Incremental Discovery of Prominent Situational Facts

Afroza Sultana[1] Naeemul Hassan[1] Chengkai Li[1] Jun Yang[2] Cong Yu[3]
[1]University of Texas at Arlington, [2]Duke University, [3]Google Research

ABSTRACT

We study the novel problem 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. Effective and efficient identification of these facts assists journalists in reporting, one of the main goals of computational journalism. Technically, 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, when a set of measure attributes are compared. 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 with every 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 also adopt a simple prominence measure to rank the discovered facts when they are numerous. Experiments over two real datasets validate the effectiveness and efficiency of our techniques.

1. INTRODUCTION

Computational journalism emerged recently as a young interdisciplinary field [5] that brings together experts in journalism, social sciences and computer science, and advances journalism by innovations in computational techniques such as information retrieval, speech recognition, video analysis, and visualization. Databases and data mining researchers have also started to push the frontiers of this field [6, 7, 10]. 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."[1]
- "The social world's most viral photo ever generated 3.5 million likes, 170,000 comments and 460,000 shares by Wednesday afternoon."[2]

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 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 paper 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* [4].

We study the problem of finding *situational facts* pertinent to new tuples in an ever-growing database, where the new tuples capture real-world events. We propose algorithms that, whenever a new tuple t enters (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 the arrival of t.

[1] http://espn.go.com/espn/elias?date=20130205

[2] http://www.cnbc.com/id/49728455/President_Obama_Sets_New_Social_Media_Record

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 1: A Mini-world of Basketball Gamelogs

Example 1. Consider the mini-world of basketball gamelogs R in Table 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 $\sigma_{\text{month}=\textit{Feb.}}(R)$ (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 $\sigma_{\text{team}=\textit{Celtics}\wedge\ \text{opp\_team}=\textit{Nets}}(R)$ 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 the records much less interesting to fans, and risk losing them to rival media. Timely identification of situational facts is also critical in areas beyond journalism. Investors want to know facts related to stock trading as soon as possible, in order to make informed investment decisions. Facts discovered from weather data can assist scientists in identifying extreme weather conditions and thus helping government and the public in coping with the weather.

Simple forms of situational facts on a single measure and a complete table, such as 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 possible 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 the fact. This is how Elias Sports Bureau tackles the task and provides sports records (such as the aforementioned record by NBA player Paul George) to many sports media [3]. With ever-growing data and limited human resources, such manual checking is be time-consuming and error-prune. The low efficiency of this approach 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 paper 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. The 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 (i.e., every context containing t) under every possible measure subspace. The obvious low-efficiency of this approach has three culprits—the 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 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 a tuple t' in a particular measure subspace M, then t does not belong to the contextual skyline of constraint-measure pair (C, M) for any constraint C satisfied by both t and t'. For example, in the full measure space $\mathcal{M}$, since t_7 is dominated by t_3, 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 either (season=*1995-96*, $\mathcal{M}$) or (no constraint, $\mathcal{M}$). Based on this intuition, we can examine the constraints satisfied by t in a certain order, such that comparison of t with skyline tuples associated with already examined constraints can be used to prune remaining constraints from consideration.

3) Sharing computation across measure subspaces Since 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'. 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 skyline tuple t_2 in $\mathcal{M}$, the algorithms realize that t_7 has smaller values on both 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 no 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 their prominence. Reporters and experts can choose to investigate the top-*k* facts or the facts with prominence values above a threshold.

The contributions of this paper 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.
- We devise efficient algorithms based on three main ideas—tuple reduction, constraint pruning and sharing computation across measure subspaces.
- We propose a simple prominence measure for ranking situational facts and discovering prominent situational facts.

- We conduct extensive experiments on two real datasets (NBA dataset and weather dataset) to investigate their prominent situational facts and to study the efficiency of various proposed algorithms and their tradeoffs.

2. RELATED WORK

Pioneers in data journalism have had considerable success in developing software to write computer-generated stories. One such software from StatSheet[3] writes about sports games based on game statistics. Another one, from Narrative Science,[4] is used by multiple clients in writing sports and stock earnings preview stories. The stories produced by such software follow certain writing patterns to narrate structured data such as box scores and play-by-play data for a basketball game and the earnings data for a company. The focus is 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 has been extensively investigated in recent years, since Börzsönyi et al. [4] brought the concept to the database field and introduced their algorithms. In [4] and most studies afterwards, it is assumed that both the context of tuples in comparison and the measure space are given as query conditions. With regard to context, a few papers (e.g., Zhang et al. [12]) integrate the evaluation of a constraint with the finding of skyline tuples in the corresponding context in a given measure space. With regard to measure, Pei et al. [8] proposed techniques to find skyline points in all subspaces of the measure space, oftentimes termed *skyline cube* in the literature. Xia et al. [11] further studied how to update the skyline cube when data change frequently. While these prior studies *find answers* (i.e., skyline points) for a given query (i.e., a context, a measure space, or their combination), this paper studies the reverse problem of *finding queries* (i.e., constraint-measure pairs that qualify a tuple as a contextual skyline tuple, among all possible contexts and measure subspaces) for an answer (i.e., a new tuple). In other words, we find ways to make a tuple stand out.

Promotion analysis by ranking [9] aims at finding the contexts (given by conjunctive constraints on dimension attributes) in which an object is ranked high. It has two key differences from this paper. (1) Objects are ranked by a single score attribute in [9], 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. (2) Wu et al. [9] considered one-shot computation on static data, while our focus is on incremental discovery on dynamic data. Due to the conceptual and modeling differences, the algorithmic approaches taken in the two works are also fundamentally different.

Wu et al. [10] formulated the concept of *one-of-the-few objects* and proposed algorithms for finding such objects. A *k-skyband* object is an object that is dominated by less than k other objects. The *one-of-the-τ object* problem is to find the largest k value such that the number of k-skyband objects is no more than τ. They consider all possible measure subspaces in defining dominance relation but do not consider different contexts formed by constraints. Similar to [9], it focuses on one-shot computation on static data.

Table 2 summarizes the differences among the aforementioned previous studies and this paper, along three modeling aspects—whether they consider all possible contexts defined on dimension attributes, all measure subspaces, and incremental computation on dynamic data.

[3] http://statsheet.com/ [4] http://www.narrativescience.com/

	all possible contexts	measure subspaces	incremental
[12]	no	no	no
[8]	no	yes	no
[11]	no	yes	yes
[9]	yes	no	no
[10]	no	yes	no
this work	yes	yes	yes

Table 2: Comparing Related Work on Three Modeling Aspects

$R(\mathcal{D}; \mathcal{M})$	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 $\trianglelefteq$
$C_1 \trianglelefteq (\trianglelefteq) C_2$	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: Notations

3. PROBLEM STATEMENT

We first provide a formal description of our data model and problem statement. Table 3 lists the major notations in the paper.

Consider a relational schema $R(\mathcal{D}; \mathcal{M})$, where the *dimension space* is defined by a set of *dimension attributes* $\mathcal{D}$=$\{d_1, \ldots, d_n\}$, and similarly the *measure space* is defined by a set of *measure attributes* $\mathcal{M}$=$\{m_1, \ldots, m_s\}$, and *constraints* are specified on the dimension attributes. Dominance relation for skyline operation is defined on the measure attributes. 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 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.

Definition 1 (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 $bound(C)$.

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 2. Given Table 4, an example constraint is $C = \langle a_1, *, c_1 \rangle$ in which attribute d_2 is unbound. $\sigma_C(R) = \{t_2, t_5\}$. ■

Definition 2 (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. $t' \succ_M t\}$. ■

Note that we use the general term "better than" in the above definition, which can mean either "larger than" or "smaller than" for numeric attributes and either "ordered before" or "ordered after"

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 4: Running Example

for ordinal attributes, depending on application semantics. Furthermore, 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 on every attribute.

Definition 3 (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 3. For Table 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 We formally define the *situational fact discovery problem* as follows. Given an append-only table with schema $R(\mathcal{D}; \mathcal{M})$ and the last tuple t that was appended onto R, find each pair of constraint $C \in \mathcal{C}_\mathcal{D}$ and measure subspace $M \subseteq \mathcal{M}$ 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 skylines for t", even though rigorously speaking it is the set of (C, M) pairs whose corresponding contextual skylines include t.

4. SOLUTION OVERVIEW

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

Definition 4 (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 all such satisfied constraints by $\mathcal{C}^t_\mathcal{D}$ or simply $\mathcal{C}^t$ when $\mathcal{D}$ is clear in context. It follows immediately that given any $C \in \mathcal{C}^t$, $t \in \sigma_C(R)$. ■

For every $C \in \mathcal{C}^t$, $C.d_i$ can attain two possible values $\{*, t.d_i\}$. Hence, $\mathcal{C}^t$ has 2^n constraints in total for $|\mathcal{D}| = n$. Algorithm 1 is a simple procedure for finding all constraints of $\mathcal{C}^t$. This is a routine used in all algorithms. Note that it generates the constraints from the most general one to the most specific one. The most general constraint $\top$ has no bound attributes ($bound(\top) = 0$), i.e., $\top = \langle *, *, \ldots, *\rangle$. The most specific constraint is $\langle t.d_1, t.d_2, \ldots, t.d_n\rangle$. The algorithm makes sure a constraint is not generated twice, for efficiency consideration, 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 2. The obvious inefficiency of this approach has three culprits—the exhaustive comparison with *every tuple*, for *every constraint* and in *every measure subspace*. We thus devise three corresponding ideas to counter these causes, as follows:

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

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 the set $\mathcal{C}^t$ of constraints satisfied by a given t, we need to determine whether t enters any contextual skyline for every constraint in $\mathcal{C}^t$. To prune the number of constraints we need to consider, we note the following property: if t is dominated by a skyline tuple t' under a measure subspace M, then t is not in the contextual skyline of constraint-measure pair (C, M) for any such C that is satisfied by both t and t'.

To enable constraint pruning, we organize the constraints to be considered into a lattice. Note that all tuple-satisfied constraints in $\mathcal{C}^t$ form 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 5 (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 $C_1 \triangleleft C_2$ or $C_2 \triangleright C_1$) 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 4. Consider two constraints $C_1 = \langle a, b, c\rangle$ 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 5, C_1 is subsumed by C_2, i.e. $C_1 \triangleleft C_2$. ■

Definition 6 (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* $\top$ that subsumes every other constraint in $\mathcal{C}_\mathcal{D}$. $\top$ is the most general constraint, which has no bound attributes, i.e., $\top = \langle *, *, \ldots, *\rangle$. 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 7 (Lattice of Tuple-Satisfied Constraints)**.** Given a tuple $t \in R(\mathcal{D}; \mathcal{M})$, $\mathcal{C}^t \subseteq \mathcal{C}_\mathcal{D}$ by definition. In fact, $(\mathcal{C}^t, \trianglelefteq)$ is a lattice. Its top element is $\top$. The bottom element, denoted $\bot(\mathcal{C}^t)$, is $\langle t.d_1, t.d_2, \ldots, t.d_n\rangle$. It is one of the minimal elements in $\mathcal{C}_\mathcal{D}$.

Given $C \in \mathcal{C}^t$, we denote C's ancestors, descendants, parents and children within the lattice 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 $\mathcal{CH}^t_C \subseteq \mathcal{CH}_C$. ■

Algorithm 1: Find $\mathcal{C}^t$

Input: $t \in R$
Output: $\mathcal{C}^t$: constraints satisfied by t

```
1  C^t ← ∅;
2  Q ← ∅; Q.enqueue(⊤);
3  while not Q.empty() do
4      C ← Q.dequeue();
5      C^t ← C^t ∪ {C};
6      i ← n;
7      while i > 0 and C.d_i = * do
8          C' ← C;
9          C'.d_i ← t.d_i;
10         Q.enqueue(C');
11         i ← i − 1;
12 return C^t;
```

Algorithm 2: BruteForce

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

```
1  S^t ← ∅;
2  foreach M ⊆ M do
3      foreach C ∈ C^t do
4          pruned ← false;
5          foreach t' ∈ R do
6              if t ≺_M t' and t' ∈ σ_C(R) then
7                  pruned ← true;
8                  break;
9          if not pruned then S^t ← S^t ∪ {(C, M)};
10 R ← R ∪ {t};
11 return S^t;
```

Algorithm 3: BaselineSeq

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

```
1  S^t ← ∅;
2  foreach M ⊆ M do
3      S ← C^t;
4      foreach t' ∈ R do
5          if t ≺_M t' then S ← S − C^{t,t'};
6      foreach C ∈ S do
7          S^t ← S^t ∪ {(C, M)};
8  R ← R ∪ {t};
9  return S^t;
```

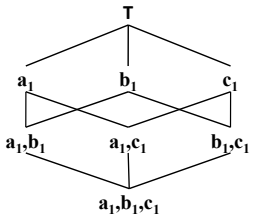


Figure 1: Lattice of Tuple-Satisfied Constraints for t_5

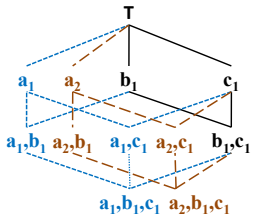


Figure 2: Lattice Intersection of $\mathcal{C}^{t_4}$ and $\mathcal{C}^{t_5}$

Example 5. In Figure 1, we present lattice $\mathcal{C}^{t_5}$ for t_5 in Table 4. For simplicity, we omit values on unbound dimension attributes. For instance, $\langle *, *, c_1\rangle$ is represented as c_1. Consider $C = \langle a_1, *, c_1\rangle$. $\mathcal{A}_C^{t_5} = \{\top, \langle a_1, *, *\rangle, \langle *, *, c_1\rangle\}$, $\mathcal{P}_C^{t_5} = \{\langle a_1, *, *\rangle, \langle *, *, c_1\rangle\}$, $\mathcal{CH}_C^{t_5} = \{\langle a_1, b_1, c_1\rangle\}$, and $\mathcal{D}_C^{t_5}$=$\{\langle a_1, b_1, c_1\rangle\}$. ■

Definition 8 (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 the two lattices with respect to t and t'. $\mathcal{C}^{t,t'}$ is non-empty and is also a lattice. According to Definition 7, 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 any common attribute value. ■

Example 6. Figure 2 shows $\mathcal{C}^{t_4}$ and $\mathcal{C}^{t_5}$ for t_4 and t_5 in Table 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 $t \notin \lambda_M(\sigma_{\perp(\mathcal{C}^{t,t'})}(R))$. It follows that, according to the Proposition 2, $t \notin \lambda_M(\sigma_C(R))$ for all $C \in \mathcal{C}^{t,t'}$. This is captured by 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. 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 first introduce two baseline algorithms, BaselineSeq and BaselineIdx, which are not as naive as the brute-force approach in Algorithm 2. They exploit the aforementioned Proposition 3 in a straightforward way. Upon the arrival of a new tuple t, for each measure subspace M, these algorithms identify existing tuples t' dominating t. The two algorithms differ in that BaselineSeq sequentially compares t with every existing tuple while BaselineIdx uses a multi-dimensional index to find tuples dominating t.

The pseudo code of BaselineSeq is in Algorithm 3. S is initialized to be $\mathcal{C}^t$, the set of all constraints satisfied by t (Line 3). Whenever BaselineSeq encounters a tuple 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 procedure is independently repeated for every measure subspace.

The pseudo code of BaselineIdx is fairly similar to Algorithm 3 and is thus omitted. Instead of comparing t with all existing 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 multi-dimensional index (specifically a k-d tree [1, 2]) on the full measure space $\mathcal{M}$.

5. ALGORITHMS

This section presents our main algorithms. We start with two algorithms, BottomUp (Section 5.1) and TopDown (Section 5.2), which exploit the ideas of tuple reduction and constraint pruning. We then extend the two algorithms to enable sharing of computation across measure subspaces (Section 5.3).

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, both BottomUp and TopDown store and maintain skyline tuples for 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.

Algorithm 4: BottomUp

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

```
1  S^t ← ∅;
2  foreach M ⊆ M do
3      foreach C ∈ C^t do C.pruned ← false;
4      Q ← ∅; Q.enqueue(⊥(C^t));
5      while not Q.empty() do
6          C ← Q.dequeue();
7          dominated ← false;
8          foreach t' ∈ μ_{C,M} do
9              if t ≺_M t' then
10                 dominated ← true;
11                 foreach C' ∈ A^t_C do
12                     C'.pruned ← true; break;
13             else if t' ≺_M t then μ_{C,M}.delete(t');
14         if not dominated then
15             S^t ← S^t ∪ {(C, M)};
16             μ_{C,M}.insert(t);
17             foreach C' ∈ P^t_C do
18                 if (not Q.contains(C')) and (not C'.pruned)
                   then Q.enqueue(C');
19 R ← R ∪ {t};
20 return S^t;
```

These two algorithms traverse, for each measure subspace M, the lattice of tuple-satisfied constraints $\mathcal{C}^t$ by certain order. When a particular 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, based on 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., any C' in $\mathcal{C}^{t,t'}$). This property allows the algorithms to avoid comparisons with skyline tuples associated with such constraints.

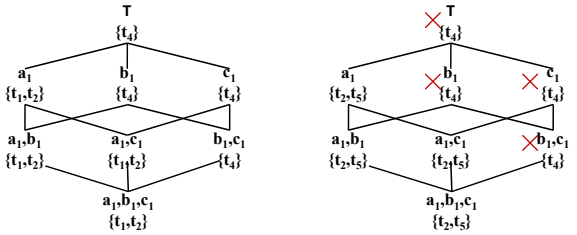
The two algorithms differ by how skyline tuples are stored in $\mu_{C,M}$. BottomUp stores a tuple for every such constraint that qualifies it as a contextual skyline tuple, while TopDown only stores it for the topmost such constraints. In our ensuing discussion, we use invariants to formalize what must be stored in $\mu_{C,M}$. The algorithms also differ in order with which they traverse the tuple-satisfied constraints in lattice $\mathcal{C}^t$. BottomUp visits the constraints in a bottom-up fashion, 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 tradeoffs between space and time. TopDown requires much less space than BottomUp since it avoids storing duplicate skyline tuples as much as possible. This saving in space comes at the cost of execution efficiency, due to more complex operations in TopDown.

5.1 Algorithm BottomUp

BottomUp (Algorithm 4) 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, for 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 Algorithm 4, where the bottom of $\mathcal{C}^t$ is inserted into a queue Q. As long as Q 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 8). Various actions are taken, depending on the comparison result. **1)** If t is dominated by any tuple t', the comparisons with remaining tuples in $\mu_{C,M}$ are skipped (Line 12). 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 the lattice $\mathcal{C}^t$ bottom-up, the dominating tuple t' must be encountered at the bottom of $\mathcal{C}^{t,t'}$. The algorithm thus skips the comparisons with all tuples stored for C's ancestors (Line 12). **2)** If t dominates a current skyline tuple t', t' is removed from $\mu_{C,M}$ (Line 13). **3)** If t is not dominated by any tuple in $\mu_{C,M}$, then it is inserted into $\mu_{C,M}$ (Line 16) and (C, M) corresponds to a contextual skyline for t (Line 15). Furthermore, each such parent constraint of C that is not already pruned is inserted into Q, for continuation of the bottom-up traversal (Line 17).



(a) Before t_5 (b) After t_5

Figure 3: Execution of BottomUp in Measure Subspace $\{m_1, m_2\}$

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 13, 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.

Hence, 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 7. We use Figure 3 to explain the execution of BottomUp on Table 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. Figure 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 are two skyline tuples stored in $\mu_{\bot(\mathcal{C}^{t_5}),M}$—$t_1$ and t_2. In subspace M, t_5 dominates t_1 and is incomparable to t_2. Hence, t_1 is deleted from $\mu_{\bot(\mathcal{C}^{t_5}),M}$ and 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 and deletion of t_1. 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

Algorithm 5: TopDown

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

```
1  S^t ← ∅;
2  foreach M ⊆ M do
3      foreach C ∈ C^t do
4          C.pruned ← false;
5          C.inAnces ← false;
6      Q ← ∅; Q.enqueue(⊤);
7      while not Q.empty() do
8          C ← Q.dequeue();
9          foreach t' ∈ μ_{C,M} do
10             if t ≺_M t' then
11                 Dominated(t', C);
12             else if t' ≺_M t then
13                 Dominates(t', C, M);
14         if not C.pruned then
15             S^t ← S^t ∪ {(C, M)};
16             if not C.inAnces then
17                 μ_{C,M}.insert(t);
18         EnqueueChindren(C);
19 R ← R ∪ {t};
20 return S^t;
```

Procedure: Dominates (t', C, M)

```
1  μ_{C,M}.delete(t');
2  foreach C' ∈ CH^{t'}_C − C^t do
3      stored ← false;
4      foreach C'' ∈ A^{t'}_{C'} − C^t do
5          if t' ∈ μ_{C'',M} then
6              stored ← true;
7              break;
8      if not stored then
9          μ_{C',M}.insert(t');
```

Procedure: Dominated (t', C)

```
1  C.pruned ← true;
2  foreach C' ∈ C^{t,t'} do
3      C'.pruned ← true;
```

Procedure: EnqueueChildren (C)

```
1  foreach C' ∈ CH^t_C do
2      if not C.pruned then
3          C'.inAnces ← true;
4      if not Q.contains(C') then
5          Q.enqueue(C');
```

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 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 Figure 3b. ■

5.2 Algorithm TopDown

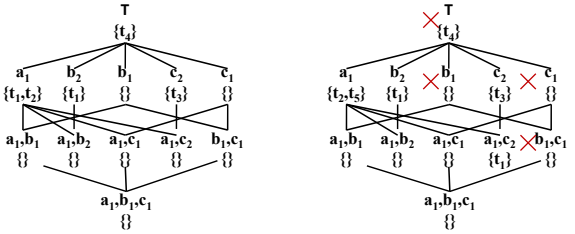
BottomUp stores t for every constraint-measure pair that qualifies t as a contextual skyline tuple. More specifically, if t is stored in $\mu_{C,M}$, then t is also stored in all such $\mu_{C',M}$ that $C' \in \mathcal{D}^t_C$, i.e., descendants of C pertinent to t. For this reason, BottomUp repeatedly compares a new tuple with a previous tuple multiple times. Such repetitive storage of tuples and comparisons increase both the space complexity and the time complexity of the algorithm. On the contrary, TopDown (Algorithm 5) stores a tuple in $\mu_{C,M}$ only if C is a maximal skyline constraint of the tuple. The concept of maximal skyline constraint is defined below.

Definition 9 (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 10 (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. $C' \in \mathcal{SC}^t_M\}$. ■

Example 8. Figure 3b shows, in measure subspace $\{m_1, m_2\}$, t_5 is in the contextual skylines of 4 constraints, i.e., $\mathcal{SC}^{t_5}_{\{m_1,m_2\}}$=$\{\langle a_1, *, *\rangle, \langle a_1, b_1, *\rangle, \langle a_1, *, c_1\rangle, \langle a_1, b_1, c_1\rangle\}$. The maximal skyline constraints of t_5 are $\{\langle a_1, *, *\rangle\}$, 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.



(a) Before t_5 (b) After t_5

Figure 4: Execution of TopDown in Measure Subspace $\{m_1, m_2\}$

Invariant 2. $\forall C \in \mathcal{C}_\mathcal{D}$ 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 $\mathcal{SC}^t_M$. Because of this difference, TopDown traverses the constraints in $\mathcal{C}^t$ in a top-down (instead of bottom-up), breadth-first manner. The traversal starts from Line 6 of Algorithm 5, where the top element $\top$ (instead of the bottom) is inserted into a queue Q. As long as Q 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 9). Various actions are taken, depending on the comparison result:

1) If t is dominated by any tuple 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 another tuple 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., $\mathcal{C}^{t,t''} - \mathcal{C}^{t,t'} \neq \varnothing$. 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' ($\mathcal{MSC}^{t'}_M$) and stores t' in descendants of C if necessary (Lines 2-9 of Dominates). If C has a child C' satisfied by t' but not t, then C' is a skyline constraint of t'. Further, C' is a maximal skyline constraint of t', if there is not such an ancestor of C' that 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 the top-down traversal (Line 18). In the subroutine, each child constraint C' of C is inserted into Q. If t is stored in $\mu_{C,M}$ or any of its ancestors, then $C'.inAnces$ 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 the arrival of t, 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

Algorithm 6: 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 $\quad S^t \leftarrow S^t \cup$ STopDownNode(M);
4 $R \leftarrow R \cup \{t\}$;
5 **return** S^t;

Procedure: STopDownRoot ()

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

Procedure: STopDownNode (M)

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

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 9. We use Figure 4 to explain the execution of TopDown on Table 4, for measure subspace 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. Figure 4a shows $\mu_{C,M}$ beside each constraint C in lattice $\mathcal{C}^{t_5}$ before the arrival of t_5. A tuple is only stored in its corresponding 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 the traversal of $\mathcal{C}^{t_5}$ from the top $\top$. There is only one skyline tuple stored in $\mu_{\top,M}$—t_4. In subspace M, t_5 is dominated by t_4, therefore the content of $\mu_{\top,M}$ 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. Two tuples t_1 and t_2 are stored at $\langle a_1, *, * \rangle$. They do not dominate t_5 in subspace 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, including $\langle a_1, b_1, * \rangle$, $\langle a_1, *, c_1 \rangle$ 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 $\langle a_1, *, c_2 \rangle$. $\langle a_1, b_2, * \rangle$ cannot be a new maximal skyline constraint, since t_1 is already stored at $\langle *, b_2, * \rangle$ which is one of its ancestors. On the contrary, $\langle a_1, *, c_2 \rangle$ becomes a new maximal skyline constraint since it is not subsumed by any existing maximal skyline constraint of t_1. Therefore t_1 is stored at $\langle a_1, *, c_2 \rangle$. The algorithm continues to the end and finds that no tuple is stored at any remaining constraint in $\mathcal{C}^{t_5}$. The content of $\mu_{C,M}$ for relevant constraints after the arrival of t_5 is depicted as Figure 4b. ■

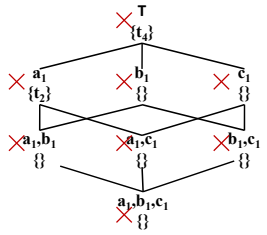
5.3 Sharing across Measure Subspaces

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 earlier in Section 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 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. 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.

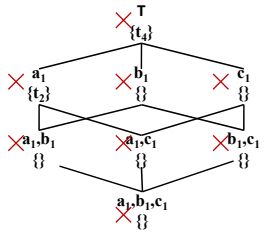
With regard to two tuples t and t', the measure space $\mathcal{M}$ can be partitioned into three disjoint sets $\mathcal{M}^>$, $\mathcal{M}^<$ and $\mathcal{M}^=$ such that 1) $\forall m \in \mathcal{M}^>, t.m > t'.m$; 2) $\forall m \in \mathcal{M}^<, t.m < t'.m$; and 3) $\forall m \in \mathcal{M}^=$,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

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

The gist of STopDown (Algorithm 6) is to compare a new tuple t with current skyline tuples t' in the full measure space $\mathcal{M}$ and, using Proposition 4, identify all subspaces M in which t is dominated by t'. It starts by finding the skyline constraints in the full space $\mathcal{M}$ using STopDownRoot, which is almost the same as 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 those subspaces M where t is dominated by t' are identified. In each such M, constraints in the two tuples' intersection $\mathcal{C}^{t,t'}$ are pruned (Lines 13-16)—indicated by setting values in a two-dimensional matrix $pruned$. After finishing STopDownRoot,



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



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

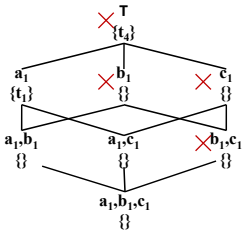
Figure 5: Execution of STopDown in Measure Subspace $\{m_1\}$ (No Comparison Required and No Change Made)

for each subspace M, the constraints C whose corresponding values of $pruned[C][M]$ are 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 13 of STopDownNode), the other is to remove tuples dominated by t and update their maximal skyline constraints (Line 11).

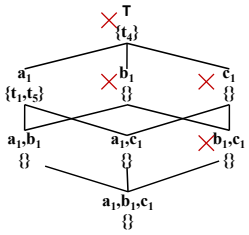
Example 10. We explain the execution of STopDown on Table 4. In the full measure space $\mathcal{M}$=$\{m_1, m_2\}$, STopDown and TopDown work in the same way. Hence, Figure 4 shows $\mu_{C,\mathcal{M}}$ beside each constraint C in lattice $\mathcal{C}^{t_5}$ before and after the arrival of t_5. Comparisons with tuples in the full space also help to prune constraints under subspaces. Consider the top element $\top$ in Figure 4a, where t_4 is stored. The new tuple t_5 is compared with t_4. Based on the comparison result, $\mathcal{M}^{>}$=$\varnothing$, $\mathcal{M}^{<}$=$\{m_1, m_2\}$ and $\mathcal{M}^{=}$=$\varnothing$, since t_5 has smaller values than t_4 on both measure attributes. By Proposition 4, t_5 is dominated by t_4 in subspaces $\{m_1\}$ and $\{m_2\}$. Hence, all the 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 $\mathcal{M}^{<}$=$\varnothing$, 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 $\mathcal{M}^{=}$=$\varnothing$. It follows that 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 $\mathcal{M}$, the algorithm continues with each measure subspace. In subspace $\{m_1\}$, all constraints of $\mathcal{C}^{t_5}$ are pruned. Hence, t_5 has no skyline constraint and nothing further needs to be done. Figure 5 depicts $\mu_{C,\{m_1\}}$ for all constraints in $\mathcal{C}^{t_5}$ before and after the arrival of t_5. For subspace $\{m_2\}$, Figure 6a depicts $\mu_{C,\{m_2\}}$ for all constraints in $\mathcal{C}^{t_5}$ before the arrival of t_5. Based on the aforementioned analysis, 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 the 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 shown in Figure 6b. Note that TopDown would have compared t_5 with other tuples seven times in total, 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 instead of seven 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 before and after encountering any tuple. We omit the proof which is largely the same as the proof for TopDown. We note that the essential difference between STopDown and TopDown is the pruning and 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.



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



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

Figure 6: Execution of STopDown in Measure Subspace $\{m_2\}$

BottomUp is extended to SBottomUp, in a way similar to how TopDown is extended to STopDown. 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 guaranteed by SBottomUp before and after encountering any tuple. Its proof is similar to the proof of the same invariant for BottomUp. Due to space limitations, we omit further discussion of SBottomUp.

6. EXPERIMENTS

6.1 Datasets

The experiments were conducted on two real-world datasets—the NBA dataset and the weather dataset. In presenting experimental results, we focus mostly on the NBA dataset due to space limitations; results on the weather dataset exhibit very similar trends.

NBA 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.

Weather Dataset[5] This dataset has more than 7.8 million daily weather forecast records collected from 5,365 locations in the four countries of the United Kingdom, the Isle of Man and the Channel Islands, from December 2011 to November 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.

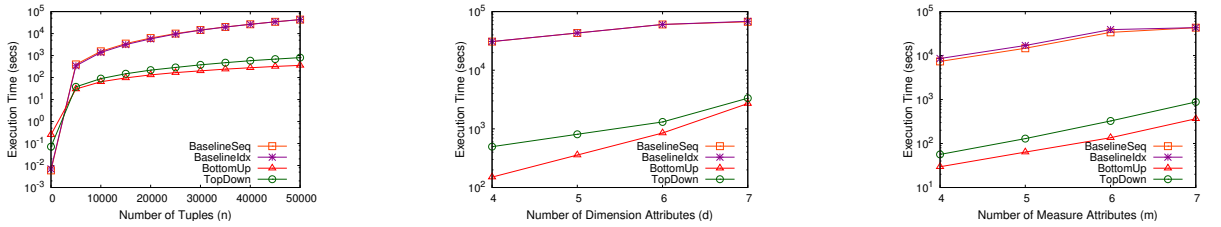
6.2 Experiment Setup

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 space limit on the heap size of Java Virtual Machine (JVM) was set to 16 GB.

Methods Compared We investigated the performance of the two baseline algorithms BaselineSeq and BaselineIdx from Section 4, as well as the four algorithms BottomUp, TopDown, SBottomUp, and STopDown from Section 5. We compared these algorithms in terms of both execution time and memory consumption.

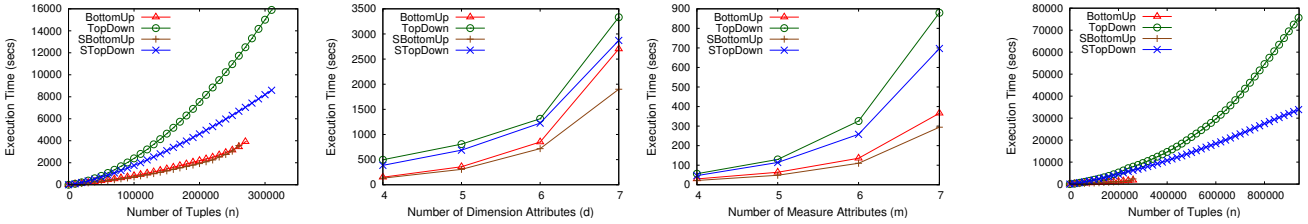
Parameters We ran our experiments under combinations of five parameters, which are number of dimension attributes (d), number of measure attributes (m), number of tuples (n), maximum number of bound dimension attributes ($\hat{d}$) and maximum number of measure attributes allowed in measure subspaces ($\hat{m}$). Dimension/measure spaces considered for different values of d and m are listed in Tables 5 and 6.

[5] 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 7: Execution Time of BaselineSeq, BaselineIdx, BottomUp and TopDown on the NBA dataset



(a) Varying n, d=5, m=7 (b) Varying d, n=50,000, m=7 (c) Varying m, n=50,000, d=5

Figure 8: Execution Time of BottomUp, TopDown, SBottomUp and STopDown on the NBA dataset

Figure 9: Execution Time of BottomUp/TopDown/SBottomUp/STopDown on Weather Dataset, Varying n, d=5, m=7

d	dimension space $\mathcal{D}$
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 5: 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 6: Measure Spaces for Different Values of m

In particular dimension/measure spaces (corresponding to d/m values), experiments were done for varying $\hat{d}$ and $\hat{m}$ values. The parameters $\hat{d}$ and $\hat{m}$ are for avoiding trivial facts. 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. For instance, if d=5 and m=4, the dimension and measure spaces are {player, season, month, team, opp_team} and {points, rebounds, assists, blocks}, respectively. If $\hat{d}$=2 and $\hat{m}$=3, we consider all constraints with at most 2 bound dimension attributes and all measure subspaces with at most 3 measure attributes.

In all experiments in this section, the value of $\hat{d}$ is 4 and $\hat{m}$ is set as $\hat{m}$=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 7, we further study how prominence of facts varies by $\hat{d}$ and $\hat{m}$ values.

6.3 Results of Memory-Based Implementation

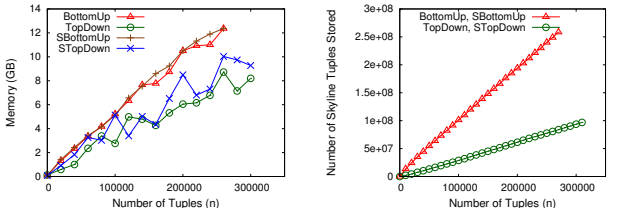
Figure 7 compares the execution times (in logarithmic scale) of BaselineSeq, BaselineIdx, BottomUp and TopDown on the NBA dataset. Figure 7a shows how the cumulative execution times of these methods increase by n, the number of tuples encountered. (The first n tuples from the NBA dataset by their timestamps are used.) The values of other parameters are d=5 and m=7. Figure 7b shows their execution times under varying d, given n=50,000 and m=7. Figure 7c shows their execution times under varying m, for n=50,000 and d=5. The figures demonstrate that both BottomUp and TopDown substantially outperformed the baselines by orders of magnitude. The relative performance gap increased by n. Furthermore, Figure 7b and Figure 7c 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 8 uses the same configurations in Figure 7 to compare BottomUp, TopDown, SBottomUp and STopDown. We make the following observations on the results. First, the execution times of all four algorithms exhibited linear growth with respect to n and superlinear growth with respect to d and m, matching the observations from Figure 7.

Second, in Figure 8a, 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 9), 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. The difference in the sizes of consumed memory by these two categories of algorithms is shown in Figure 10a.[6] 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 10b, 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 exactly the same skyline tuple materialization scheme. Correspondingly BottomUp and SBottomUp store tuples in the same way.

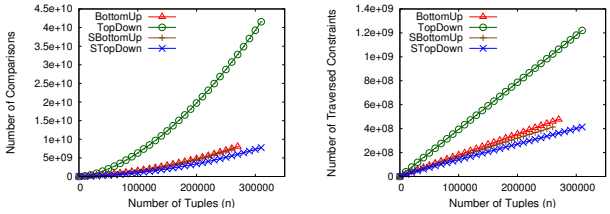
Third, in terms of execution time, TopDown/STopDown were outperformed by BottomUp/SBottomUp. The reason is, if a new tuple t dominates a previous tuple t' in constraint C and measure subspace M, TopDown/STopDown must update $\mathcal{MSC}^{t'}_M$. On the contrary, BottomUp/SBottomUp do not carry this 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.

[6] The consumed memory in Figure 10a was measured before processing each tuple. It thus does not account for the maximal consumed memory during processing a tuple, in which all available memory may be exhausted.



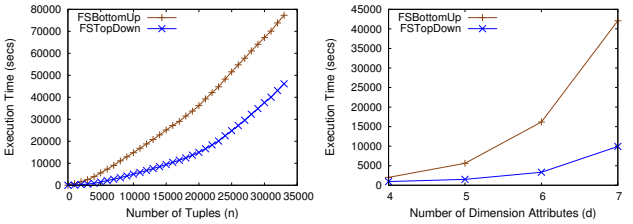
(a) Size of Consumed Memory (b) Number of Skyline Tuples Stored

Figure 10: Memory Consumption by BottomUp, TopDown, SBottomUp and STopDown on the NBA Dataset, Varying n, d=5, m=7



(a) Number of Comparisons (b) Number of Traversed Constraints

Figure 11: Work Done by BottomUp, TopDown, SBottomUp and STopDown on the NBA Dataset, Varying n, d=5, m=7



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

Figure 12: Execution Time of FSBottomUp and FSTopDown on the NBA Dataset

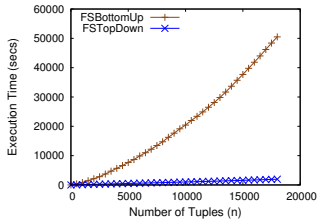


Figure 13: Execution Time of FSBottomUp and FSTopDown on the Weather Dataset, Varying n, d=5, m=7

Finally, SBottomUp/STopDown are faster than BottomUp/TopDown, thanks to the sharing of computation across measure subspaces. From Figures 8b and 8c, we observe that the benefit of sharing computation became more prominent with the increase of both d and m. Figure 11 further presents the amount of work done by these algorithms, in terms of compared tuples (Figure 11a) and traversed constraints (Figure 11b). 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 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.

6.4 Results of File-Based Implementation

The memory-based implementation of all aforementioned algorithms consumes large amount of memory to store skyline tuples for all combinations of constraints and measure subspaces. As new tuples keep coming, sooner or later, all algorithms ultimately lead to memory overflow. To address this, we investigated file-based implementations of STopDown and SBottomUp, which are called FSTopDown and FSBottomUp, respectively. In these 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 constraint-measure pair is visited. Insertion and deletion on $\mu_{C,M}$ are then performed on the buffer. When an algorithm finishes processing the constraint-measure pair, the file is overwritten by the content of the buffer.

Figure 12 uses the same configurations in Figures 7 and 8 to compare the execution times of FSBottomUp and FSTopDown on the NBA dataset. Figure 13 further compares them on the weather dataset. The figures show that FSTopDown outperformed FSBottomUp by multiple times. Even for only n=5,000, the performance gap between them was already clear in Figures 12b and 12c. The reason is as follows. In file-based implementation, while traversing a constraint-measure pair (C, M), a file-read operation occurs if $\mu_{C,M}$ is non-empty. Since FSTopDown stores significantly fewer tuples than FSBottomUp (as Figure 10 shows), FSTopDown is much 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 much 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 in file-based implementations.

7. CASE STUDY

A newly arrived tuple may be in the contextual skylines of many constraint-measure pairs. For instance, the new tuple t_7 in Example 1 belongs to 196 contextual skylines (of course partly because the table is very small and most contexts contain only t_7). Reporting all such situational facts can overwhelm users and make important facts harder to spot. It is thus crucial to report truly *prominent* facts. Intuitively, prominent facts should be rare. We thus 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 corresponding context.[7] As an example, consider two constraint-measure pairs from Example 1: (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 five tuples, among which t_2 and t_7 are in the skyline in subspace M_1. Hence, the prominence of (C_1, M_1) is $5/2$. Similarly the prominence of (C_2, M_2) is $3/2$, which makes (C_1, M_1) 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 above (or equal to) 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 1. From the 196 facts in S^{t_7}, the

[7] The notion of prominence is a natural extension of *uniqueness* introduced in [10] that makes it relative to the size of context.

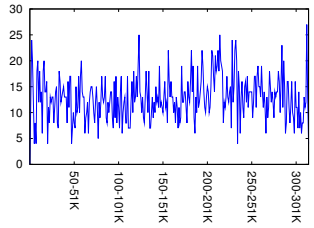
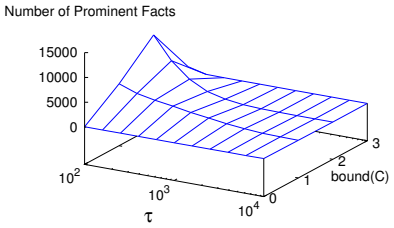
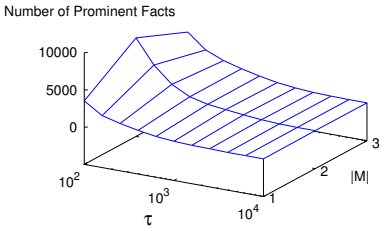


Figure 14: Number of Prominent Facts for Each 1,000 Tuples, τ=10^3



(a) By Number of Bound Dimension Attributes



(b) By Dimensionality of Measure Subspaces

Figure 15: Distribution of Prominent Facts, Varying τ

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}). If $\tau > 3$, then there are no prominent facts for t_7. 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 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. Note that these facts 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. This is the highest score made by any Trail Blazers in NBA history.

Figures 14 and 15 help us further understand the prominent facts from this experiment at the macro-level. Specifically, Figure 14 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 14 mostly oscillate between 5 and 25. Consider the number of tuples and the huge number of possible constraint-measure pairs, these prominent facts are truly selective. One might expect a downward trend in Figure 14. The reason why it does not occur is the constant formulation of new contexts. Each year, a new NBA regular season commences and some new players start to play in the NBA league. 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 streak.

Figure 15a shows the distribution of prominent facts by the number of bound dimension attributes in constraint for varying τ in $[10^2, 10^4]$. Similarly, Figure 15b shows the distribution by the dimensionality of measure subspace. We observed that there are fewer prominent facts with 0 and 3 bound attributes (out of d=5 dimension attributes) than those with 1 and 2 bound attributes, and there are fewer prominent facts in measure subspaces with 1 and 3 attributes than those with 2 attributes. The analysis of these observations is as follows. **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 streaks 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.

8. CONCLUSION

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 discovery of prominent facts. We also used a simple prominence measure to rank the discovered facts. Extensive experiments over two real datasets validated the effectiveness and efficiency of the techniques. This is our first step towards general fact finding for realizing computational journalism. Going forward we plan to explore several directions, including generalizing the solution for other forms of facts, allowing deletion and update of data, and narrating facts in natural-language text.

9. REFERENCES

[1] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

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

[3] N. Best. Hirdt enjoying long run as stats guru. http://www.newsday.com/sports/columnists/neil-best/hirdt-enjoying-long-run-as-stats-guru-1.3174737, September 2011. Accessed: February 8, 2013.

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

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

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

[7] X. Jiang, C. Li, P. Luo, M. Wang, and Y. Yu. Prominent streak discovery in sequence data. In *KDD*, pages 1280–1288, 2011.

[8] 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.

[9] T. Wu, D. Xin, Q. Mei, and J. Han. Promotion analysis in multi-dimensional space. *Proc. VLDB Endow.*, 2(1):109–120, 2009.

[10] Y. Wu, P. K. Agarwal, C. Li, J. Yang, and C. Yu. On "one of the few" objects. In *KDD*, pages 1487–1495, 2012.

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

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