 return $S^t$;
```

skylines for t", even though rigorously speaking it is the set of (C, M) pairs whose corresponding contextual skylines include t.

3.4 Solution Overview

Discovering situational facts for a new tuple t entails finding constraint-measure pairs that qualify t as a contextual skyline tuple. We identify three sources of inefficiency in a straightforward brute-force method, and we propose corresponding ideas to tackle them. To facilitate the discussion, we define the concept of *tuple-satisfied constraints*, which are all constraints pertinent to t, corresponding to the contexts containing t.

Definition 14 (Tuple-Satisfied Constraint)**.** Given a tuple $t \in R(\mathcal{D}; \mathcal{M})$ and a constraint $C \in \mathcal{C}_{\mathcal{D}}$, if $\forall d_i \in \mathcal{D}$, $C.d_i = *$ or $C.d_i = t.d_i$, we say t *satisfies* C. We denote the set of

Algorithm 9: BaselineSeq

```
Input: $R(\mathcal{M}, \mathcal{D})$: existing tuples; $t$: the new tuple
Output: $S^t$: the contextual skylines for $t$
1 $S^t \leftarrow \emptyset$;
2 foreach $M \subseteq \mathcal{M}$ do
3     $S \leftarrow \mathcal{C}^t$;
4     foreach $t' \in R$ do
5         if $t \prec_M t'$ then $S \leftarrow S - \mathcal{C}^{t,t'}$;;
6     foreach $C \in S$ do
7         $S^t \leftarrow S^t \cup \{(C, M)\}$;
8 $R \leftarrow R \cup \{t\}$;
9 return $S^t$;
```

all such satisfied constraints by $\mathcal{C}^t_{\mathcal{D}}$ or simply $\mathcal{C}^t$ when $\mathcal{D}$ is clear in context. It follows that given any $C \in \mathcal{C}^t$, $t \in \sigma_C(R)$. ■

For $C{\in}\mathcal{C}^t$, $C.d_i$ can attain two possible values $\{*, t.d_i\}$. Hence, $\mathcal{C}^t$ has 2^n constraints in total for $|\mathcal{D}|{=}n$. Algorithm is a simple routine used in all algorithms for finding all constraints of $\mathcal{C}^t$. It generates the constraints from the most general constraint $\top{=}\langle *, *, \ldots, * \rangle$ to the most specific constraint $\langle t.d_1, t.d_2, \ldots, t.d_n \rangle$. $\top$ has no bound attributes, i.e., $bound(\top){=}0$. Algorithm makes sure a constraint is not generated twice, for efficiency, by not continuing the while-loop in Line 7 once a specific attribute value is found in C.

A brute-force approach to the contextual skyline discovery problem would compare a new tuple t with every tuple in R to determine if t is dominated, repeatedly for every constraint satisfied by t in every possible measure subspace. It is shown in Algorithm. The obvious inefficiency of this approach has three culprits—the exhaustive comparison with *every tuple*, for *every constraint* and in *every measure subspace*. We devise three corresponding ideas to counter these causes, as follows:

(1) Tuple reduction For a constraint-measure pair (C, M), t is in the contextual skyline $\lambda_M(\sigma_C(R))$ if t belongs to $\sigma_C(R)$ and is not dominated by any tuple in $\sigma_C(R)$. Instead of comparing t with every tuple, it suffices to only compare with current skyline tuples. This simple optimization is based on the following proposition which ways, if any tuple dominates t, there must exist a skyline tuple that also dominates t.

Proposition 1. *Given a new tuple t inserted into R, a constraint $C \in \mathcal{C}^t$ and a measure subspace M, $t \in \lambda_M(\sigma_C(R))$ if and only if $\nexists\, t' \in \lambda_M(\sigma_C(R))$ such that $t' \succ_M t$.* ■

To exploit this idea, our algorithms conceptually maintain the contextual skyline tuples for each context (i.e., measure subspace and constraint), and compare t only with these tuples for constraints that t satisfies.

(2) Constraint pruning For constraints satisfied by t, we need to determine whether t enters the contextual skyline. To prune constraints from consideration, we note the following property: if t is dominated by a skyline tuple t' under measure subspace M, t is not in the contextual skyline of constraint-measure pair (C, M) for any C satisfied by both t and t'.

To enable constraint pruning, we organize all constraints in $\mathcal{C}^t$ into a lattice by their subsumption relation. The constraints satisfied by both t and t', denoted $\mathcal{C}^{t,t'}$, also form a lattice, which is the intersection of lattices $\mathcal{C}^t$ and $\mathcal{C}^{t'}$. Below we formalize the concepts of lattice and lattice intersection.

Definition 15 (Constraint Subsumption). Given $C_1, C_2 \in \mathcal{C}_\mathcal{D}$, C_1 is subsumed by or equal to C_2 (denoted $C_1 \trianglelefteq C_2$ or $C_2 \trianglerighteq C_1$) iff

1. $\forall d_i \in \mathcal{D}$, $C_2.d_i = C_1.d_i$ or $C_2.d_i = *$.

C_1 is subsumed by C_2 (denoted $C_1 \lhd C_2$ or $C_2 \rhd C_1$) iff $C_1 \trianglelefteq C_2$ but $C_1 \neq C_2$. In other words, the following condition is also satisfied in addition to the above one—

2. $\exists d_i \in \mathcal{D}$ such that $C_2.d_i{=}*$ and $C_1.d_i{\neq}*$, i.e, d_i is bound to a value belonging to $dom(d_i)$ in C_1 but is unbound in C_2.

By definition, $\sigma_{C_1}(R) \subseteq \sigma_{C_2}(R)$ if $C_1 \trianglelefteq C_2$. ■

Example 20. Consider C_1 = $\langle a, b, c\rangle$ and C_2 = $\langle a, *, c\rangle$. Here $C_1.d_1$ = $C_2.d_1$, $C_1.d_3$ = $C_2.d_3$, $C_1.d_2{=}b$ and $C_2.d_2$ = $*$. By Definition 15, C_1 is subsumed by C_2, i.e. $C_1 \triangleleft C_2$. ■

Definition 16 (Partial Order on Constraints)**.** The subsumption relation $\trianglelefteq$ on $\mathcal{C}_\mathcal{D}$ forms a partial order. The partially ordered set (poset) ($\mathcal{C}_\mathcal{D}$, $\trianglelefteq$) has a *top element* $\top = \langle *, *, \ldots, *\rangle$ that subsumes every other constraint in $\mathcal{C}_\mathcal{D}$. $\top$ is the most general constraint, since it has no bound attributes. Note that ($\mathcal{C}_\mathcal{D}$, $\trianglelefteq$) is not a lattice and does not have a single bottom element. Instead, it has multiple *minimal elements*. Every minimal element C satisfies the condition that $\forall d_i$, $C.d_i \neq *$.

If $C_1 \triangleleft C_2$, we say C_1 is a descendant of C_2 (C_2 is an ancestor of C_1). If $C_1 \triangleleft C_2$ and $bound(C_1) - bound(C_2) = 1$, then C_1 is a child of C_2 (C_2 is a parent of C_1). Given $C \in \mathcal{C}_\mathcal{D}$, we denote C's ancestors, descendants, parents and children by $\mathcal{A}_C$, $\mathcal{D}_C$, $\mathcal{P}_C$ and $\mathcal{CH}_C$, respectively. ■

Definition 17 (Lattice of Tuple-Satisfied Constraints)**.** Given $t{\in}R(\mathcal{D};\mathcal{M})$, $\mathcal{C}^t \subseteq \mathcal{C}_\mathcal{D}$ by definition. In fact, ($\mathcal{C}^t$, $\trianglelefteq$) is a lattice. Its top element is $\top$. Its bottom element $\langle t.d_1$, $t.d_2$, $\ldots$, $t.d_n\rangle$, denoted $\perp(\mathcal{C}^t)$, is a minimal element in $\mathcal{C}_\mathcal{D}$.

Given $C{\in}\mathcal{C}^t$, we denote C's ancestors, descendants, parents and children within $\mathcal{C}^t$ by $\mathcal{A}^t_C$, $\mathcal{D}^t_C$, $\mathcal{P}^t_C$ and $\mathcal{CH}^t_C$, respectively. $|\mathcal{CH}^t_C|{=}n{-}bound(C)$ where $n{=}|\mathcal{D}|$, i.e., each child of C is a constraint by adding conjunct $d_i{=}t.d_i$ into C for unbound attribute d_i. It is clear that $|\mathcal{P}^t_C|{=}bound(C)$. By definition, $\mathcal{A}^t_C{=}\mathcal{A}_C$ and $\mathcal{P}^t_C{=}\mathcal{P}_C$, while $\mathcal{D}^t_C{\subseteq}\mathcal{D}_C$ and $\mathcal{CH}^t_C{\subseteq}\mathcal{CH}_C$. ■

Example 21. Figure3.1 presents lattice $\mathcal{C}^{t_5}$ for t_5 in Table 3.4. For simplicity, we omit values on unbound dimension attributes (e.g., $\langle$ *, *, $c_1\rangle$ is represented as c_1). Consider C

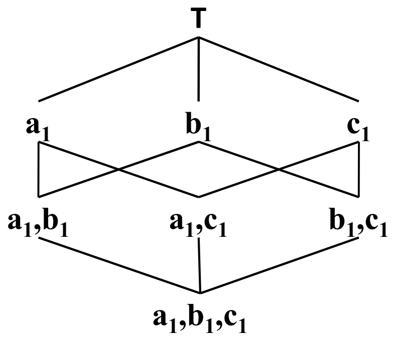


Figure 3.1: Lattice $\mathcal{C}^{t_5}$

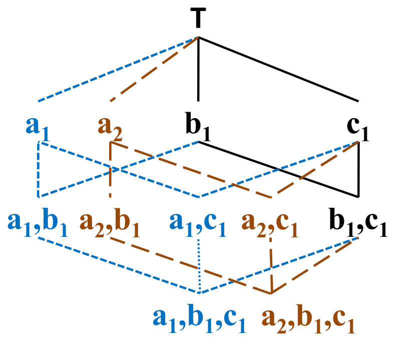


Figure 3.2: Intersection of $\mathcal{C}^{t_4}$ and $\mathcal{C}^{t_5}$

= $\langle a_1, *, c_1\rangle$. $\mathcal{A}_C^{t_5}$ = $\{\top, \langle a_1, *, * \rangle, \langle *, *, c_1\rangle\}$, $\mathcal{P}_C^{t_5}$ = $\{\langle a_1, *, * \rangle, \langle *, *, c_1\rangle\}$, $\mathcal{CH}_C^{t_5}$ = $\{\langle a_1, b_1, c_1\rangle\}$ and $\mathcal{D}_C^{t_5}$ = $\{\langle a_1, b_1, c_1\rangle\}$. ■

Definition 18 (Lattice Intersection). Given $t, t' \in R(\mathcal{D}; \mathcal{M})$, $\mathcal{C}^{t,t'}$ = $\mathcal{C}^t \cap \mathcal{C}^{t'}$ is the intersection of lattices $\mathcal{C}^t$ and $\mathcal{C}^{t'}$. $\mathcal{C}^{t,t'}$ is non-empty and is also a lattice. By Definition 17, the lattices for all tuples share the same top element $\top$. Hence $\top$ is also the top element of $\mathcal{C}^{t,t'}$. Its bottom $\perp(\mathcal{C}^{t,t'})$ = $\langle v_1, v_2, \ldots, v_n\rangle$ where $v_i{=}t.d_i$ if $t.d_i{=}t'.d_i$ and $v_i{=}*$ otherwise. $\perp(\mathcal{C}^{t,t'})$ equals $\top$ when t and t' do not have common attribute value. ■

Example 22. Figure3.2 shows $\mathcal{C}^{t_4}$ and $\mathcal{C}^{t_5}$ for t_4 and t_5 in Table 3.4. The constraints connected by solid lines represent the lattice intersection $\mathcal{C}^{t_4,t_5}$. Its bottom is $\perp(\mathcal{C}^{t_4,t_5})$ = $\langle *, b_1, c_1\rangle$. In addition to $\mathcal{C}^{t_4,t_5}$, $\mathcal{C}^{t_4}$ and $\mathcal{C}^{t_5}$ further include the constraints connected by dashed and dotted lines, respectively. ■

The algorithms we are going to propose consider the constraints in certain lattice order, compare t with skyline tuples associated with visited constraints, and use t's dominating tuples to prune unvisited constraints from consideration—thereby reducing cost. This idea of lattice-based pruning of constraints is justified by Propositions 2 and 3 below.

Proposition 2. *Given a tuple t, if $t \notin \lambda_M(\sigma_C(R))$, then $t \notin \lambda_M(\sigma_{C'}(R))$, for all $C' \in \mathcal{A}_C$.* ■

If $t \prec_M t'$, then $t \notin \lambda_M(\sigma_{\perp(\mathcal{C}^{t,t'})}(R))$. Hence, according to Proposition 2, we have the following Proposition 3.

Proposition 3. *Given two tuples t and t', if $t \prec_M t'$, then $t \notin \lambda_M(\sigma_C\ (R))$, for all $C \in \mathcal{C}^{t,t'}$.* ■

(3) Sharing computation across measure subspaces Given t, we need to consider not only all constraints satisfied by t, but also all possible measure subspaces. Sharing computation across measure subspaces is challenging because of anti-monotonicity of dominance relation—a skyline tuple under space M may or may not be a skyline tuple in another space M', regardless of whether M' is a superspace or subspace of M [30]. We thus propose algorithms that first traverse the lattice in the full measure space, during which a frontier of constraints is formed for each measure subspace. Top-down (respectively, bottom-up) lattice traversal in a subspace commences from (respectively, stops at) the corresponding frontier instead of the root, which in effect prunes some top constraints.

Two Baseline Algorithms We introduce two baseline algorithms BaselineSeq (Algorithm 9) and BaselineIdx. They are not as naive as the brute-force Algorithm. Instead, they exploit Proposition 3 straightforwardly. Upon t's arrival, for each subspace M, they identify existing tuples t' dominating t. BaselineSeq sequentially compares t with every existing tuple. S is initialized to be $\mathcal{C}^t$ (Line 3). Whenever BaselineSeq encounters a t' that dominates t, it removes constraints in $\mathcal{C}^{t,t'}$ from S (Line 5). By Proposition 3, t is not in the contextual skylines for those constraints. After t is compared with all tuples, the constraints having t in their skylines remain in S. The same is independently repeated for every M. The pseudo code of BaselineIdx is similar to Algorithm and thus omitted. Instead of comparing t with all tuples, BaselineIdx directly finds tuples dominating t by a one-sided range query $\bigwedge_{m_i \in M}(m_i \geq t.m_i)$ using a *k*-d tree [7] on full measure space $\mathcal{M}$.

3.5 Algorithms

This section starts with algorithms BottomUp (Section 3.5.1) and TopDown (Section 3.5.2), which exploit the ideas of tuple reduction and constraint pruning. We then extend them to enable sharing of computation across measure subspaces (Section 3.5.3). Lastly, we analyze the complexity of the problem of contextual skylines maintenance (Section 3.5.4).

Based on the tuple-reduction idea (Proposition 1), a new tuple t should be included into a contextual skyline if and only if t is not dominated by any current skyline tuple in the context. Therefore, BottomUp and TopDown store and maintain skyline tuples with regard to each constraint-measure pair (C, M) and compare t with only the skyline tuples. For clarity of discussion, we differentiate between the contextual skyline ($\lambda_M(\sigma_C(R))$) and the space for storing it ($\mu_{C,M}$), since tuples stored in $\mu_{C,M}$ do not always equal $\lambda_M(\sigma_C(R))$, by our algorithm design.

The algorithms traverse, for each measure subspace M, the lattice of tuple-satisfied constraints $\mathcal{C}^t$ by certain order. When a constraint C is visited, the algorithms compare t with the skyline tuples stored in $\mu_{C,M}$. If t is dominated by t', then t does not belong to the contextual skyline of constraint-measure pair (C, M). Further, according to the constraint-pruning idea (Proposition 3), t does not belong to the contextual skyline of (C', M) for any C' satisfied by both t and t' (i.e., $C' \in \mathcal{C}^{t,t'}$). This property allows the algorithms to avoid comparisons with skyline tuples associated with such constraints.

The algorithms differ by how skyline tuples are stored in $\mu_{C,M}$. BottomUp stores a tuple for every constraint that qualifies it as a contextual skyline tuple, while TopDown only stores it for the topmost such constraints. In our ensuing discussion, we use invariants to formalize what must be stored in $\mu_{C,M}$. The algorithms also differ in the traversing order of the constraints in $\mathcal{C}^t$. BottomUp visits the constraints bottom-up, while TopDown makes the traversal top-down. Our discussion focuses on how the invariants are kept true under the algorithms' different traversal orders and execution logics. The algorithms present space-time tradeoffs. TopDown requires less space than BottomUp since it avoids storing duplicate skyline tuples as much as possible. The saving in space comes at the cost of execution efficiency, due to more complex operations in TopDown.

Pei et al. [30] proposed bottom-up and top-down algorithms to compute skycube. However, their algorithms are for the lattice of measure subspaces instead of constraints.

3.5.1 Algorithm BottomUp

BottomUp (Algorithm10) stores a tuple for every such constraint that qualifies it as a contextual skyline tuple. Formally, Invariant 1 is guaranteed to hold before and after the arrival of any tuple.

Invariant 1. *$\forall C \in \mathcal{C}_{\mathcal{D}}$ and $\forall M \subseteq \mathcal{M}$, $\mu_{C,M}$ stores all skyline tuples $\lambda_M(\sigma_C(R))$.* ■

Upon the arrival of a new tuple t, with regard to each measure subspace M, BottomUp traverses the constraints in $\mathcal{C}^t$ in a bottom-up, breadth-first manner. The traversal starts from Line 4 of Algorithm10, where the bottom of $\mathcal{C}^t$ is inserted into a queue Q. As long as Q is not empty, BottomUp visits the next constraint C from the head of Q and compares t with current skyline tuples in $\mu_{C,M}$ (Line 13). Various actions are taken, depending on comparison outcome. **1)** If t is dominated by any t', the comparison with remaining tuples in $\mu_{C,M}$ is skipped (Line 14). The tuple t is disqualified from not only C but also all constraints in $\mathcal{C}^{t,t'}$, by Proposition 3. Because BottomUp stores a tuple in all constraints that qualify it as a contextual skyline tuple, and because it traverses $\mathcal{C}^t$ bottom-up, the dominating tuple t' must be encountered at the bottom of $\mathcal{C}^{t,t'}$. BottomUp thus skips the comparisons with all tuples stored for C's ancestors (Line 14). **2)** If t dominates t', t' is removed from $\mu_{C,M}$ (Line 16). **3)** If t is not dominated by any tuple in $\mu_{C,M}$, it is inserted into $\mu_{C,M}$ (Line 16) and (C, M) corresponds to a contextual skyline for t (Line 15). Further,

```
Algorithm 10: BottomUp
    Input: $R(\mathcal{M}, \mathcal{D})$: existing tuples; $t$: the new tuple
    Output: $S^t$: the contextual skylines for $t$
1   $S^t \leftarrow \emptyset$;
2   foreach $M \subseteq \mathcal{M}$ do
3       foreach $C \in \mathcal{C}^t$ do $C.pruned \leftarrow$ false;
4       $Q \leftarrow \emptyset$; $Q.enqueue(\bot(\mathcal{C}^t))$;
5       while not $Q.empty()$ do
6           $C \leftarrow Q.dequeue()$;
7           $dominated \leftarrow$ false;
8           foreach $t' \in \mu_{C,M}$ do
9               if $t \prec_M t'$ then
10                  $dominated \leftarrow$ true;
11                  foreach $C' \in \mathcal{A}^t_C$ do
12                      $C'.pruned \leftarrow$ true; break;
13              else if $t' \prec_M t$ then $\mu_{C,M}.delete(t')$ ;
14          if not $dominated$ then
15              $S^t \leftarrow S^t \cup \{(C, M)\}$;
16              $\mu_{C,M}.insert(t)$;
17              foreach $C' \in \mathcal{P}^t_C$ do
18                  if (not $Q.contains(C')$) and (not $C'.pruned$) then $Q.enqueue(C')$;
19  $R \leftarrow R \cup \{t\}$;
20  return $S^t$;
```

each parent constraint of C that is not already pruned is inserted into Q, for continuation of bottom-up traversal (Line 18).

Below we prove that Invariant 1 is satisfied by BottomUp throughout its execution over all tuples.

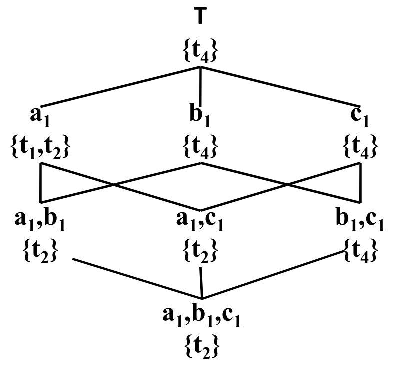
Proof of Invariant 1 We prove by induction on the size of table R. Invariant 1 is trivially true when R is empty. If the invariant is true before the arrival of t, i.e., $\mu_{C,M}$ stores all tuples in $\lambda_M(\sigma_C(R))$, we prove that it remains true after the arrival of t. The proof entails showing that both insertions into and deletions from $\mu_{C,M}$ are correct.

With regard to insertion, the only place where a tuple can be inserted into $\mu_{C,M}$ is Line 16 of BottomUp, which is reachable if and only if t is not dominated by any tuple in $\mu_{C,M}$ and C belongs to $\mathcal{C}^t$. This ensures that $\mu_{C,M}$ stores t if and only if $t \in \lambda_M(\sigma_C(R))$. Further, it enures that no previous tuple is inserted into $\mu_{C,M}$ upon the arrival of t, which is correct since such a tuple was not even in the skyline before.

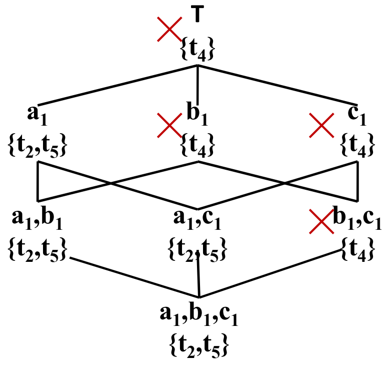
With regard to deletion, the only place where a previous skyline tuple t' can be deleted from $\mu_{C,M}$ is Line 16, which is reachable if and only if t dominates t' and C is satisfied by both tuples. This ensures that t' is removed from $\mu_{C,M}$ if and only if t' is not a skyline tuple anymore.

Therefore, regardless of whether insertion/deletion takes place upon t's arrival, $\mu_{C,M}$ stores all tuples in $\lambda_M(\sigma_C(R))$ afterwards.

Example 23. We use Figure3.3 to explain the execution of BottomUp on Table 3.4, for measure subspace M=$\{m_1$,$m_2\}$. Assume the tuples are inserted into the table in the order of t_1, t_2, t_3, t_4 and t_5. Figure3.3a shows the lattice $\mathcal{C}^{t_5}$ before the arrival of t_5. Beside each constraint C, the figure shows $\mu_{C,M}$. Upon the arrival of t_5, BottomUp starts the traversal of $\mathcal{C}^{t_5}$ from its bottom $\bot(\mathcal{C}^{t_5})$=$\langle a_1, b_1, c_1\rangle$. There is one skyline tuple stored in $\mu_{\bot(\mathcal{C}^{t_5}),M}$— t_2. In subspace M, t_5 is incomparable to t_2. Hence, t_5 is inserted into it. The traversal continues with the parents of $\bot(\mathcal{C}^{t_5})$. Among its three parents, $\langle a_1, b_1, *\rangle$ and $\langle a_1, *, c_1\rangle$ undergo the same insertion of t_5. However, the contextual skyline for $\langle *, b_1, c_1\rangle$ does not change, since t_5 is dominated by t_4 in M. All constraints in $\mathcal{C}^{t_4,t_5}$ (i.e., $\langle *, b_1, c_1\rangle$ and all its ancestors) are pruned from consideration by Property 3. The traversal continues at $\langle a_1, *, *\rangle$, for which t_1 is removed from the contextual skyline as it is dominated by t_5 in subspace M and t_5 is inserted into it. After that, the algorithm stops since there is no more unpruned constraints. The content of $\mu_{C,M}$ for constraints in $\mathcal{C}^{t_5}$ after the arrival of t_5 is shown in Figure3.3b. ■



(a) Before t_5



(b) After t_5

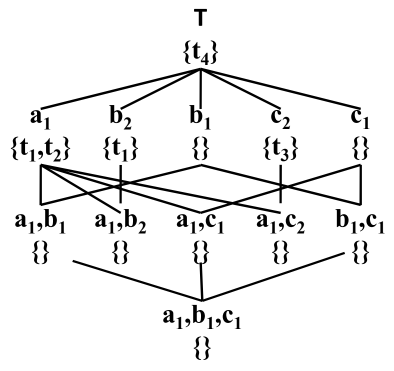
Figure 3.3: Execution of BottomUp in Measure Subspace $\{m_1, m_2\}$

3.5.2 Algorithm TopDown

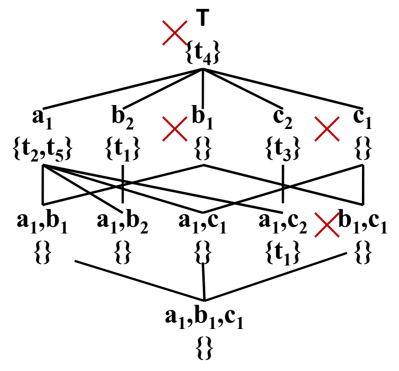
BottomUp stores t for every constraint-measure pair that qualifies t as a contextual skyline tuple. If t is stored in $\mu_{C,M}$, then t is also stored in $\mu_{C',M}$ for all $C'{\in}\mathcal{D}^t_C$, i.e., descendants of C pertinent to t. For this reason, BottomUp repeatedly compares a new tuple with a previous tuple multiple times. Such repetitive storage of tuples and comparisons increase both space complexity and time complexity. On the contrary, TopDown (Algorithm11) stores a tuple in $\mu_{C,M}$ only if C is a *maximal skyline constraint* of the tuple, defined as follows.

Definition 19 (Skyline Constraint)**.** Given $t \in R(\mathcal{D}; \mathcal{M})$ and $M \subseteq \mathcal{M}$, the *skyline constraints* of t in M, denoted $\mathcal{SC}^t_M$, are the constraints whose contextual skylines include t. Formally, $\mathcal{SC}^t_M = \{C | C \in \mathcal{C}^t, t \in \lambda_M(\sigma_C(R))\}$. Correspondingly, other constraints in $\mathcal{C}^t$ are *non-skyline constraints*. ■

Definition 20 (Maximal Skyline Constraints)**.** With regard to t and M, a skyline constraint is a *maximal skyline constraint* if it is not subsumed by any other skyline constraint of t. The set of t's maximal skyline constraints is denoted $\mathcal{MSC}^t_M$. In other words, it includes those skyline constraints for which no parents (and hence ancestors) are skyline constraints. Formally, $\mathcal{MSC}^t_M = \{C | C \in \mathcal{SC}^t_M$, and $\nexists C' \in \mathcal{A}_C$ s.t. $C' \in \mathcal{SC}^t_M\}$. ■



(a) Before t_5



(b) After t_5

Figure 3.4: Execution of TopDown in Measure Subspace $\{m_1, m_2\}$

Example 24. Figure 3.3b shows, in measure subspace $\{m_1, m_2\}$, t_5 belongs to the contextual skylines of 4 constraints, i.e., $\mathcal{SC}^{t_5}_{\{m_1,m_2\}} = \{\langle a_1, *, * \rangle, \langle a_1, b_1, * \rangle, \langle a_1, *, c_1\rangle, \langle a_1, b_1, c_1\rangle\}$. $\langle a_1, *, * \rangle$ is an ancestor of the remaining skyline constraints and thus it is the only maximal skyline constraint, i.e., $\mathcal{MSC}^{t_5}_{\{m_1,m_2\}} = \{\langle a_1, *, * \rangle\}$. ■

Formally, Invariant 2 is guaranteed by TopDown before and after the arrival of any tuple.

Invariant 2. *$\forall C \in \mathcal{C_D}$ and $\forall M \subseteq \mathcal{M}$, $\mu_{C,M}$ stores a tuple t if and only if $C \in \mathcal{MSC}^t_M$.* ■

Different from BottomUp, TopDown stores a tuple in its maximal skyline constraints $\mathcal{MSC}^t_M$ instead of all skyline constraints $\mathcal{SC}^t_M$. Due to this difference, TopDown traverses $\mathcal{C}^t$ in a top-down (instead of bottom-up) breadth-first manner. The traversal starts from Line 6 of Algorithm11, where the top element $\top$ is inserted into a queue Q. As long as Q

Algorithm 11: TopDown

```
Input: $R(\mathcal{M}, \mathcal{D})$: existing tuples; $t$: the new tuple
Output: $S^t$: the contextual skylines for $t$

1  $S^t \leftarrow \emptyset$;
2  foreach $M \subseteq \mathcal{M}$ do
3      foreach $C \in \mathcal{C}^t$ do
4          $C.pruned \leftarrow$ false;
5          $C.inAnces \leftarrow$ false;
6      $Q \leftarrow \emptyset$; $Q.enqueue(\top)$;
7      while not $Q.empty()$ do
8          $C \leftarrow Q.dequeue()$;
9          foreach $t' \in \mu_{C,M}$ do
10             if $t \prec_M t'$ then
11                 Dominated$(t', C)$;
12             else if $t' \prec_M t$ then
13                 Dominates$(t', C, M)$;
14         if not $C.pruned$ then
15             $S^t \leftarrow S^t \cup \{(C, M)\}$;
16             if not $C.inAnces$ then
17                 $\mu_{C,M}.insert(t)$;
18         EnqueueChindren$(C)$;
19 $R \leftarrow R \cup \{t\}$;
20 return $S^t$;

Procedure: Dominates $(t', C, M)$
1  $\mu_{C,M}.delete(t')$;
2  foreach $C' \in \mathcal{CH}_C^{t'} - \mathcal{C}^t$ do
3      $stored \leftarrow$ false;
4      foreach $C'' \in \mathcal{A}_{C'}^{t'} - \mathcal{C}^t$ do
5          if $t' \in \mu_{C'',M}$ then
6              $stored \leftarrow$ true;
7              break;
8      if not $stored$ then
9          $\mu_{C',M}.insert(t')$;

Procedure: Dominated $(t', C)$
1  $C.pruned \leftarrow$ true;
2  foreach $C' \in \mathcal{C}^{t,t'}$ do
3      $C'.pruned \leftarrow$ true;

Procedure: EnqueueChildren $(C)$
1  foreach $C' \in \mathcal{CH}_C^t$ do
2      if not $C.pruned$ then
3          $C'.inAnces \leftarrow$ true;
4      if not $Q.contains(C')$ then
5          $Q.enqueue(C')$;
```

is not empty, the algorithm visits the next constraint C from the head of Q and compares t with current skyline tuples in $\mu_{C,M}$ (Line 13). Various actions are taken, depending on the comparison result:

1) If t is dominated by t', t is disqualified from not only C but also all constraints in $\mathcal{C}^{t,t'}$, by Proposition 3. The pruning is done by calling Dominated in Line 11 which sets $C'.pruned$ to true for every pruned constraint C'. Since C is a maximal skyline constraint for t', the pruned constraints are all descendants of C in $\mathcal{C}^{t,t'}$. Note that TopDown cannot skip the comparisons with the remaining tuples stored in $\mu_{C,M}$. The reason is that there might be t'' in $\mu_{C,M}$ such that i) t'' also dominates t and ii) t'' and t share some dimension attribute values that are not shared by t', i.e., $\mathcal{C}^{t,t''}-\mathcal{C}^{t,t'}\neq\emptyset$. Since t'' is only stored in its maximal skyline constraints, skipping the comparison with t'' may incorrectly establish t as a contextual skyline tuple for those constraints in $\mathcal{C}^{t,t''} - \mathcal{C}^{t,t'}$.

2) If t dominates a current tuple t', t' is removed from $\mu_{C,M}$ by calling Dominates (Line 13). An extra work is to update the maximal skyline constraints of t' and store t' in descendants of C if necessary (Lines 2-9 of Dominates). If C has a child C' satisfied by t' but not t, C' is a skyline constraint of t'. Further, C' is a maximal skyline constraint of t', if no ancestor of C' is already a maximal skyline constraint of t'.

3) If t is not dominated by any tuple in $\mu_{C,M}$ and C was not pruned before when its ancestors were visited, (C, M) corresponds to a contextual skyline for t (Line 15). If t was not already stored in C's ancestors (indicated by $C.inAnces$), then C is a maximal skyline constraint and thus t is inserted into $\mu_{C,M}$ (Line 17).

Furthermore, subroutine EnqueueChildren is called for continuation of top-down traversal (Line 18). It inserts each child constraint C' of C into Q. If t is stored in $\mu_{C,M}$ or any of its ancestors, $C'.inAnces$ is set to true and t will not be stored again in $\mu(C', M)$ when the traversal reaches C'.

Below we prove that Invariant 2 is satisfied by TopDown throughout its execution over all tuples.

Proof of Invariant 2 We prove by induction on the size of table R. If the invariant is true before t's arrival, i.e., $\mu_{C,M}$ stores a tuple t if and only if $C\in\mathcal{MSC}^{t}_{M}$, we prove that it is kept true after the arrival of t. The proof constitutes showing that both insertions into and deletions from $\mu_{C,M}$ are correct.

With regard to insertion, there are two places where a tuple can be inserted. **1)** In Line 17 of TopDown, t is inserted into $\mu_{C,M}$. This line is reachable if and only if i) C is satisfied by t, ii) t is not dominated by any tuple stored at C or C's ancestors, and iii) t is not already stored at any of C's ancestors. This ensures that $\mu_{C,M}$ stores t if and only if C is a maximal skyline constraint of t, i.e., $C \in \mathcal{MSC}^t_M$. **2)** In Line 9 of Dominates, t' is inserted into $\mu_{C',M}$. This line is reachable if and only if i) t dominates t', ii) C, which is a parent of C', is satisfied by both tuples, iii) C' is satisfied by t' but not t, and iv) t' is not stored at any ancestor of C'. Since C was a maximal skyline constraint of t' before the arrival of t, C' must be a skyline constraint of t. Therefore these conditions ensure that $\mu_{C',M}$ stores t' if and only if C' becomes a maximal skyline constraint of t'.

With regard to deletion, the only place where a previous skyline tuple t' can be deleted from $\mu_{C,M}$ is Line 13 of Dominates, which is reachable if and only if t dominates t' and C is satisfied by both tuples. This ensures that t' is removed from $\mu_{C,M}$ if and only if C is not a maximal skyline constraint of t' anymore.

Therefore, regardless of whether any insertion or deletion takes place upon the arrival of t, afterwards $\mu_{C,M}$ stores all tuples for which C is a maximal skyline constraint.

Example 25. We use Figure3.4 to explain the execution of TopDown on Table 3.4 for M=$\{m_1$,$m_2\}$. Again, assume the tuples are inserted into the table in the order of t_1, t_2, t_3, t_4 and t_5. Figure3.4a shows $\mu_{C,M}$ beside each constraint C in $\mathcal{C}^{t_5}$ before the arrival of t_5. A tuple is only stored in its maximal skyline constraints. The figure also shows constraints outside of $\mathcal{C}^{t_5}$ where various tuples are also stored. The maximal skyline constraints for t_2 and t_4 are $\langle a_1, *, * \rangle$ and $\top$, respectively. The maximal skyline constraints for t_1 include $\langle a_1, *, * \rangle$ and $\langle *, b_2, * \rangle$. For t_3, the only maximal skyline constraint is $\langle *, *, c_2 \rangle$.

Upon the arrival of t_5, TopDown starts to traverse $\mathcal{C}^{t_5}$ from $\top$. Only t_4 is stored in $\mu_{\top,M}$. In M, t_5 is dominated by t_4, thus $\mu_{\top,M}$ does not change and t_5 does not belong to the contextual skylines of the constraints in $\mathcal{C}^{t_4,t_5}$—$\langle$ *, b_1, $c_1\rangle$, $\langle$ *, *, $c_1\rangle$, $\langle$ *, b_1, * $\rangle$ and $\top$. The traversal continues with the children of $\top$. Among its three children, $\langle$ *, b_1, * $\rangle$ and $\langle$ *, *, $c_1\rangle$ do not store any tuple, and t_1 and t_2 are stored at $\langle a_1$, *, * $\rangle$. They do not dominate t_5 in M. Since t_5 was not stored in any of its ancestors, $\langle a_1$, *, * $\rangle$ is a maximal skyline constraint of t_5. Hence, t_5 is inserted into it and will not be stored at its descendants $\langle a_1$, b_1, *$\rangle$, $\langle a_1$, *, $c_1\rangle$ and $\langle a_1$, b_1, $c_1\rangle$. Since t_5 dominates t_1, t_1 is deleted from $\langle a_1$, *, * $\rangle$. To update the maximal skyline constraints of t_1, TopDown considers the two children of $\langle a_1, *, *\rangle$—$\langle a_1, b_2, *\rangle$ and $\langle a_1$, *, $c_2\rangle$. $\langle a_1$, b_2, *$\rangle$ is not a new maximal skyline constraint, since t_1 is already stored at its ancestor $\langle$ *, b_2, *$\rangle$. $\langle a_1, *, c_2\rangle$ becomes a new maximal skyline constraint since it is not subsumed by any existing maximal skyline constraint of t_1. Thus t_1 is stored at $\langle a_1$, *, $c_2\rangle$. TopDown continues to the end and finds no tuple at any remaining constraint in $\mathcal{C}^{t_5}$. Figure3.4b depicts the content of $\mu_{C,M}$ for relevant constraints after t_5's arrival. ■

3.5.3 Sharing across Measure Subspaces

Given a new tuple, both TopDown and BottomUp compute its contextual skylines in each measure subspace separately, without sharing computation across different subspaces. As mentioned in Section 3.4, the challenge in such sharing lies in the anti-monotonicity of dominance relation—with regard to the same context of tuples, a skyline tuple in space M may or may not be a skyline tuple in another space M', regardless of whether M' is a superspace or subspace of M [30]. To share computation across different subspaces, we devise algorithms STopDown and SBottomUp. They discover the contextual skylines in all subspaces by leveraging initial comparisons in the full measure space $\mathcal{M}$. In this section, we first introduce STopDown and then briefly explain SBottomUp, which is based on similar principles.

With regard to two tuples t and t', the measure space $\mathcal{M}$ can be partitioned into three disjoint sets $\mathcal{M}^{>}$, $\mathcal{M}^{<}$ and $\mathcal{M}^{=}$ such that 1) $\forall m{\in}\mathcal{M}^{>}$, $t.m$ ¿ $t'.m$; 2) $\forall m{\in}\mathcal{M}^{<}$, $t.m$ ¡ $t'.m$; and 3) $\forall m{\in}\mathcal{M}^{=}$, $t.m = t'.m$. Then, t is dominated by t' in a subspace M if and only if M contains at least one attribute in $\mathcal{M}^{<}$ and no attribute in $\mathcal{M}^{>}$, as stated by Proposition 4.

Proposition 4. *In a measure subspace $M \subseteq \mathcal{M}$, $t \prec_M t'$ if and only if $M \cap \mathcal{M}^{<} \neq \emptyset$ and $M \cap \mathcal{M}^{>} = \emptyset$.* ■

The gist of STopDown (Algorithm12) is to compare a new tuple t with current tuples t' in full space $\mathcal{M}$ and, using Proposition 4, identify all subspaces M in which t' dominates t. It starts by finding the skyline constraints in $\mathcal{M}$ using STopDownRoot, which is similar to TopDown except Lines 13-16. While traversing a constraint C, t is compared with the tuples in $\mu_{C,\mathcal{M}}$ (Line 10 of STopDownRoot). By Proposition 4, all subspaces M where t' dominates t are identified. In each such M, constraints in $\mathcal{C}^{t,t'}$ are pruned (Lines 13-16)—indicated by setting values in a two-dimensional matrix $pruned$. After finishing STopDownRoot, with regard to each M, the constraints C satisfying $pruned[C][M]$ = **false** are the skyline constraints of t in M. STopDown then continues to traverse these skyline constraints in M by calling STopDownNode(M), for two purposes—one is to store t at its maximal skyline constraints (Line 11 of STopDownNode), the other is to remove tuples dominated by t and update their maximal skyline constraints (Line 10).

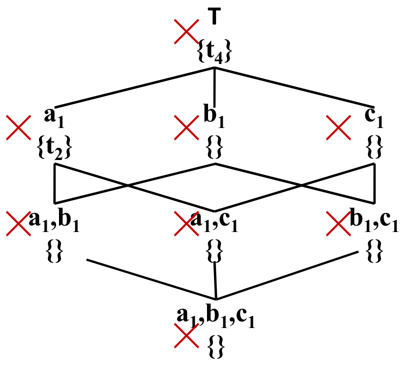
Example 26. We explain STopDown's execution on Table 3.4. In full space $\mathcal{M}$=$\{m_1, m_2\}$, STopDown and TopDown work the same. Hence, Figure3.4 shows $\mu_{C,\mathcal{M}}$ beside each C in $\mathcal{C}^{t_5}$ before and after t_5 arrives. Comparisons with tuples in $\mathcal{M}$ also help to prune constraints in subspaces. Consider $\top$ in Figure3.4a, where t_4 is stored. The new tuple t_5 is compared with t_4. The outcome is $\mathcal{M}^{>}$=$\emptyset$, $\mathcal{M}^{<}$=$\{m_1, m_2\}$ and $\mathcal{M}^{=}$=$\emptyset$, since t_5 is smaller than t_4 on both m_1 and m_2. By Proposition 4, t_5 is dominated by t_4 in subspaces $\{m_1\}$ and $\{m_2\}$.

Hence, all constraints in $\mathcal{C}^{t_4,t_5}$ (including $\langle$ *, b_1, $c_1\rangle$, $\langle$ *, b_1, * $\rangle$, $\langle$ *, *, $c_1\rangle$ and $\top$) are pruned in $\{m_1\}$ and $\{m_2\}$ simultaneously, by Lines 13-16 of STopDownRoot. As STopDownRoot proceeds, t_5 is also compared with t_1 and t_2. With regard to the comparison with t_1, since $\mathcal{M}^{<}$ = $\emptyset$, t_5 is not dominated by t_1 in any space. With regard to t_2, $\mathcal{M}^{>}$ = $\{m_2\}$, $\mathcal{M}^{<}$ = $\{m_1\}$ and $\mathcal{M}^{=}$ = $\emptyset$. Thus t_5 is dominated by t_2 in $\{m_1\}$. Hence, all the constraints in $\mathcal{C}^{t_2,t_5}$, which is identical to $\mathcal{C}^{t_5}$, are pruned in $\{m_1\}$.

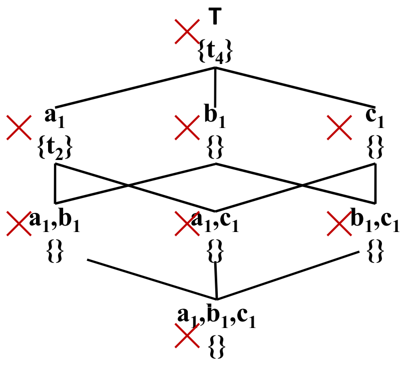
After the traversal in $\mathcal{M}$, STopDown continues with each measure subspace. In $\{m_1\}$, all constraints of $\mathcal{C}^{t_5}$ are pruned. Hence, t_5 has no skyline constraint and nothing further needs to be done. Figure3.5 depicts $\mu_{C,\{m_1\}}$ for all C in $\mathcal{C}^{t_5}$ before and after the arrival of t_5. For $\{m_2\}$, Figure3.6a depicts $\mu_{C,\{m_2\}}$ for all C in $\mathcal{C}^{t_5}$ before the arrival of t_5. Based on the analysis above, the skyline constraints of t_5 in $\{m_2\}$ include $\langle a_1$, *, * $\rangle$, $\langle a_1$, b_1, * $\rangle$, $\langle a_1$, *, $c_1\rangle$ and $\langle a_1$, b_1, $c_1\rangle$. Since non-skyline constraints are pruned, t_5 is not compared with the tuples stored at those constraints. Instead, t_5 is compared with t_1 stored at $\langle a_1$, *, * $\rangle$. Since they do not dominate each other in $\{m_2\}$, $\langle a_1$, *, * $\rangle$ is a maximal skyline constraint of t_5 and t_5 is stored at it together with t_1. The content of $\mu_{C,\{m_2\}}$ in $\mathcal{C}^{t_5}$ after encountering t_5 is in Figure3.6b. Note that TopDown would have compared t_5 with other tuples seven times, including comparisons with t_1, t_2 and t_5 in $\{m_1, m_2\}$, with t_2 and t_4 in $\{m_1\}$, and with t_1 and t_4 in $\{m_2\}$. In contrast, STopDown needs four comparisons, including the same three comparisons in $\{m_1, m_2\}$ and another comparison with t_1 in $\{m_2\}$. ■

Invariant 2 is also guaranteed by STopDown all the time. We omit the proof which is largely the same as the proof for TopDown. We note the essential difference between STopDown and TopDown is the skipping of non-skyline constraints in measure subspaces. Since the new tuple is dominated under these constraints, it does not and should not make any change to $\mu_{C,M}$ for any such constraint-measure pair.

BottomUp is extended to SBottomUp, similar to how STopDown extends TopDown. While in STopDown lattice traversal in a measure subspace commences from the topmost skyline constraints instead of the root of a lattice, lattice traversal in SBottomUp stops at them. Invariant 1 is also warranted by SBottomUp. Its proof is similar to that for BottomUp. Due to space limitations, we do not further discuss SBottomUp.

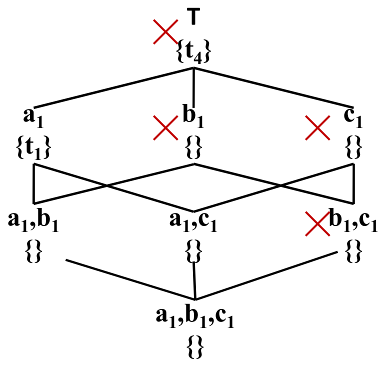


(a) Before Visiting $\mathcal{C}^{t_5}$ in $\{m_1\}$

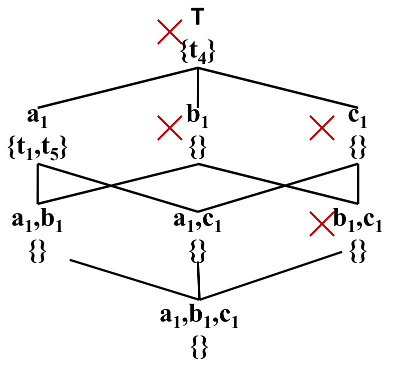


(b) After Visiting $\mathcal{C}^{t_5}$ in $\{m_1\}$

Figure 3.5: Execution of STopDown in Measure Subspace $\{m_1\}$ (No Comparison Required and No Change Made)



(a) Before Visiting $\mathcal{C}^{t_5}$ in $\{m_2\}$



(b) After Visiting $\mathcal{C}^{t_5}$ in $\{m_2\}$

Figure 3.6: Execution of STopDown in Measure Subspace $\{m_2\}$

3.5.4 Complexity Analysis

According to previous study, given a measure subspace, the computation of skyline is polynomial with respect to the number of tuples. Therefore, for a certain constrain-measure pair (C, M), the complexity of contextual skyline maintenance is polynomial with regard to the number of tuples that gets the opportunity to traverse C. The following theorem shows that the problem of contextual skylines computation is NP-hard.

Theorem 7. *The problem of contextual skylines computation is NP-hard.*

Proof of Theorem 7 We prove the complexity by reducing the problem of mining frequent itemsets [3] to computing the contextual skylines with regard to the constraints of $\mathcal{C}^t$ such that t is the new entry of R. According to [5], mining categorical frequent itemsets is NP-hard.

Consider $\mathcal{I}$ as a set of items and I is a subset of $\mathcal{I}$. A transaction is a set of itemsets. Given a transaction T, the support of I with respect to T is defined as the proportion of transactions in T which contains I. Given a threshold, if I's support satisfies it then I is called frequent. The problem of frequent itemset mining is to find all frequent itemsets.

To reduce the problem, we consider a database R' having dimension space $\mathcal{D}'$ and measure space $\mathcal{M}' = \{m\}$. Let assume $\mathcal{V}'$ = $\{v \in dom(d) | \forall d \in \mathcal{D}'\}$. Now for a itemset I = $\{i_1, i_2, \ldots, i_k\}$, we consider $\mathcal{V}' = \{v_1, v_2, \ldots, v_k\}$. An attribute value $v_i \in \mathcal{V}'$ represents an item $i_k \in I$, i.e., I interprets a constraint. We assume that for each tuple $m = 1$. Each transaction T is transformed to a tuple t_T according to this rule: if an item i_j belongs to a transaction T, then t_T includes v_j. This rule interprets that for any subset $I \subseteq \mathcal{I}$, the transaction T contains I if and only if t_T occupies the contextual skyline of I-interpreted constraint. This reduction is linear. Therefore, the problem of contextual skylines computation is NP-hard.

3.6 Instant Discovery of Situational Facts

In Section 3.1, we have exemplified news statements backed by data from domains including sports and social media. The first statement conveys the performance of Paul George in a particular match retrieved from the final score. The second statement demonstrates a photo's achieving significance over time. In both of the cases, each numeric value is the accumulation of the corresponding measure within a time frame. An instant dataset contains the cumulative measures with regard to each instant of time. Such instant dataset reveals the fluctuation of an id's significance within a time period—e.g., an athlete's regaining form in a match, or a youtube video's accumulating huge dislikes after the title being edited. We consider instant dataset in order to discover newsworthy facts synchronously with the live event, i.e, to stream live news. Clearly, our framework described in Section 3.3 is not compatible with an instant dataset. Nevertheless, the aforementioned fluctuations can be interpreted through update of tuples. An update can be represented by the combination of an old tuple's deletion and a new tuple's entrance.

Example 27. Consider the instant data R in Table 3.5, where $\mathcal{D}$ = $\{d_1, d_2, d_3\}$ and $\mathcal{M}$ = $\{m_1, m_2\}$. m_1 and m_2 are assumed as "as larger as better" and "as smaller as better", respectively. Suppose t_1 to t_6 as the existing tuples (ignore t_7 or t'_7 right now). For a particular id, $id.prev$ indicates the corresponding *previous tuple*. We assume that upon the arrival of a certain id, $id.prev$ *expires*. For instance, as t_3 is the $id.prev$ of t_4, the entrance of t_4 leads t_3 to be *expired*. Therefore, the skyline on context $\langle a_1$, *, * $\rangle$ under measure space $\mathcal{M}$ is $\{t_1, t_2\}$.

Suppose t_7 just arrives. This arrival causes the expiration of t_7's previous tuple t_2. Since t_7 dominates t_1, the skyline becomes $\{t_7\}$. Obviously, t_7 dominates t_2.

Now suppose t'_7 arrives instead of t_7 while t'_7 considers t_2 as its previous tuple as well. In this case, t'_7 is dominated by t_2. As t'_7 is dominated by t_1, t'_7 fails to occupy the skyline. Furthermore, a previous non-skyline tuple t_5 is promoted as a skyline tuple. t_2

was the only tuple that dominated t_5. Now t'_7 causes the expiration of t_2 and t'_7 can not dominate t_5. Therefore, the skyline is $\{t_1, t_5\}$. ■

In a scenario where each measure attribute is assumed as "as larger as better", a new tuple t always dominates its previous tuple $t.prev$. Therefore, the data model is not affected by the deletions and our proposed solution framework remains compatible with this case. However, the presence of a single measure attribute that is assumed as "as smaller as better" can lead t to be dominated by $t.prev$. This may allow a previously considered non-skyline tuple t' to get promoted as a skyline tuple. In this chapter, we name the phenomenon "$t.prev$ dominates t" as "relegation". To make our solution compatible with instant data framework, we extend our data model described in Section 3.3 as instant framework.

3.6.1 Instant Framework

Consider a relational schema $R(\mathcal{D};\mathcal{M})$, where the *dimension space* and the *measure space* are defined as before (Section 3.3). For each tuple, our framework records the corresponding previous tuple. Given $t, t' \in R$ such that t' is newer than t, t is recorded to be the previous tuple of t' if they share values on all dimensions and there does not exist any tuple t'' that shares values with t or t' on all dimensions. Formally, $(t'\ is\ newer\ than\ t) \wedge (\forall d \in \mathcal{D} : t.d = t'.d) \wedge (t''\ appears\ between\ t\ and\ t') \wedge (\nexists d \in \mathcal{D} : t.d = t''.d) \implies t'.prev = t$. For any $t \in R$, if $\exists t' \in R$ s.t. $t'.prev = t$, t is considered to be expired. Tuples other than the expired ones are considered as *alive*, i.e., $\{t \in R | \nexists t' \in R$ s.t. $t'.prev = t\}$. In Table 3.5, $R(\mathcal{D};\mathcal{M}) = \{t_1, t_2, t_3, t_4, t_5, t_6, t'_7\}$ (ignore t_7 for this discussion), $\mathcal{D} = \{d_1, d_2, d_3\}$, $\mathcal{M}$=$\{m_1, m_2\}$. We use this table as the running example. Note, for a tuple t, if there is no existing tuple that shares values with t on all dimensions, $t.prev$ is considered to be null. We assume a null tuple to be dominated by any non-null tuple regardless of measure subspace.

Definition 21 (Skyline)**.** An alive tuple t is a *skyline tuple* in subspace M if it is not dominated by any other alive tuples in R with respect to M. The set of all skyline tuples in R with respect to M is denoted by $\lambda_M(R)$, i.e., $\lambda_M(R)$=$\{t \in R | (\nexists t' \in R \text{ s.t. } t' \succ_M t) \wedge (\nexists t'' \in R \text{ s.t. } t''.prev = t)\}$. ■

Consider a case of "relegation" where the new tuple is t. For a constraint-measure pair (C, M), a straightforward approach to the contextual skyline discovery problem would check whether the contextual skyline contains $t.prev$. i) If $t.prev$ does not belong to the skyline, the approach would not go through any comparison regarding t. This is obvious as the contextual skyline does not include $t.prev$, there exists an alive tuple t' that dominates $t.prev$. Since $t.prev$ dominates t, t is dominated by t' (Proposition 1). ii) If $t.prev$ belongs to the skyline, the approach would compare t with every alive tuple satisfying C. Because at this point a previously decided non-skyline tuple may occupy the contextual skyline. The bottleneck of this approach is—the exhaustive comparison with *every alive tuple* with regard to *every constraint-measure pair* that qualifies $t.prev$ in contextual skyline. In order to ease these issues, we extend the ideas described in Section 3.4.

(1) Tuple reduction As we discussed in Section 3.4, for a certain constraint-measure pair, comparisons with skyline tuples are sufficient to decide whether an alive tuple in R belongs to the contextual skyline or not. In order to utilize this concept, we divide the phenomenon "t arrives $t.prev$ expires" into three time periods.

i) *Before the arrival of* t*:* Consider the data as R_B that contains the tuples in $R - \{t\}$, i.e., $t.prev$ is alive in this period.

ii) *Transition period:* In this period, the data R_T is described as $R_B - \{t.prev\}$, i.e., $t.prev$ is expired while t is not included yet. If $t.prev$ belonged to $\lambda_M(\sigma_C(R_B))$, then according to Definition 21, $\lambda_M(\sigma_C(R_T))$ contains tuples from $\lambda_M(\sigma_C(R_B))$ other than $t.prev$. Instead, $\lambda_M(\sigma_C(R_T))$ includes the *exclusively dominated tuples* of $t.prev$.

Definition 22 (Exclusively Dominated Tuples)**.** Given a constraint-measure pair (C, M) and two alive tuples $t, t' \in \sigma_C(R)$, if t dominates t' with respect to M and there exists no $t'' \in R - \{t\}$ such that t'' dominates t' with respect to M, t' is called *exclusively dominated tuple* of t. The set of all such exclusively dominated tuples of t in R with respect to (C, M) is denoted by $\mathcal{E}^{t}_{C,M}(R)$, i.e., $\mathcal{E}^{t}_{C,M}(R) = \{t' \in R | (t \succ_M t') \land (\nexists t'' \in R - \{t\}$ s.t. $t'' \succ_M t')\}$. Obviously, if $\mathcal{E}^{t}_{C,M}(R)$ is nonempty then t belongs to $\lambda_M(\sigma_C(R))$. Similarly, if $\lambda_M(\sigma_C(R))$ does not contain t, then $\mathcal{E}^{t}_{C,M}(R)$ is empty. ■

Example 28. Assuming the tuples are inserted into Table 3.5 in the order of t_1, t_2, t_3, t_4, t_5, t_6 and t'_7, if $M = \mathcal{M}$, and the constraint is $C = \langle a_1, b_1, * \rangle$, $\lambda_M(\sigma_C(R)) = \{t'_7\}$, $\mathcal{E}^{t'_7}_{C,M}(R) = \{t_6\}$, $\mathcal{E}^{t_2}_{C,M}(R - \{t'_7\}) = \{t_6\}$. Similarly, if $M = \mathcal{M}$, and the constraint is $C = \langle a_1, *, c_1 \rangle$, $\lambda_M(\sigma_C(R)) = \{t_5, t'_7\}$, $\mathcal{E}^{t'_7}_{C,M}(R) = \phi$ as well as $\mathcal{E}^{t_2}_{C,M}(R - \{t'_7\}) = \{t_5\}$. ■

In transition period, the content of the skyline is formalized through Equation 3.1.

$$\lambda_M(\sigma_C(R_T)) = \lambda_M(\sigma_C(R_B)) \cup \mathcal{E}^{t.prev}_{C,M}(R_B) - \{t.prev\} \tag{3.1}$$

Example 29. From Example 28, t_2 is the previous tuple of t'_7. Consider $C = \langle a_1, b_1, * \rangle$. For the phenomenon "$t'_7$ arrives t_2 expires", $\lambda_M(\sigma_C(R_B)) = \{\}$ and $\mathcal{E}^{t_2}_{C,M}(R_B) = \{t_6\}$. Therefore, in the transition period, the skyline $\lambda_M(\sigma_C(R_T))$ contains $\{t_6\}$ (Equation 3.1). ■

iii) *After the arrival of t:* Consider the data as R_A such that $R_A = R_T \cup \{t\}$. In any case of relegation, if $t.prev \notin \lambda_M(\sigma_C(R_B))$, then t does not belong to $\lambda_M(\sigma_C(R_A))$ as well. On the other hand, if $t.prev \in \lambda_M(\sigma_C(R_B))$, then $\lambda_M(\sigma_C(R_A))$ includes t if t is not dominated by any tuple in $\lambda_M(\sigma_C(R_T))$.

a) If t is dominated by any tuple in $\lambda_M(\sigma_C(R_T))$, then $\lambda_M(\sigma_C(R_A)) = \lambda_M(\sigma_C(R_T))$.

b) If t is not dominated by any tuple in $\lambda_M(\sigma_C(R_T))$, then t takes place in $\lambda_M(\sigma_C(R_A))$. Besides, tuples dominated by t do not belong to $\lambda_M(\sigma_C(R_A))$. According to Equation 3.1,

t can not dominate any tuple in $\lambda_M(\sigma_C(R_B))$. This is because none of these tuples is dominated by $t.prev$ and t is dominated by $t.prev$. If t dominates any tuple t' in $\mathcal{E}^{t.prev}_{C,M}(R_B)$, then t' becomes an exclusively dominated tuple of t, i.e., $t' \in \mathcal{E}^{t}_{C,M}(R_A)$ and $\mathcal{E}^{t}_{C,M}(R_A) \subseteq \mathcal{E}^{t.prev}_{C,M}(R_B)$. Clearly, $\lambda_M(\sigma_C(R_A))$ excludes $\mathcal{E}^{t}_{C,M}(R_A)$.

The aforementioned discussion is summarized in Proposition 5.

Proposition 5. *Given two tuples $t, t' \in R$, such that $t' = t.prev$ where t is the newest entry and $t \prec t'$,*

1. *$t \notin \lambda_M(\sigma_C(R))$ if and only if $\lambda_M(\sigma_C(R)) = \lambda_M(\sigma_C(R - \{t\})) \cup \mathcal{E}^{t'}_{C,M}(R - \{t\}) - \{t'\}$.*
2. *$t \in \lambda_M(\sigma_C(R))$ if and only if $\lambda_M(\sigma_C(R)) = \lambda_M(\sigma_C(R - \{t\})) \cup (\mathcal{E}^{t'}_{C,M}(R - \{t\}) - \mathcal{E}^{t}_{C,M}(R)) \cup \{t\} - \{t'\}$.*

■

Instead of comparing t with every tuple satisfying constraint as a straightforward approach, our algorithms exploit the aforementioned idea and conceptually maintain the exclusively dominated tuples for each tuple, i.e., compare t only with the skyline tuples from the transition period. This idea is justified through Proposition 5.

(2) Constraint pruning For constraints satisfied by t, we need to check whether the contextual skyline contains $t.prev$. To prune constraints from consideration, we note the following property: if $t.prev$ does not belong to the skyline with regard to constraint-measure pair (C, M), then (C', M) for any C' that subsumes C will not qualify t as a contextual skyline tuple. This concept is justified through Proposition 2 and 6.

Proposition 6. *Given two tuples $t, t' \in R$, such that $t = t'.prev$, if $t \notin \lambda_M(\sigma_C(R))$ then $t' \notin \lambda_M(\sigma_{C'}(R))$ where $C' \in \mathcal{A}_C \cup \{C\}$.* ■

3.6.2 Algorithms for Instant Framework

In this section, we propose algorithms PBottomUp, PTopDown, PSTopDown and PSBottom which are compatible with the extended framework. These algorithms exploit the idea described in Section 3.6.1.

1) Algorithm PBottomUp

Upon the arrival of t, if $t \succ_M t.prev$, then PBottomUp calls BottomUp. In case of $t \prec_M t.prev$, the actions are same as BottomUp except in Lines 10-11. While traversing, if any constraint C does not contain $t.prev$, then t is disqualified from not only C but also each of C's ancestors (Line 31) (Proposition 6). On the other hand, for each C belongs to $\mathcal{SC}_M^{t.prev}$, $t.prev$ is discarded from $\mu_{C,M}$ (Line 10) as well as tuples containing $\mathcal{E}_{C,M}^{t.prev}(R - \{t\})$ are inserted into $\mu_{C,M}$ (Line 11) (Proposition 5).

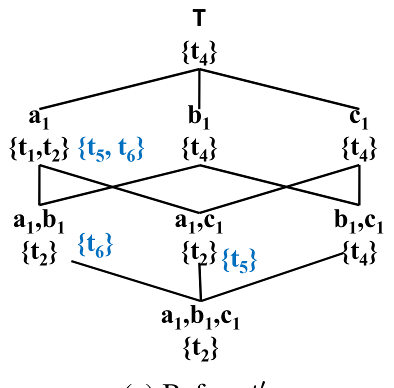
Furthermore, if $t.m = t.prev.m$ for all $m \in M$, the actions are same as PBottomUp except: 1) There is no action of comparison. 2) For each C belongs to $\mathcal{SC}_M^{t.prev}$, $t.prev$ is discarded from $\mu_{C,M}$ (Line 33) as well as t is stored into $\mu_{C,M}$ (Line 33). 1) and 2) are justified by Proposition 7. Invariant 1 (Section 3.5) is guaranteed by PBottomUp.

Proposition 7. *Given an alive tuple $t \in R$ such that $t.m = t.prev.m$ for all $m \in M$, a constraint $C \in \mathcal{C}^t$,*

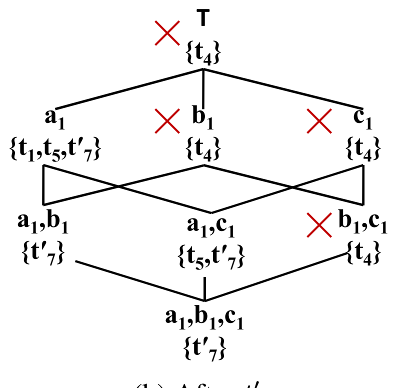
1. $t \notin \lambda_M(\sigma_C(R))$ *if and only if* $\lambda_M(\sigma_C(R)) = \lambda_M(\sigma_C(R - \{t\}))$.
2. $t \in \lambda_M(\sigma_C(R))$ *if and only if* $\lambda_M(\sigma_C(R)) = (\lambda_M(\sigma_C(R - \{t\})) - \{t.prev\}) \cup \{t\}$.

■

Example 30. We use Figure 3.7 to explain the execution of PBottomUp on Table 3.5, for measure subspace $\mathcal{M} = \{m_1, m_2\}$. Assume the tuples are inserted into the table in the order of t_1, t_2, t_3, t_4, t_5, t_6 and t_7'. Figure 3.7a depicts the content of $\mu_{C,M}$ for each C in $\mathcal{C}^{t_7'}$ before the arrival of t_7'. In addition, the content of $\mathcal{E}_{C,M}^{t}(R - \{t_2\})$ is depicted in blue color.



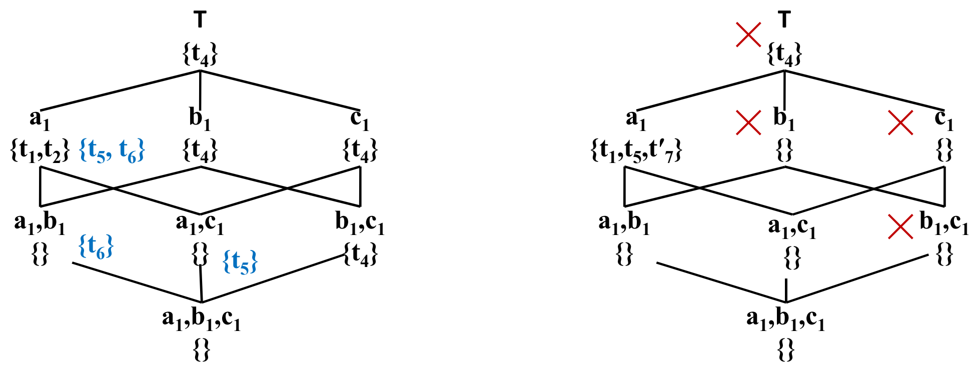
(a) Before t'_7



(b) After t'_7

Figure 3.7: Execution of PBottomUp in Measure Subspace $\{m_1, m_2\}$

When t'_7 arrives, it is compared with its previous tuple t_2. t_2 dominates t'_7. Now BottomUp starts the traversal of $\mathcal{C}^{t'_7}$ from its bottom $\bot(\mathcal{C}^{t'_7}) = \langle a_1, b_1, c_1\rangle$. $\langle a_1, b_1, c_1\rangle$ contains t_2 as a skyline tuple. As there is no other tuple in this context, t'_7 becomes skyline tuple here. The traversal proceeds along the parents of $\bot(\mathcal{C}^{t'_7})$. Among its three parents, $\langle a_1, b_1, *\rangle$ and $\langle a_1, *, c_1\rangle$ also contain t_2 as the only skyline tuple. In context of $\langle a_1, b_1, *\rangle$, t_6 is the only tuple exclusively dominated by t_2. After the expiration of t_2 and before the consideration of t'_7 , the intermittent skyline becomes $\{t_6\}$. However, upon the arrival of t'_7, t'_7 replaces t_6 from the skyline and t_6 remains as an exclusively dominated tuple of t'_7. In case of $\langle a_1, *, c_1\rangle$, the intermittent skyline becomes $\{t_5\}$ before the consideration of t'_7. Since t_5 is incomparable to t'_7, after considering t'_7, the final skyline becomes $\{t_5, t'_7\}$. The contextual skyline for $\langle *, b_1, c_1\rangle$ remains unaffected as it does not contain t_2. Along with $\langle *, b_1, c_1\rangle$, all the constraints in $\mathcal{A}_{\langle *, b_1, c_1\rangle}$ are pruned from further consideration (Proposition 6). The traversal continues at $\langle a_1, *, *\rangle$ since the skyline contains t_2. Now, after the expiration of t_2 and before the consideration of t'_7, the intermittent skyline becomes $\{t_1$, t_5, $t_6\}$ as $\{t_5$, $t_6\}$ is the set of exclusively dominated tuples of t_2. Since t'_7 is not dominated by any of these intermittent skyline tuple and it dominates t_6, the skyline becomes $\{t_1$, t_5, $t'_7\}$. The algorithm resumes at this point as there is no more unpruned constraints left. The content of $\mu_{C,M}$ for each C in $\mathcal{C}^{t'_7}$ after the arrival of t'_7 is depicted in Figure 3.7b. ■



(a) Before t'_7 (b) After t'_7

Figure 3.8: Execution of PTopDown in Measure Subspace $\{m_1, m_2\}$

2) Algorithm PTopDown

Upon the arrival of t, if $t \succ_M t.prev$, then PTopDown calls TopDown. If $t \prec_M t.prev$, the actions are same as PTopDown except:

1) *In Lines 3- 5*: For each constraint C in $\mathcal{MSC}_M^{t.prev}$, $t.prev$ is discarded from $\mu_{C,M}$ (Line3) as well as tuples containing $\mathcal{E}_{C,M}^{t.prev}(R-\{t\})$ are inserted into $\mu_{C,M}$ (Line 4) (Proposition 5). After that, the traversal starts from the maximal skyline constraints of $t.prev$ (Line 5) (Proposition 6).

2) *In Lines 7- 16*: i) If t is not stored into any of C's ancestors (indicated by $C.inAccess$), then t is compared with each tuple stored in $\mu_{C,M}$ to decide whether t occupies the contextual skyline (Line 7). ii) Otherwise, t is compared with only the tuples belong to $\mathcal{E}_{C,M}^{t.prev}(R-\{t\})$ in order to discard $\mathcal{E}_{C,M}^{t}(R)$ from the contextual skyline (Proposition 5). Note that, t is not compared with any tuples stored in $\mu_{C,M}$ (Line 13). Because at this point, PTopDown knows that $t \in \lambda_M(\sigma_C(R))$ and thereby, $\lambda_M(\sigma_C(R)) \supseteq \lambda_M(\sigma_C(R-t))$ (Property 2 of Proposition 5).

On the contrary, if $t.m$ = $t.prev.m$ for all $m \in M$, the actions are same as PTopDown except: 1) There is no comparison. 2) For each C in $\mathcal{MSC}_M^{t.prev}$, $t.prev$ is discarded from $\mu_{C,M}$ (Line 26) as well as t is stored into $\mu_{C,M}$ (Line 27). 1) and 2) are justified by Proposition 7. Invariant 2 (Section 3.5) is guaranteed by PTopDown.

Example 31. We use Figure 3.8 to explain the execution of PTopDown on Table 3.5, for measure subspace $\mathcal{M} = \{m_1, m_2\}$. Assume the tuples are inserted into the table in the order of t_1, t_2, t_3, t_4, t_5, t_6 and t'_7. Figure 3.8a depicts the content of $\mu_{C,M}$ for each C in $\mathcal{C}^{t'_7}$ before the arrival of t'_7. Besides, the tuples shown in blue color represents the corresponding intermittent contextual skyline.

Upon the arrival of t'_7, PTopDown compares it with its previous tuple t_2. Since t'_7 is dominated by t_2, the traversal starts from $\langle a_1, *, * \rangle$ as it is the only maximal skyline constraint of t_2. As t_2 is expired now, it is deleted from $\langle a_1, *, * \rangle$. At this point the intermittent skyline becomes $\{t_1, t_5, t_6\}$. Upon the consideration of t'_7, the skyline becomes $\{t_1, t_5, t'_7\}$. The traversal continues with the children of $\langle a_1, *, * \rangle$ which are unpruned. The descendents of $\langle a_1, *, * \rangle$ are $\langle a_1, b_1, * \rangle$, $\langle a_1, *, c_1 \rangle$ and $\langle a_1, b_1, c_1 \rangle$. Since $\langle a_1, b_1, * \rangle$ has t_6 as exclusively dominated tuple of t_2 and t'_7 dominates t_6; $\mu_{\langle a_1, b_1, * \rangle, M}$ remains empty. In case of $\langle a_1, *, c_1 \rangle$, $\langle a_1, *, c_1 \rangle$ has t_5 as exclusively dominated tuple of t_2 and t_5 is not dominated by t'_7. At this point, $\langle a_1, *, c_1 \rangle$ is subsumed by t_5's maximal skyline constraint $\langle a_1, *, * \rangle$. Therefore, t_5 is not stored in $\mu_{\langle a_1, *, c_1 \rangle, M}$. Since there exists no tuple as exclusively dominated tuple of t_2 in $\langle a_1, b_1, c_1 \rangle$, no further comparison is required. PTopDown stops at this point. Note that, after traversing $\langle a_1, *, * \rangle$, it becomes obvious that t belongs to $\lambda_M(\sigma_C\ (R))$ for all $C \in \mathcal{D}_{\langle a_1, *, * \rangle}$. Therefore, t'_7 does not need to be compared with any tuple belongs to $\mu_{C,M}$ (Property 2 of Proposition 5). Figure 3.8b depicts the content of $\mu_{C,M}$ for each C in $\mathcal{C}^{t'_7}$ after t'_7's arrival. ■

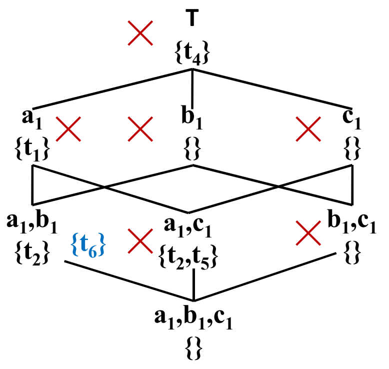
3) Sharing across Measure Subspaces

In order to share computation across subspaces, we design PSBottomUp and PSTopDown. These two algorithms compute the contextual skylines with regard to the full space at first. The computations correspond to the remaining subspaces are leveraged though adapting the computations in the full space. In this section, we first introduce PSTopDown and then discuss PSBottomUp briefly.

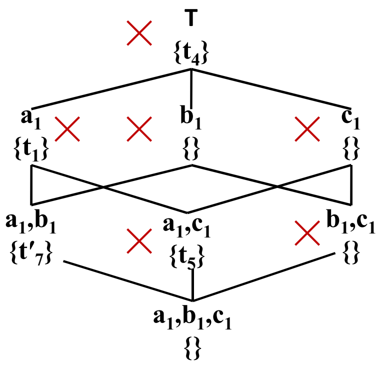
PSTopDown incorporates the idea behind PTopDown and STopDown. The gist of PSTopDown (Algorithm15) is to compare a new tuple t with the corresponding previous tuple $t.prev$ in full space $\mathcal{M}$ and, using Proposition 4, identify all subspaces M in which either t dominates $t.prev$ or t is dominated by $t.prev$. At first, the algorithm computes the contextual skyline tuples in $\mathcal{M}$ using PSTopDownRoot, which is similar to STopDown-Root except in (Lines 4-7). PSTopDownRoot starts by discarding $t.prev$ from its maximal skyline constraints in $\mathcal{M}$ (Line 5) as well as adding $\mathcal{E}_{C,\mathcal{M}}^{t.prev}(R - \{t\})$ into $\mu_{C,\mathcal{M}}$ (Line 7) (Proposition 5). As described in Section 3.5, while traversing a constraint C, t is compared with the tuples in $\mu_{C,\mathcal{M}}$ (Line 10 of STopDownRoot). STopDownRoot identifies the subspaces M where t is dominated by any t' in $\mu_{C,\mathcal{M}}$ and thereby, prunes each such constraint-measure pair (C, M) (Lines 13-16 of STopDownRoot) (Proposition 4). Other than $\mathcal{M}$, with regard to each M, PSTopDown calls PSTopDownNode(M) to traverse the skyline constraints of $t.prev$. Now PSTopDownNode stores t in its maximal skyline constraints (Line 21 of PSTopDownNode). Note that, PSTopDownNode traverses a skyline constraints C of $t.prev$ in spite of C is pruned. This is because, during this traversal the tuples belong to $\mathcal{E}_{C,M}^{t.prev}(R - \{t\})$ are inserted into $\mu_{C,M}$ (Line 12). Thus Invariant 2 is also guaranteed by PSTopDown all the time.

Example 32. We explain PSTopdown's execution on Table 3.5 considering t_7' as the newest tuple. In full space $\mathcal{M}$ = $\{m_1, m_2\}$, PTopDown and PSTopDown work the same which is explained through Figure 3.8. Consider $\langle a_1, *, *\rangle$ from Example 31, where t_1 and t_5 are stored intermittently. While comparing t_7' with t_1 and t_5, we get that t_7' is dominated by both t_1 and t_5 on $\{m_2\}$. Therefore, all constraints in $\mathcal{C}^{t_1,t_7'} \cup \mathcal{C}^{t_5,t_7'}$ are pruned in $\{m_2\}$ simultaneously.

For $\{m_2\}$, PSTopDown starts the traversal from the maximal skyline constraints of $t.prev$: $\{\langle a_1, b_1, *\rangle, \langle a_1, *, c_1\rangle\}$. Before the consideration of t_7', $\langle a_1, b_1 *\rangle$ consider $\{t_6\}$ as intermittent skyline. After considering t_7', the skyline becomes $\{t_7'\}$. In case of $\langle a_1, *, c_1\rangle$,



(a) Before Visiting $\mathcal{C}^{t'_7}$ in $\{m_2\}$



(b) After Visiting $\mathcal{C}^{t'_7}$ in $\{m_2\}$

Figure 3.9: Execution of PSTopDown in Measure Subspace $\{m_2\}$

the intermittent skyline is $\{t_5\}$. Since $\langle a_1, *, c_1\rangle$ is pruned, this intermittent skyline is finalized. The traversal proceeds through $\langle a_1, b_1, c_1\rangle$, where the intermittent skyline is ϕ. Therefore, there is no more comparison and PSTopDown resumes at this point. Figure 3.9 depicts $\mu_{C,\{m_2\}}$ for all C in $\mathcal{C}^{t'_7}$ before and after the arrival of t'_7. ■

PBottomUp is extended to PSBottomUp, similar to how PSTopDown extends PTopDown. Invariant 1 is also warranted by PSBottomUp.

3.7 Case Study

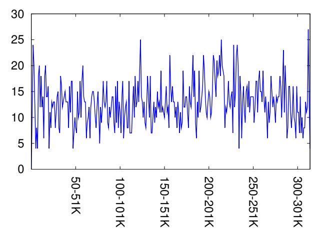
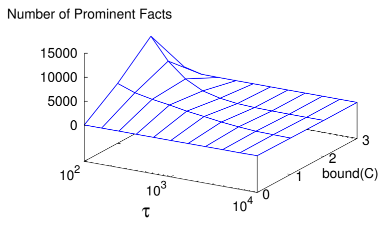
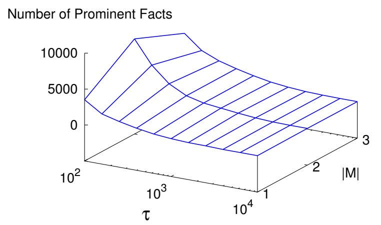


Figure 3.10: Number of Prominent Facts for Each 1K Tuples on the NBA box score dataset, τ=10^3



(a) By Number of Bound Dimension Attributes



(b) By Dimensionality of Measure Subspaces

Figure 3.11: Distribution of Prominent Facts on the NBA box score dataset, Varying τ

A tuple may be in the contextual skylines of many constraint-measure pairs. For instance, t_7 in Example 17 belongs to 196 contextual skylines (of course partly because the table is tiny and most contexts contain only t_7). Reporting all such facts overwhelms users and makes important facts harder to spot. It is crucial to report truly *prominent* facts, which should be rare. We measure the *prominence* of a fact (i.e., a constraint-measure pair (C, M)) by $\frac{|\sigma_C(R)|}{|\lambda_M(\sigma_C(R))|}$, the cardinality ratio of all tuples to skyline tuples in the context. Consider two pairs in Example 17: (C_1:month = *Feb*, M_1:{points, assists, rebounds}) and (C_2:team=*Celtics*$\wedge$opp_team = *Nets*, M_2:{assists, rebounds}). The context of C_1 contains 5 tuples, among which t_2 and t_7 are in the skyline in M_1. Hence, the prominence of (C_1, M_1) is $5/2$. Similarly the prominence of (C_2, M_2) is $3/2$. Therefore, (C_1, M_1) is more prominent, because larger ratios indicate rarer events.

For a newly arrived tuple t, we rank all situational facts S^t pertinent to t in descending order of their prominence. A fact is *prominent* if its prominence value is the highest among S^t and is not below a given threshold τ. (There can be multiple prominent facts pertinent to the arrival of t, due to ties in their prominence values.) Consider t_7 in Example 17. From the 196 facts in S^{t_7}, the highest prominence value is 3. If $\tau \leq 3$, those facts in S^{t_7} attaining value 3 are the prominent facts pertinent to t_7. Among many such facts, examples are (player = *Wesley*, {rebounds}) and (month = *Feb.*$\wedge$team = *Celtics*,{points}). Note that, based on the definition of the prominence measure and the threshold τ, a context must have at least τ tuples in order to contribute a prominent fact.

We studied the prominence of situational facts from the NBA box score dataset, under the parameter setting d = 5, m = 7, $\hat{d}$ = 3, $\hat{m}$ = 3 and τ = 500. In other words, each prominent fact on a new tuple t is about a contextual skyline that contains t and at most 0.2% of the tuples in the context. Below we show some of the discovered prominent facts. They do not necessarily stand in the real world, since our dataset does not include the complete NBA records from all seasons.

- Lamar Odom had 30 points, 19 rebounds and 11 assists on March 6, 2004. No one before had a better or equal performance in NBA history.
- Allen Iverson had 38 points and 16 assists on April 14, 2004 to become the first player with a 38/16 (points/assists) game in the 2004-2005 season.
- Damon Stoudamire scored 54 points on January 14, 2005. It is the highest score in history made by any Trail Blazers.

Figures 3.10 and 3.11 help us further understand the prominent facts from this experiment at the macro-level. Figure 3.10 shows the number of prominent facts for each 1000 tuples, given threshold $\tau = 10^3$. For instance, there are 11 prominent facts in total from the $100{,}000^{\text{th}}$ tuple to the $101{,}000^{\text{th}}$ tuple. We observed that the values in Figure 3.10 mostly oscillate between 5 and 25. Consider the number of tuples and the huge number of constraint-measure pairs, these prominent facts are truly selective. One might expect a downward trend in Figure 3.10. It did not occur due to the constant formulation of new contexts. Each year, a new NBA regular season commences and some new players start to play. Such new values of dimension attributes season and player, coupled with combinations of other dimension attributes, form new contexts. Once a context is populated with enough tuples (at least τ), a newly arrived tuple belonging to the context may trigger a prominent fact.

Figure 3.11a shows the distribution of prominent facts by the number of bound dimension attributes in constraint for varying τ in $[10^2, 10^4]$. Figure 3.11b shows the distribution by the dimensionality of measure subspace. We observed fewer prominent facts with 0 and 3 bound attributes (out of $d = 5$ dimension attributes) than those with 1 and 2 bound attributes, and fewer prominent facts in measure subspaces with 1 and 3 attributes than those with 2 attributes. The reasons are: **1)** With regard to dimension attributes, if there are no bound attributes in the constraint, the context includes the whole table. Naturally it is more challenging to establish a prominent fact for the whole table. If the constraint has more bound attributes, the corresponding context becomes more specific and contains fewer tuples, which may not be enough to contribute a prominent fact (recall that having one prominent fact entails a context size of no less than τ). Therefore, there are fewer prominent facts with 3 bound attributes. **2)** With regard to measure attributes, on a single measure, a tuple must have the highest value in order to top other tuples, which does not often happen. There are thus fewer prominent facts in single-attribute subspaces. In a subspace with 3 attributes, there are also fewer prominent facts, because the contextual skyline contains more tuples, leading to a smaller prominence value that may not beat the threshold τ.

We studied the prominence of situational facts from the NBA play by play dataset as well, under the parameter setting $d = 5$, $m = 7$, $\hat{d} = 3$, $\hat{m} = 3$ and $\tau = 5,000$. Below we present some of the discovered prominent facts. As our collected NBA play by play dataset does not include the complete NBA records, these discovered facts may not stand in the real world. Note that, in order to discover prominent facts, we don not need to wait for the box score this time. The algorithms regarding the instant framework discovers prominent facts synchronously with an ongoing match and thus we can provide live commentary.

- Shawn Marion has 30 points, 16 rebounds and 2 assists. No one before had a better or equal performance in NBA history.
- Russell Westbrook has 21 points and 15 assists to become the first player with a 21/15 (points/assists) game in this 2015-2016 season in the Oklahoma City Thunder's match against Los Angeles Clippers. ...Kevin Durant has scored 30 points. It is the highest score in this season made by any Los Angeles Clippers.
- Kendall Gill has scored 21 points and 0 fouls. No one had a better performance while having no foul in the month of January in the Orlando Magic's match with New Jersey Nets. ...Dennis Scott has scored 22 points to become the first player with a 22 (points) game in in the month of January in the Orlando Magic's match with New Jersey Nets.

Dennis Scott has scored 1 foul. ...Yet Kendall Gill has scored 0 foul and he is the highest pointer while having no foul in the month of January in the Orlando Magic's match with New Jersey Nets.

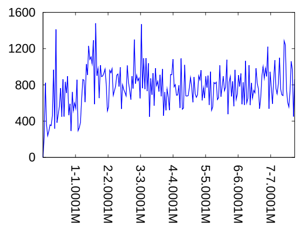
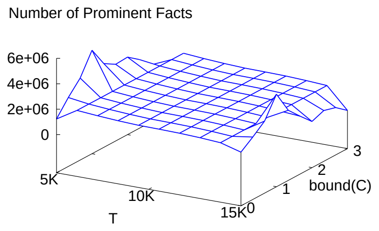
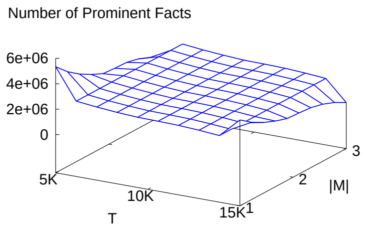


Figure 3.12: Number of Prominent Facts for Each 1K Tuples on the NBA play by play dataset, τ=$15,000$



(a) By Number of Bound Dimension Attributes



(b) By Dimensionality of Measure Subspaces

Figure 3.13: Distribution of Prominent Facts on the NBA play by play dataset, Varying τ

Figures 3.12 and 3.13 further demonstrate the prominent facts discovered from the NBA play by play dataset at the macro-level. As the number of tuples is comparatively higher than the box score dataset, we consider higher thresholds in this experiment. Figure 3.12 shows the number of prominent facts for each $1,000$ tuples, given threshold τ = $15,000$. Figures 3.12 reveals that the values in Figure 3.12 mostly oscillate between 300 and 1500. It is noticeable that the number of prominent facts is significantly greater in comparison to Figures 3.12. This is because the play by play dataset contains cumulative measures and intuitively, any significant measure is distributed in comparatively higher number of tuples. In other words, players having outstanding performances appear in comparatively higher number of tuples. Therefore, a tuple is more likely belongs to a contextual skyline, i.e., appear as a prominent fact. Nevertheless, with regar to the number of tuples in play by play dataset, these prominent facts are quite selective. Similar to Figure 3.12, we observe a downward trend is Figure 3.12 and the reason behind is same as before.

Figure 3.13a shows the distribution of prominent facts by the number of bound dimension attributes in constraint for varying τ in $[5 \times 10^3, 15 \times 10^3]$. We observe highest prominent facts with 1 bound attributes (out of $d = 5$ dimension attributes) which concurs with Figure 3.11a. Besides, Figure 3.13b demonstrates the distribution by the dimensionality of measure subspace. Figure 3.13b reveals that measure subspaces with 1 attibute have the highest prominent facts while Figure 3.11b shows fewer prominent facts with 1 attribute. Since a player is more likely to score for only one measure and higher measures are comparatively highly distribute, again the cumulative measures answer the reason behind this phenomenon.

3.8 Experiments

The algorithms were implemented in Java. The experiments were conducted on a computer with 2.0 GHz Quad Core 2 Duo Xeon CPU running Ubontu 8.10. The limit on the heap size of Java Virtual Machine (JVM) was set to 16 GB.

3.8.1 Experiment Setup

Datasets We used three real datasets of which NBA boxscore and weather datasets have similar trends. We mainly discuss the results on the two NBA datasets.

NBA Box Score Dataset We collected $317{,}371$ tuples of NBA box scores from 1991-2004 regular seasons. We considered 8 dimension attributes: player, position, college, state, season, month, team, and opp_team. College denotes from where a player graduated, if applicable. State records the player's state of birth. For measure attributes, 7 performance statistics were considered: points, rebounds, assists, blocks, steals, fouls and turnovers. Smaller values are preferred on turnovers and fouls, while larger values are preferred on all other attributes.

NBA Play by Play Dataset We crawled NBA play by play scores from 1996-2016 regular seasons. We considered 5 dimension attributes: player, season, month, team, and opp_team. For measure attributes, we collected the same 7 performance statistics that were considered in the NBA box score dataset. We translated each play by play score corresponding to these attributes as a tuple compatible with our instant framework. For translation, we made these following two rules:

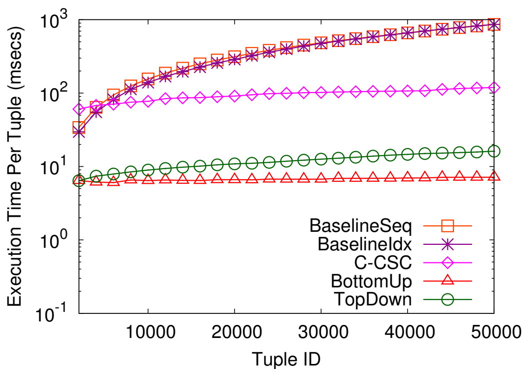
1. Rule 1: If two consecutive tuples t_1 and t_2 share same values in all of the 5 dimensions, then t_1 is considered as t_2's previous tuple. Formally, $\forall d \in \mathcal{D} t_1.d = t_2.d \Rightarrow t_1 = t_2.prev$.
2. Rule 2: Consider a play by play score demonstrated as a score of v in a certain measure $m \in \mathcal{M}$. Then the corresponding tuple t would be interpreted such that $t.m = t.prev.m + v$.

Thus we collected $9{,}154{,}078$ tuples in total.

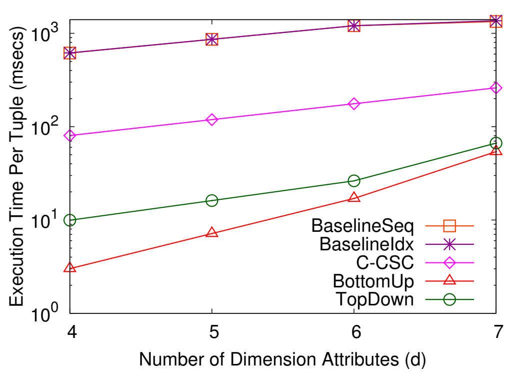
Weather Dataset [1] It has more than 7.8 million daily weather forecast records collected from $5{,}365$ locations in six countries and regions of UK from Dec. 2011 to Nov. 2012. Each record has 7 dimension attributes: location, country, month, time step, wind direction [day], wind direction [night] and visibility range and 7 measure attributes: wind speed [day], wind speed [night], temperature [day], temperature [night], humidity [day], humidity [night] and wind gust. We assumed larger values dominate smaller values on all attributes.

Methods Compared For NBA box score and weather datasets, we investigated the performances of 7 algorithms—the baseline algorithms BaselineSeq and BaselineIdx from Section 3.4, C-CSC which is the CSC adaptation described in Section 3.2, and the algorithms BottomUp, TopDown, SBottomUp and STopDown from Section 3.5. For NBA play by play dataset, we evaluate the performances of 3 algorithms—the baseline algorithm

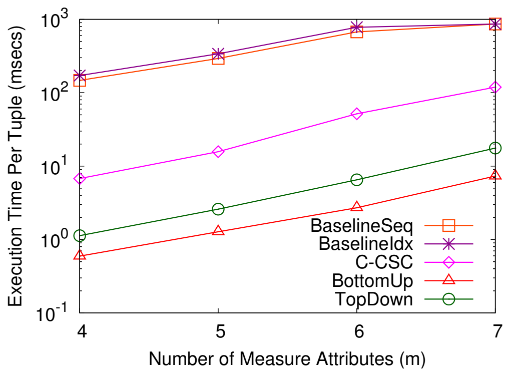
[1] http://data.gov.uk/metoffice-data-archive



(a) Varying n, d=5, m=7



(b) Varying d, n=50,000, m=7

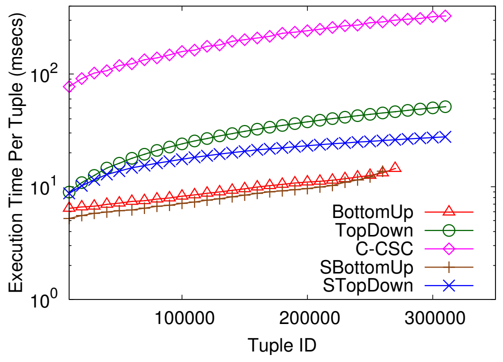


(c) Varying m, n=50,000, d=5

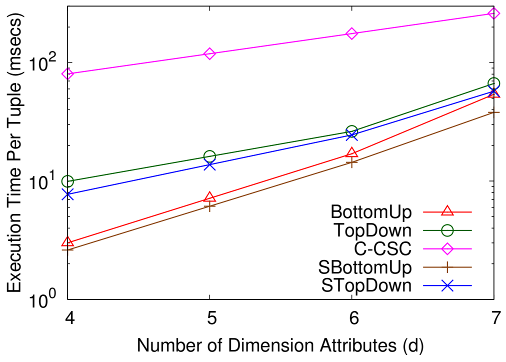
Figure 3.14: Execution Time of BaselineSeq, BaselineIdx, C-CSC, BottomUp and TopDown on the NBA Box score Dataset

Baseline from Section 3.4, and the algorithms PBottomUp, PTopDown, and PSTopDown from Section 3.6. We compared these algorithms on both execution time and memory consumption.

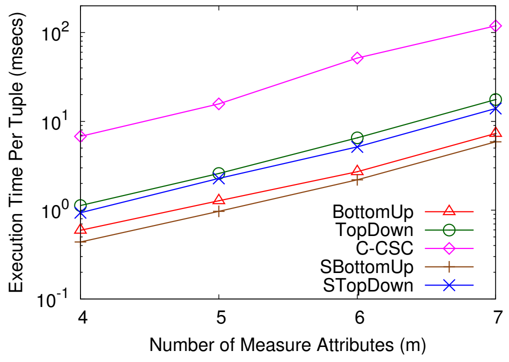
Parameters We ran our experiments under combinations of five parameters, which are number of dimension attributes (d), number of measure attributes (m), number of tuples (n), maximum number of bound dimension attributes ($\hat{d}$) and maximum number of measure attributes allowed in measure subspaces ($\hat{m}$). In Table 3.6 (3.7), we list the dimension (measure) spaces considered for different values of d (m), which are subsets of the aforementioned dimension (measure) attributes in the datasets.



(a) Varying n, d=5, m=7



(b) Varying d, n=50,000, m=7



(c) Varying m, n=50,000, d=5

Figure 3.15: Execution Time of C-CSC, BottomUp, TopDown, SBottomUp, STopDown on NBA Box score Dataset

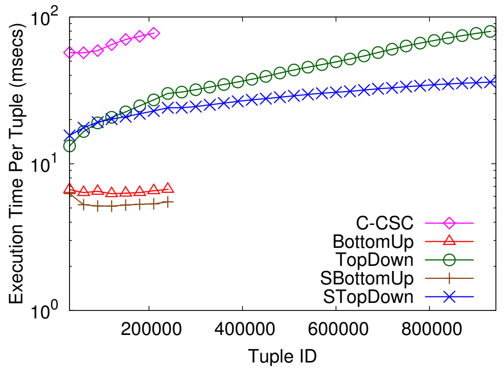


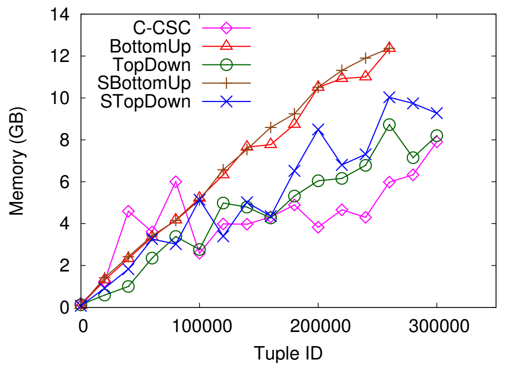
Figure 3.16: Execution Time on the Weather Dataset, Varying n, d=5, m=7

In particular dimension/measure spaces (corresponding to d/m values), experiments were done for varying $\hat{d}$ and $\hat{m}$ values. A constraint with more bound dimension attributes represents a more specific context. Similarly, a measure subspace with more measure attributes is more specific. Considering all possible constraint-measure pairs may thus produce many over-specific and uninteresting facts. The parameters $\hat{d}$ and $\hat{m}$ are for avoiding trivial facts. For instance, if d=5, m=4, $\hat{d}$=2 and $\hat{m}$=3, we consider all constraints with at most 2 (out of 5) bound dimension attributes and all measure subspaces with at most 3 (out of 4) measure attributes. In all experiments in this section, we set $\hat{d} = 4$ and $\hat{m} = m$. That means a constraint is allowed to have up to 4 bound attributes and a measure subspace can be any subspace of the whole space $\mathcal{M}$ including $\mathcal{M}$ itself. In Section 3.7, we further study how prominence of facts varies by $\hat{d}$ and $\hat{m}$ values.

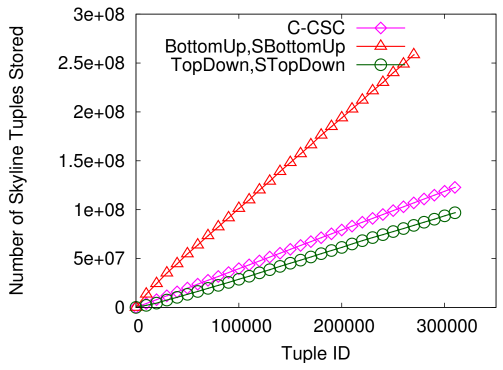
3.8.2 Experiments on NBA Box score and Weather Datasets

Results of Memory-Based Implementation Figure 3.14 compares the per-tuple execution times (by milliseconds, in logarithmic scale) of BaselineSeq, BaselineIdx, C-CSC, BottomUp and TopDown on the NBA box score dataset. Figure 3.14a shows how the per-tuple execution times increase as the algorithms process tuples sequentially by their timestamps. The values of d and m are d = 5 and m = 7. Figure 3.14b shows the times under varying d, given n = 50,000 and m = 7. Figure 3.14c is for varying m, n = 50,000 and d = 5. The figures demonstrate that BottomUp and TopDown outperformed the baselines by orders of magnitude and C-CSC by one order of magnitude. Furthermore, Figure 3.14b and Figure 3.14c show that the execution time of all these algorithms increased exponentially by both d and m, which is not surprising since the space of possible constraint-measure pairs grows exponentially by dimensionality.

Figure 3.15 uses the same configurations in Figure 3.14 in order to compare C-CSC, BottomUp, TopDown, SBottomUp and STopDown. We make the following observations on the results. First, C-CSC was outperformed by one order of magnitude. The per-tuple execution times of all algorithms exhibited moderate growth with respect to n and superlinear growth with respect to d and m, matching the observations from Figure 3.14.

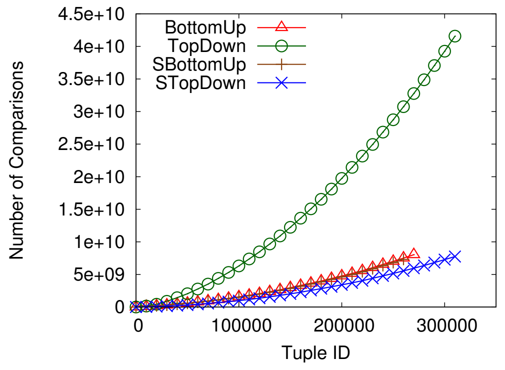


(a) Size of Consumed Memory

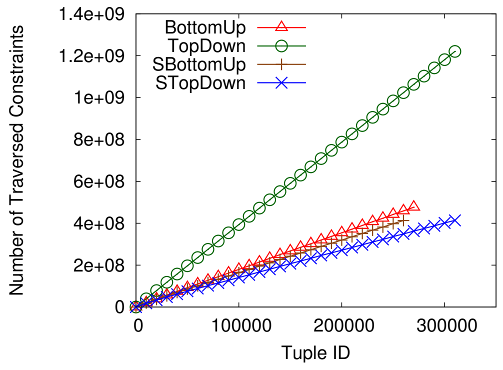


(b) Num of Skyline Tuples Stored

Figure 3.17: Memory Consumption by C-CSC, BottomUp, TopDown, SBottomUp, STopDown on the NBA Box score Dataset, Varying n, d=5, m=7



(a) Number of Comparisons



(b) Num of Traversed Constraints

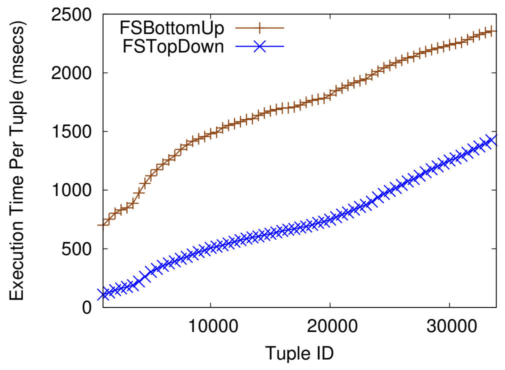
Figure 3.18: Work Done by BottomUp, TopDown, SBottomUp and STopDown on the NBA Box score Dataset, Varying n, d=5, m=7

Second, as depicted in Figure 3.15a, the bottom-up algorithms exhausted available JVM heap and were terminated due to memory overflow before all tuples were consumed. On the contrary, the top-down algorithms finished all tuples. This difference was more clear on the larger weather dataset (Figure 3.16), on which the bottom-up algorithms caused memory overflow shortly after 0.2 million tuples were encountered, while the top-down algorithms were still running normally after 0.9 million tuples. As the difference was already clear after 0.9 million tuples, we terminated the executions of top-down algorithms at that point. The difference in the sizes of consumed memory by these two categories of algorithms is shown in Figure 3.17a. The difference in memory consumption is due to that TopDown/STopDown only store a skyline tuple at its maximal skyline constraints, while BottomUp/SBottomUp store it at all skyline constraints. This observation is verified by Figure 3.17b, which shows how the number of stored skyline tuples increases by n. We see that BottomUp/SBottomUp stored several times more tuples than TopDown/STopDown. Note that TopDown and STopDown use the same skyline tuple materialization scheme. Correspondingly BottomUp and SBottomUp store tuples in the same way.

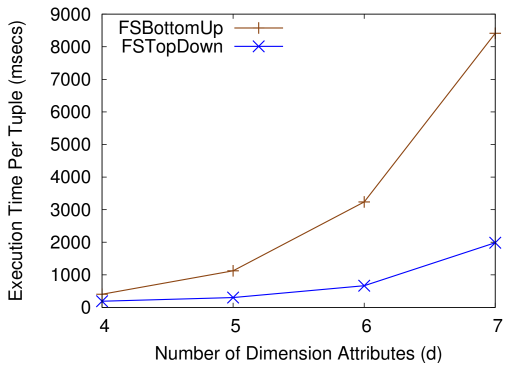
Figure 3.16 also shows that, for the weather dataset, C-CSC could not proceed shorty after 0.2 million tuples were processed. This was also due to memory overflow caused by C-CSC, since it needs to store skyline tuples in their "minimum subspaces". C-CSC did not exhaust memory when it processed the NBA box score dataset (Figure 3.15a), since there were less skyline tuples in the smaller dataset.

Third, in terms of execution time, TopDown/STopDown were outperformed by BottomUp/SBottomUp. The reason is, if a new tuple t dominates a previous tuple t' in constraint C and measure subspace M, TopDown/STopDown must update $\mathcal{MSC}_M^{t'}$. On the contrary, BottomUp/SBottomUp do not carry this overhead; they only need to delete t' from $\mu_{C,M}$. Thus, there is a space-time tradeoff between the top-down and bottom-up strategies.

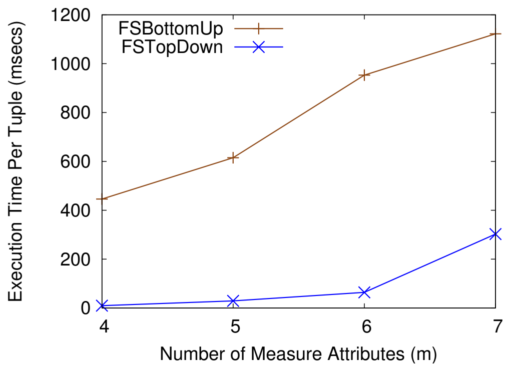
Finally, SBottomUp/STopDown are faster than BottomUp / TopDown, which is the benefit of sharing computation across measure subspaces. Figures 3.15b and 3.15c show that this benefit became more prominent with the increase of both d and m. Figure 3.18 further presents the amount of work done by these algorithms, in terms of compared tuples (Figure 3.18a) and traversed constraints (Figure 3.18b). There are substantial differences between TopDown and STopDown, but the differences between BottomUp and SBottomUp are insignificant. The reason is as follows. STopDown avoids visiting pruned non-skyline constraints, which TopDown cannot avoid. Although SBottomUp avoids such non-skyline constraints too, BottomUp also avoids most of them. The difference between BottomUp and SBottomUp is that BottomUp still visits the boundary non-skyline constraints that are



(a) Varying n, d=5, m=7



(b) Varying d, n=5,000, m=7



(c) Varying m, n=5,000, d=5

Figure 3.19: Execution Time of FSBottomUp and FSTopDown on the NBA Box score Dataset

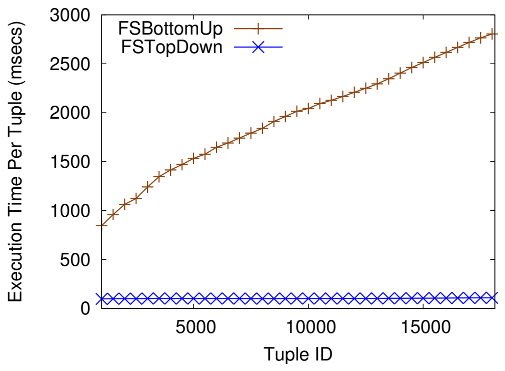


Figure 3.20: Execution Time of FSBottomUp and FSTopDown on the Weather Dataset, Varying n, d=5, m=7

parents of skyline constraints and then skips their ancestors, while SBottomUp skips all non-skyline constraints. Such a difference on boundary non-skyline constraints is not significant.

Results of File-Based Implementation The memory-based implementations of all algorithms store skyline tuples for all combinations of constraints and measure subspaces. As a dataset grows, sooner or later, all algorithms will lead to memory overflow. To address this, we investigated file-based implementations of STopDown and SBottomUp, denoted FSTopDown and FSBottomUp, respectively. We did not include C-CSC in this experiment since Figures 3.14-3.23 clearly show TopDown/STopDown is one order of magnitude faster than C-CSC and consumes about the same amount of memory.

In the file-based implementations, each non-empty $\mu_{C,M}$ is stored as a binary file. Since the size of $\mu_{C,M}$ for any particular constraint-measure pair (C, M) is small, all tuples in the corresponding file are read into a memory buffer when the pair is visited. Insertion and deletion on $\mu_{C,M}$ are then performed on the buffer. When an algorithm finishes process the pair, the file is overwritten by the buffer's content.

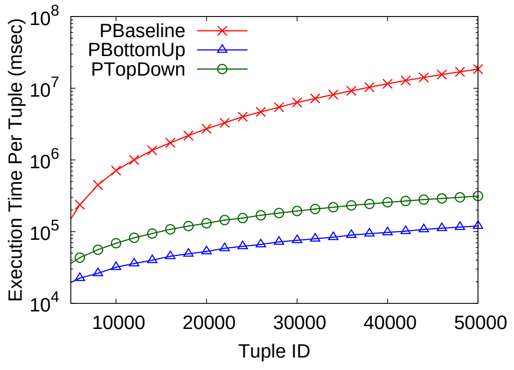
Figure 3.19 uses the same configurations in Figures 3.14 and 3.15 to compare the per-tuple execution times of FSBottomUp and FSTopDown on the NBA box score dataset. Figure 3.20 further compares them on the weather dataset. The figures show that FSTopDown outperformed FSBottomUp by multiple times. Even for only n=5,000, their performance gap was already clear in Figures 3.19b and 3.19c. The reason is as follows. In file-based implementation, while traversing a pair (C, M), a file-read operation occurs if $\mu_{C,M}$ is non-empty. Since FSTopDown stores significantly fewer tuples than FSBottomUp (cf. Figure 3.23), FSTopDown is more likely to encounter empty $\mu_{C,M}$ and thus triggers fewer file-read operations. Further, a file-write operation occurs if the algorithms must update $\mu_{C,M}$. Again, since FSTopDown stores fewer tuples, it requires fewer file-write operations. Hence, although SBottomUp outperformed STopDown on in-memory execution time, FSTopDown triumphed FSBottomUp because I/O-cost dominates in-memory computation.

3.8.3 Experiments on NBA Play by Play Dataset

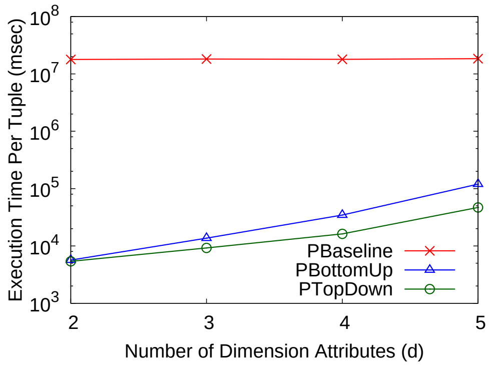
Implementation Details In the implementations of PBaseline, PBottomUp, and PTopDown, the challenge lies in retrieving the set of exclusively dominated tuples for a certain tuple t with regard to a constraint-measure pair (C, M), i.e., $\mathcal{E}^{t.prev}_{C,M}(R)$. In order to resolve this issue, we have maintained a *k*-d tree [7] on full measure space $\mathcal{M}$. Obviously, the one-sided range query $\bigwedge_{m_i \in M}(m_i \leq t.prev.m_i)$ includes $\mathcal{E}^{t.prev}_{\top,M}(R)$. However, $\bigwedge_{m_i \in M}(m_i \leq t.prev.m_i) \gg \mathcal{E}^{t.prev}_{\top,M}(R)$. To get rid of unnecessary search space, we have considered the query $\bigwedge_{m_i \in M}(m_i \leq t.prev.m_i) - \bigwedge_{m_i \in M}(m_i \leq t.m_i)$. This query result includes the tuples in $\mathcal{E}^{t.prev}_{\top,M}(R)$ that are not dominated by t, i.e., $\{t' \in R | t' \in \mathcal{E}^{t.prev}_{\top,M}(R) \wedge t' \not\prec_M t\}$. Now from this query result, we can retrieve the tuples satisfied by C, for each $C \in \mathcal{C}^t$. In other word, we can extract $\{t' \in R | t' \in \mathcal{E}^{t.prev}_{C,M}(R) \wedge t' \not\prec_M t\}$, for each $C \in \mathcal{C}^t$.

The query $\bigwedge_{m_i \in M}(m_i \leq t.prev.m_i) - \bigwedge_{m_i \in M}(m_i \leq t.m_i)$ is the subtraction of these two one-sided range queries: $\bigwedge_{m_i \in M}(m_i \leq t.prev.m_i)$ and $\bigwedge_{m_i \in M}(m_i \leq t.m_i)$. In order to reduce the computation cost, we have derived a tuple t'' from t and $t.prev$ such that $\bigwedge_{m_i \in M}(m_i \leq t.prev.m_i \wedge m_i > t''.m_i) = \bigwedge_{m_i \in M}(m_i \leq t.prev.m_i) - \bigwedge_{m_i \in M}(m_i \leq t.m_i)$. Finally, we have used a single two-sided range query $\bigwedge_{m_i \in M}(m_i \leq t.prev.m_i \wedge m_i > t''.m_i)$ instead of two aforementioned one-sided range queries to extract $\mathcal{E}^{t.prev}_{C,M}(R)$ for all $C \in \mathcal{C}^t$and thus reduced the cost regarding exclusively dominated tuples retrieval.

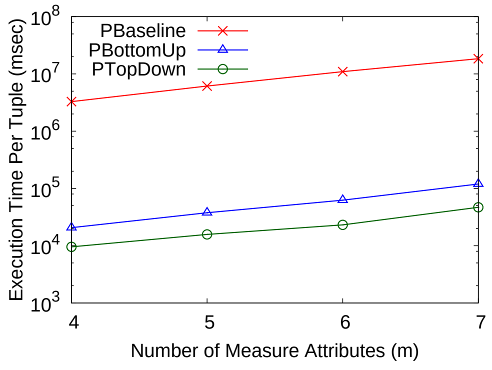
Experimental Results Figure 3.21 compares the per-tuple execution times (by milliseconds, in logarithmic scale) of PBaseline, PBottomUp, and PTopDown on the NBA play by play dataset. Figure 3.21a shows how the per-tuple execution times increase as the algorithms process tuples sequentially by their timestamps. The values of d and m are d = 5 and m = 7, respectively. Figure 3.21b shows the times under varying d, given n = 50,000 and m = 7. Figure 3.21c is for varying m, n = 50,000 and d=5. The figures demonstrate that both PBottomUp and PTopDown outperformed the baselines by orders of magnitude.



(a) Varying n, d=5, m=7



(b) Varying d, n=50,000, m=7



(c) Varying m, n=50,000, d=5

Figure 3.21: Execution Time of PBaseline, PBottomUp, and PTopDown on NBA Play by Play Dataset

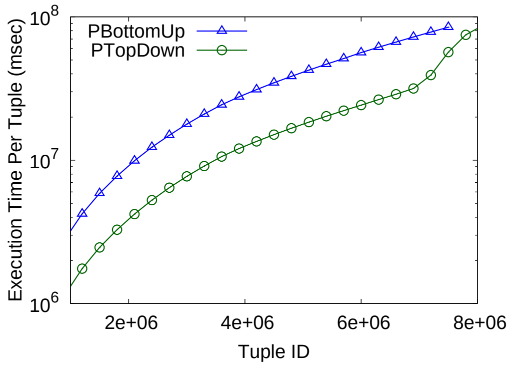
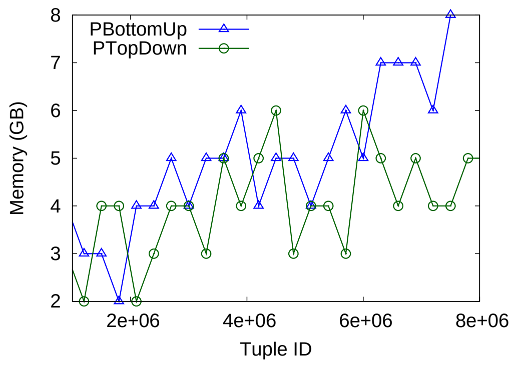
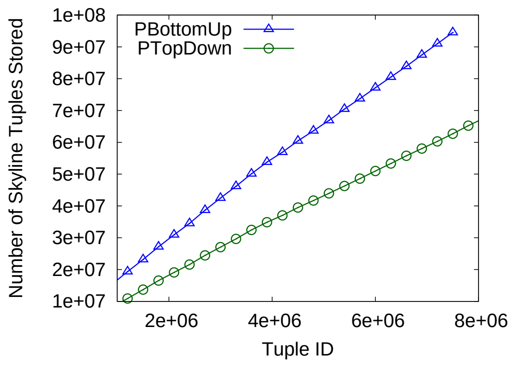


Figure 3.22: Execution Time on the NBA Play by Play Dataset, Varying n, d=5, m=7

Furthermore, Figure 3.21b and Figure 3.21c demonstrate the impact of d and m in these algorithms, respectively. Figure 3.21b depicts that the execution time of PBottomUp and PTopDown increased exponentially by d as the number of constraints to be traversed grows

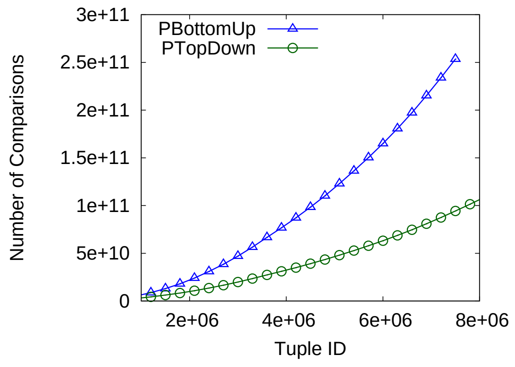


(a) Size of Consumed Memory

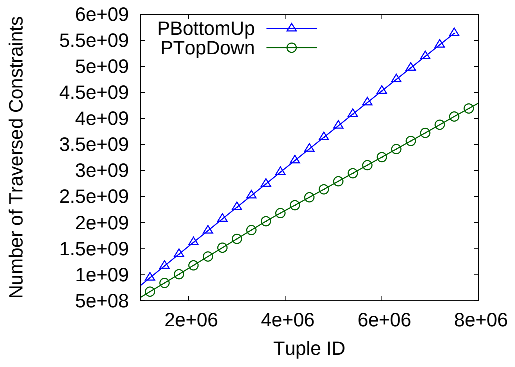


(b) Num of Skyline Tuples Stored

Figure 3.23: Memory Consumption by PBottomUp and PTopDown on the NBA Play by Play Dataset, Varying n, d=5, m=7



(a) Number of Comparisons



(b) Num of Traversed Constraints

Figure 3.24: Work Done by PBottomUp and PTopDown on the NBA Play by Play Dataset, Varying n, d=5, m=7

exponentially. However, the variation of d does not have noticeable impact on PBaseline. While the bottom-up and top-down approaches compare a new tuple t with regard to the tuples stored in the traversed constraints, PBaseline compares t with the tuples in R regardless of the constraints. Therefore, the number of comparisons regarding t is not affected by the value of d. On the contrary, Figure 3.21c shows that the execution time of all these algorithms increased exponentially by m as the number of measure spaces grow exponentially.

Figure 3.22 uses the same configurations in Figure 3.21a to compare PBottomUp and PTopDown. We make the following observations on the results. First, in Figure 3.21b, the execution times of PBottomUp and PTopDown differs with the increase of d. This is because, intuitively, for a certain tuple t, the set of maximal skyline constraints is similar to the set of skyline constraints with respect to a smaller d, i.e., $\mathcal{MSC}_M^t \approx \mathcal{SC}_M^t$. As a result, t needs to go through similar number of traversal as well as comparisons for both algorithms.

Second, in Figure 3.21a, PBottomUp exhausted available JVM heap and were terminated due to memory overflow before 8 million tuples while PTopDown were still running. This phenomenon is explained through Figure 3.23a as it reveals the sizes of consumed memory by these two algorithms. Furthermore, Figure 3.23b shows how the number of stored skyline tuples increases by n in PBottomUp and PTopDown. As we discussed in Section 3.8.2, while the top-down approach stores a tuple only in its maximal skyline constraints, the bottom-up approach stores the tuple in all possible skyline constraints. Thus the corresponding memory consumptions differ.

Finally, in terms of execution time, PBottomUp were outperformed by PTopDown. The reason is clarified through Figure 3.24a and Figure 3.24b as these two figures reveal the work done by the two algorithms, in terms of number of comparisons and number of traversals, respectively. However, it is noticeable that, this behaviour does not concur with the corresponding evaluation of bottom-up and top-down approaches in Section 3.8.2. This interprets that for play by play dataset, the overhead regarding “maximal skyline constraints maintenance” does not cause significant interference. In play by play dataset, most of the tuples contain smaller values in their measure attributes. These tuples are easily dominated by the existing skyline tuples. Since these tuples never get the opportunity to occupy any skyline, they never cause such “maximal skyline constraints maintenance”. In conclusion, with respect to both space and time, PTopDown dominates PBottomUp.

Consider a case of “relegation” with regard to a newly arrived tuple t, a constraint $C \in \mathcal{C}^t$ and the full space $\mathcal{M}$. We can identify all subspaces M in which t is dominated by an existing tuple t' such that $C \in \mathcal{C}^{t,t'}$ (Proposition 4). Suppose $t.prev$ occupied the contextual skyline of $(C, \mathcal{M})$. Since this a case of "relegation", upon the arrival of t, $(C, \mathcal{M})$ includes the exclusively dominated tuples $\mathcal{E}^{t.prev}_{C,\mathcal{M}}(R)$ in its contextual skyline. Now for each $t'' \in \mathcal{E}^{t.prev}_{C,\mathcal{M}}(R)$, PSTopDown needs to determine the constraint-measure pairs (C, M) where t'' belongs to the contextual skyline, by using Procedure PSTopDownNode. In summary, in a case of "relegation", upon the arrival of t, PSTopDown needs to consider the entrance of $\mathcal{E}^{t.prev}_{C,\mathcal{M}}(R)$ as well for each $C \in \mathcal{C}^t$. Practically, $\mathcal{E}^{t.prev}_{C,\mathcal{M}}(R)$ can be significantly larger than $\mathcal{E}^{t.prev}_{C,M}(R)$ even if $|\mathcal{M}| = |M| + 1$. Because of this overhead, PSTopDown is outperformed by PTopDown and it is excluded from the depictions.

3.9 Summary

We studied the novel problem of discovering prominent situational facts, which is formalized as finding the constraint-measure pairs that qualify a new tuple as a contextual skyline tuple. We presented algorithms for efficient and effective discovery of prominent facts. Extensive experiments over three real datasets validated the effectiveness and efficiency of the proposed techniques. Through allowing update on data, our system has become capable to stream live news. We will apply our algorithms on domains other than sports and thus bring out regarding prominent situational facts instantly.

```
Algorithm 12: STopDown
Input: $R(\mathcal{M}, \mathcal{D})$: existing tuples; $t$: the new tuple
Output: $S^t$: the contextual skylines for $t$
1  $S^t \leftarrow$ STopDownRoot();
2  foreach $M \subset \mathcal{M}$ do
3      $S^t \leftarrow S^t \cup$ STopDownNode($M$);
4  $R \leftarrow R \cup \{t\}$;
5  return $S^t$;

Procedure: STopDownRoot ()
1  $S^t \leftarrow \emptyset$;
2  foreach $C \in \mathcal{C}^t$ do
3      $C.pruned \leftarrow$ false;
4      $C.inAnces \leftarrow$ false;
5      foreach $M \subset \mathcal{M}$ do
6          $pruned[C][M] \leftarrow$ false;
7  $Q \leftarrow \emptyset$; $Q.enqueue(\top)$;
8  while not $Q.empty()$ do
9      $C \leftarrow Q.dequeue()$;
10     foreach $t' \in \mu_{C,\mathcal{M}}$ do
11         if $t \prec_{\mathcal{M}} t'$ then Dominated($t', C$);
12         else if $t' \prec_{\mathcal{M}} t$ then Dominates($t'$,$C$,$\mathcal{M}$);
13         foreach $M \subset \mathcal{M}$ do
14             if $t \prec_M t'$ (Proposition 4) then
15                 foreach $C' \in \mathcal{C}^{t,t'}$ do
16                     $pruned[C'][M] \leftarrow$ true;
17     if not $C.pruned$ then
18         $S^t \leftarrow S^t \cup \{(C, \mathcal{M})\}$;
19         if not $C.inAnces$ then $\mu_{C,\mathcal{M}}.insert(t)$ ;
20     EnqueueChildren($C$);
21 return $S^t$;

Procedure: STopDownNode ($M$)
1  $S^t \leftarrow \emptyset$;
2  foreach $C \in \mathcal{C}^t$ do
3      $C.pruned \leftarrow pruned[C][M]$;
4      $C.inAnces \leftarrow$ false;
5  $Q \leftarrow \emptyset$; $Q.enqueue(\top)$;
6  while not $Q.empty()$ do
7      $C \leftarrow Q.dequeue()$; if not $C.pruned$ then
8          $S^t \leftarrow S^t \cup \{(C, M)\}$;
9          foreach $t' \in \mu_{C,M}$ do
10             if $t' \prec_M t$ then Dominates($t', C, M$);
11         if not $C.inAnces$ then $\mu_{C,M}.insert(t)$;
12         ;
13     EnqueueChildren($C$);
14 return $S^t$;
```

Algorithm 13: PBottomUp

```
Input: $R(\mathcal{M}, \mathcal{D})$: existing tuples; $t$: the new tuple
Output: $S^t$: the contextual skylines for $t$
$S^t \leftarrow \emptyset$; if $t \succ_M t.prev$ then BottomUp$(R, t)$ ;
else if $t \prec_M t.prev$ then
    foreach $M \subseteq \mathcal{M}$ do
        foreach $C \in \mathcal{C}^t$ do $C.pruned \leftarrow$ false;
        $Q \leftarrow \emptyset$; $Q.enqueue(\perp(\mathcal{C}^t))$;
        while not $Q.empty()$ do
            $C \leftarrow Q.dequeue()$;
            if not $\mu_{C,M}.contains(t.prev)$ then
                $C' \in \mathcal{A}^t_C$ $C'.pruned \leftarrow$ true;
                continue;
            $\mu_{C,M}.delete(t.prev)$;
            $t' \in \mathcal{E}^{t.prev}_{C,M}(R - \{t\})$
            $\mu_{C,M}.insert(t')$;
            $dominated \leftarrow$ false;
            $t' \in \mu_{C,M}$ if $t \prec_M t'$ then
                $dominated \leftarrow$ true;
                $C' \in \mathcal{A}^t_C$ $C'.pruned \leftarrow$ true;
                break;
            else if $t' \prec_M t$ then
                $\mu_{C,M}.delete(t')$
            if not $dominated$ then
                $S^t \leftarrow S^t \cup \{(C, M)\}$;
                $\mu_{C,M}.insert(t)$;
            $C' \in \mathcal{P}^t_C$ if (not $Q.contains(C')$)
            and (not $C'.pruned$) then
            $Q.enqueue(C')$;
    $R \leftarrow R \cup \{t\}$;
    return $S^t$;
else
    foreach $M \subseteq \mathcal{M}$ do
        foreach $C \in \mathcal{C}^t$ do $C.pruned \leftarrow$ false;
        $Q \leftarrow \emptyset$; $Q.enqueue(\perp(\mathcal{C}^t))$;
        while not $Q.empty()$ do
            $C \leftarrow Q.dequeue()$;
            if not $\mu_{C,M}.contains(t.prev)$ then
                foreach $C' \in \mathcal{A}^t_C$ do
                    $C'.pruned \leftarrow$ true;
                continue;
            $\mu_{C,M}.delete(t.prev)$;
             $\mu_{C,M}.insert(t)$;
            foreach $C' \in \mathcal{P}^t_C$ do
                if (not $Q.contains(C')$) and
                 (not $C'.pruned$) then
                 $Q.enqueue(C')$;
    $R \leftarrow R \cup \{t\}$;
    return $S^t$;
```

Algorithm 14: PTopDown

```
Input: $R(\mathcal{M}, \mathcal{D})$: existing tuples; $t$: the new tuple
Output: $S^t$: the contextual skylines for $t$
1  $S^t \leftarrow \emptyset$; if $t \succ_M t.prev$ then TopDown$(R, t)$ ;
2  else if $t \prec_M t.prev$ then
3      $M \subseteq \mathcal{M}$  $Q \leftarrow \emptyset$; $C \in \mathcal{C}^t$  $C.pruned \leftarrow$ false;
       $C.inAnces \leftarrow$ false  $C \in \mathcal{MSC}_M^{t.prev}$
       $\mu_{C,M}.delete(t.prev)$;
4      foreach $t' \in \mathcal{E}_{C,M}^{t.prev}(R - \{t\})$ do
        $\mu_{C,M}.insert(t')$ ;
5      $Q.enqueue(C)$  while not $Q.empty()$ do
6          $C \leftarrow Q.dequeue()$;
7          if not $C.inAnces$ then
8              $t' \in \mu_{C,M}$ if $t \prec_M t'$ then
9                  Dominated$(t', C)$;
10             else if $t' \prec_M t$ then
11                 Dominates$(t', C, M)$;
12         else
13             $t' \in \mathcal{E}_{C,M}^{t.prev}(R - \{t\})$ if $t' \prec_M t$ then
14                 $stored \leftarrow$ false;
15                 $C' \in \mathcal{A}^C \cap \mathcal{SC}_M^{t.prev}$ if
                   $t' \in \mathcal{E}_{C',M}^{t.prev}(R - \{t\})$ then
                   $stored \leftarrow$ true; break ;
16                 if not $stored$ then  $\mu_{C,M}.insert(t')$ ;
17         if not $C.pruned$ then
18             $S^t \leftarrow S^t \cup \{(C, M)\}$;
19             if not $C.inAnces$ then
20                 $\mathcal{MSC}_M^t \leftarrow$
                    $\mathcal{MSC}_M^t \cup \{C\}$; $\mu_{C,M}.insert(t)$;
21         EnqueueChindren$(C)$;
22     $R \leftarrow R \cup \{t\}$; return $S^t$;
23 else
24     $\mathcal{MSC}_M^t \leftarrow \mathcal{MSC}_M^{t.prev}$;
25     foreach $C \in \mathcal{MSC}_M^{t.prev}$ do
26         $\mu_{C,M}.delete(t.prev)$;
27         $\mu_{C,M}.insert(t)$
28     return $S^t$;
```

Algorithm 15: PSTopDown

Input: $R(\mathcal{M}, \mathcal{D})$: existing tuples; t: the new tuple

Output: S^t: the contextual skylines for t

```
1 $S^t \leftarrow$ PSTopDownRoot();
2 foreach $M \subset \mathcal{M}$ do
3     $S^t \leftarrow S^t \cup$ PSTopDownNode($M$);
4 $R \leftarrow R \cup \{t\}$;
5 return $S^t$;
```

Procedure: PSTopDownRoot ()

```
1 if $t \succ_{\mathcal{M}} t.prev$ then
2     STopDownRoot();
3 else if $t \prec_{\mathcal{M}} t.prev$ then
4     foreach $C \in \mathcal{MSC}^{t.prev}_{\mathcal{M}}$ do
5         $\mu_{C,\mathcal{M}}.delete(t.prev)$;
6         foreach $t' \in \mathcal{E}^{t.prev}_{C,\mathcal{M}}(R - \{t\})$ do
7             $\mu_{C,\mathcal{M}}.insert(t')$;
8     STopDownRoot();
```

Procedure: PSTopDownNode (M)

```
1 $S^t \leftarrow \emptyset$;
2 if $t \succ_M t.prev$ then
3     STopDownNode($M$);
4 else if $t \prec_M t.prev$ then
5     foreach $C \in \mathcal{C}^t$ do
6         $C.pruned \leftarrow pruned[C][M]$;
7         $C.inAnces \leftarrow$ false;
8     $Q \leftarrow \emptyset$;
9     foreach $C \in \mathcal{MSC}^{t.prev}_{M}$ do
10        $\mu_{C,M}.delete(t.prev)$;
11        foreach $t' \in \mathcal{E}^{t.prev}_{C,M}(R - \{t\})$ do
12            $\mu_{C,M}.insert(t')$;
13        $Q.enqueue(C)$;
14    while not $Q.empty()$ do
15        $C \leftarrow Q.dequeue()$;
16        if not $C.pruned$ then
17            $S^t \leftarrow S^t \cup \{(C, M)\}$;
18            foreach $t' \in \mu_{C,M}$ do
19                if $t' \prec_M t$ then Dominates($t', C, M$);
20            if not $C.inAnces$ then
21                $\mu_{C,M}.insert(t)$;
22        EnqueueChildren($C$);
23    return $S^t$;
24 else
25    $\mathcal{MSC}^t_M \leftarrow \mathcal{MSC}^{t.prev}_M$;
26    foreach $C \in \mathcal{MSC}^{t.prev}_M$ do
27        $\mu_{C,M}.delete(t.prev)$; $\mu_{C,M}.insert(t)$;
28    return $S^t$;
```

d	dimension space $\mathcal{D}$
2	player, season
3	player, season, team
4	player, season, team, opp_team
5	player, season, month, team, opp_team
6	position, college, state, season, team, opp_team
7	position, college, state, season, month, team, opp_team

Table 3.6: Dimension Spaces for Different Values of d

m	measure space $\mathcal{M}$
4	points, rebounds, assists, blocks
5	points, rebounds, assists, blocks, steals
6	points, rebounds, assists, blocks, steals, fouls
7	points, rebounds, assists, blocks, steals, fouls, turnovers

Table 3.7: Measure Spaces for Different Values of m

CHAPTER 4

Conclusion

In this dissertation, we studied efficient evaluation of contextual and reverse Pareto-optimality queries with regard to two types of queries, namely preference query and skyline query. In this regard, we formalized the real-world problem of finding "outstanding" objects as finding Pareto-optimal objects. To resolve the challenges in evaluating such queries, we developed efficient algorithms. Going forward, we will apply and adjust our algorithms in additional domains where there are important applications of contextual and reverse Pareto-optimality queries. Such a study will assist researchers and practitioners in supporting Pareto-optimality queries in broader contexts.

REFERENCES

[1] https://en.wikipedia.org/wiki/pareto_efficiency. Accessed: Nov. 2016.

[2] http://www.newsday.com/sports/columnists/neil-best/hirdt-enjoying-long-run-as-stats-guru-1.3174737. Accessed: Jul. 2013.

[3] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Acm sigmod record*, volume 22, pages 207–216. ACM, 1993.

[4] F. Alvanaki, E. Ilieva, S. Michel, and A. Stupar. Interesting event detection through hall of fame rankings. In *DBSocial*, pages 7–12, 2013.

[5] F. Angiulli, G. Ianni, and L. Palopoli. On the complexity of inducing categorical and quantitative association rules. *Theoretical Computer Science*, 314(1-2):217–249, 2004.

[6] O. Barndorff-Nielsen and M. Sobel. On the distribution of the number of admissible points in a vector random sample. *Theory of Probability & Its Applications*, 11(2), 1966.

[7] J. L. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. Softw. Eng.*, 5(4):333–340, July 1979.

[8] A. Bharadwaj. Automatic Discovery of Significant Events From Databases. Master's thesis, Universioty of Tesax at Arlington, Dec. 2011.

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

[10] C.-Y. Chan, P.-K. Eng, and K.-L. Tan. Stratified computation of skylines with partially-ordered domains. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, 2005.

[11] F. Chierichetti, R. Kumar, S. Pandey, and S. Vassilvitskii. Finding the jaccard median. In *SODA*, 2010.

[12] J. Chomicki. Preference formulas in relational queries. *ACM Transactions on Database Systems (TODS)*, 28(4), 2003.

[13] S. Cohen, J. T. Hamilton, and F. Turner. Computational journalism. *Commun. ACM*, 54(10):66–71, Oct. 2011.

[14] S. Cohen, C. Li, J. Yang, and C. Yu. Computational journalism: A call to arms to database researchers. In *CIDR*, pages 148–151, 2011.

[15] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *Proceedings of the VLDB Endowment*, 2007.

[16] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, 1996.

[17] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[18] Q. Fan, Y. Li, D. Zhang, and K. Tan. Discovering newsworthy themes from sequenced data: A step towards computational journalism. *TKDE*, 29(7):1398–1411, 2017.

[19] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 3rd edition, 2011.

[20] N. Hassan, A. Sultana, Y. Wu, G. Zhang, C. Li, J. Yang, and C. Yu. Data in, fact out: Automated monitoring of facts by factwatcher. *PVLDB*, 7(13):1557–1560, 2014.

[21] X. Jiang, C. Li, P. Luo, M. Wang, and Y. Yu. Prominent streak discovery in sequence data. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1280–1288, 2011.

[22] C. Kalyvas and T. Tzouramanis. A survey of skyline query processing. *Computing Research Repository*, abs/1704.01788:1–128, 2017.

[23] T. Kamishima and S. Akaho. Efficient clustering for orders. In *Mining Complex Data*. 2009.

[24] T. Kamishima and J. Fujiki. Clustering orders. In *Discovery Science*, 2003.

[25] W. Kießling. Foundations of preferences in database systems. In *Proceedings of the VLDB Endowment*, 2002.

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

[27] K. C. Lee, B. Zheng, H. Li, and W.-C. Lee. Approaching the skyline in z order. In *Proceedings of the 33rd international conference on Very large data bases*, 2007.

[28] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *21st International Conference on Data Engineering (ICDE'05)*, 2005.

[29] M. Morse, J. M. Patel, and W. I. Grosky. Efficient continuous skyline computation. *Information Sciences*, 177(17), 2007.

[30] 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. *ACM Trans. Database Syst.*, 31(4):1335–1381, 2006.

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

[32] N. Sarkas, G. Das, N. Koudas, and A. K. Tung. Categorical skylines for streaming data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008.

[33] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, 2001.

[34] A. Sultana, N. Hassan, C. Li, J. Yang, and C. Yu. Incremental discovery of prominent situational facts. In *ICDE*, pages 112–123, 2014.

[35] A. Sultana and C. Li. Continuous monitoring of pareto frontiers on partially ordered attributes for many users. In *EDBT*, pages 85–96, 2018.

[36] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. *IEEE Transactions on Knowledge and Data Engineering*, 18(3), 2006.

[37] A. Ukkonen. Clustering algorithms for chains. *The Journal of Machine Learning Research*, 12, 2011.

[38] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Norvag. Reverse top-k queries. In *ICDE*, 2010.

[39] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis. Branch-and-bound algorithm for reverse top-k queries. In *SIGMOD*, 2013.

[40] B. Walenz et al. Finding, monitoring, and checking claims computationally based on structured data. In *Computation+Journalism Symposium*, 2014.

[41] R. C.-W. Wong, A. W.-C. Fu, J. Pei, Y. S. Ho, T. Wong, and Y. Liu. Efficient skyline querying with variable user preferences on nominal attributes. *Proceedings of the VLDB Endowment*, 1(1), 2008.

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

[43] R.-W. Wong, J. Pei, A.-C. Fu, and K. Wang. Online skyline analysis with dynamic preferences on nominal attributes. *IEEE Transactions on Knowledge and Data Engineering*, 21(1), 2009.

[44] P. Wu, D. Agrawal, O. Egecioglu, and A. El Abbadi. Deltasky: Optimal maintenance of skyline deletions without exclusive dominance region generation. In *2007 IEEE 23rd International Conference on Data Engineering*, 2007.

[45] T. Wu, D. Xin, Q. Mei, and J. Han. Promotion analysis in multi-dimensional space. *Proceedings of the VLDB Endowment*, 2(1):109–120, 2009.

[46] Y. Wu, P. K. Agarwal, C. Li, J. Yang, and C. Yu. On "one of the few" objects. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1487–1495, 2012.

[47] Y. Wu, B. Harb, J. Yang, and C. Yu. Efficient evaluation of object-centric exploration queries for visualization. *PVLDB*, 8(12):1752–1763, 2015.

[48] T. Xia and D. Zhang. Refreshing the sky: the compressed skycube with efficient support for frequent updates. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006.

[49] A. Yu, P. K. Agarwal, and J. Yang. Processing a large number of continuous preference top-k queries. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[50] G. Zhang, X. Jiang, P. Luo, M. Wang, and C. Li. Discovering general prominent streaks in sequence data. *TKDD*, 8(2):9:1–9:37, 2013.

[51] M. Zhang and R. Alhajj. Skyline queries with constraints: Integrating skyline and traditional query operators. *DKE*, 69(1):153 – 168, 2010.

[52] S. Zhang, N. Mamoulis, D. W. Cheung, and B. Kao. Efficient skyline evaluation over partially ordered domains. *Proceedings of the VLDB Endowment*, 3(1-2), 2010.