Optimization of Skyline queries in dynamic contexts

Karim Alami

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR DE

L'UNIVERSITÉ DE BORDEAUX

.....

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE

SPÉCIALITÉ : Informatique

Par **Karim ALAMI**

Optimization of Skyline queries in dynamic contexts

.....

Sous la direction de Mcf. Sofian MAABOUT

Soutenue le 09 Octobre 2020

Membres du jury :

M. Guillaume BLIN	Professeur à l'Université de Bordeaux	Président du jury
Mme Karine ZEITOUNI	Professeur à l'Université Versailles St-Quentin	Rapportrice
M. Farouk TOUMANI	Professeur à l'Université Clermont Auvergne	Rapporteur
Mme Nicole BIDOIT	Professeur à l'Université Paris-Sud	Examinatrice
M. Sofian MAABOUT	Mcf. à l'Université de Bordeaux	Directeur de thèse

Titre

Optimisation des requêtes de préférence Skyline dans des contextes dynamiques.

Résumé

Les requêtes de préférence sont des outils intéressants pour traiter les données. Elles permettent par exemple de récupérer à partir d'un ensemble de données, un sous ensemble qui résume les données en entrée. Dans cette thèse, nous abordons principalement l'optimisation des requêtes Skyline dans des contextes dynamiques. Dans un premier temps, nous abordons la maintenance incrémentale de la structure d'indexation NSC qui a été démontrée efficace pour répondre aux requêtes Skyline dans un contexte statique. Plus précisément, nous abordons (i) le cas des données dynamiques, quand les tuples sont insérés ou supprimés à tout moment, et (ii) le cas des données en flux quand les tuples sont ajoutés et supprimés à des intervalles de temps spécifiques. Dans un deuxième temps, nous abordons l'optimisation des requêtes Skyline en présence d'ordres dynamiques, c'est-à-dire que certains ou tous les attributs de l'ensemble de données sont nominaux et que chaque utilisateur exprime son propre ordre partiel sur le domaine de ces attributs. Dans ce cas, nous proposons des algorithmes parallèles qui décomposent une requête soumise en un ensemble de sous-requêtes et traitent chacune indépendamment.

Mots-clés

Requêtes Skyline, Données dynamiques, Optimization des requêtes, Structures d'indexation.

Title

Optimization of Skyline queries in dynamic contexts.

Abstract

Preference queries are interesting tools to compute small representatives of datasets. In this thesis, we mainly focus on the optimization of Skyline queries in dynamic contexts. In a first part, we address the incremental maintenance of the multidimensional indexing structure NSC which has been shown efficient for answering skyline queries in a static context. More precisely, we address (i) the case of dynamic data, i.e. tuples are inserted or deleted at any time, and (ii) the case of streaming data, i.e. tuples are appended and discarded at specific interval of time. In a second part, we address the optimization of skyline queries in presence of dynamic orders, i.e, some or all attributes of the dataset are nominal and each user expresses his/her own partial order on these attributes' domain. In that case, we propose scalable parallel algorithms that decompose an issued query into a set of sub-queries and process each sub-query independently.

Keywords

Skyline queries, Dynamic data, Query optimization, Index structures.

Unité de recherche

Université de Bordeaux, CNRS, LaBRI, UMR 5800, F-33400 Talence, France

Acknowledgements

Firstly, I would like to thank my supervisor Sofian Maabout for his immense support and guidance through the three years of Ph.D. His research vision and quality of work have had an invaluable impact on my research work. I would like to thank BKB Team as well. The discussions and presentations during the meetings have always been fruitful.

I would like to thank Karine Zeitouni and Farouk Toumani for reviewing my thesis manuscript. I thank as well Nicole Bidoit and Guillaume Blin for being part of my jury.

My sincere gratitude goes to colleagues and professors in LaBRI. I would like to name Carole Blanc and Marie Beurton-Aimar. PhD students Trang, Karim, Paul, Jason, Chahrazad, Attila and many others who made this journey exceptional. I would like to thank as well the administrative team in LaBRI, Maïté Labrousse, Cathy Roubineau and Sylvaine Granier for their help and support.

I cannot be grateful enough to my parents who did everything for my success, my brothers Hicham and Amine, and my sister Maryam.

Résumé

Les requêtes de préférence sont des outils intéressants pour traiter les données. Elles permettent par exemple de récupérer à partir d'un ensemble de données, un sous ensemble qui *résume* les données en entrée, ou bien d'ordonner les données selon les préférences de l'utilisateur. Ces requêtes sont utilisées dans plusieurs contextes. Par exemple, filtrer les données pour ne garder que les tuples intéressants vis à vis de l'utilisateur. Aussi, elles sont utilisées dans les systèmes de recommandation pour aider l'utilisateur à faire son choix en proposant un sous ensemble de taille limitée. Un cas réel serait un système de réservation de vol où étant donnée les préférences de l'utilisateur, le système propose un ensemble restreint de vols à l'utilisateur.

Les requêtes *Skyline* sont une classe des requêtes de préférence. Elles retournent un sous ensemble de données qui constituent les "meilleurs" éléments d'un ensemble de données. Elles se basent sur le principe de la domination. Soit deux tuples t et t', t est dit dominé par t' si t' est meilleur ou égal à t sur tous les attributs, et strictement meilleur sur au moins un attribut. Le skyline est alors l'ensemble des tuples non dominés. Le concept de la requête skyline est une adaptation en bases de données de la frontière de Pareto. Celle-ci représente en études économiques, un ensemble d'états qui ne permettent plus aucune optimisation. Considérons la figure 1 qui représente un ensemble de points décrits par deux attributs $f1$ et $f2$. Supposons que les plus petites valeurs sur ces attributs sont préférées. Ainsi l'ensemble skyline est l'ensemble de points reliés par la ligne rouge. Le point C ne fait pas partie du skyline car les points B et A le dominent. Notons qu'un seul des deux points est suffisant pour que C ne fasse pas partie du skyline.

Etant donné un ensemble de données T sur un ensemble d'attributs $\mathcal{D}$, l'évaluation d'une requête skyline est de $O(|T|^2 \cdot |\mathcal{D}|)$ en temps d'exécution. Notre travail de recherche vise à étudier et à optimiser l'évaluation de ces requêtes. En base de données, il existe trois moyens principaux pour répondre à une requête: (i) parcourir tout l'ensemble de donnée en entrée sans avoir aucune information préalable sur la localisation des données, (ii) précalcul ou matérialisation des résultats, c'est à dire, sauvegarder en mémoire les résultats à des requêtes émises auparavant pour répondre à des requêtes futures, et (iii) utiliser des structures d'indexation qui permettent une recherche rapide des informations.

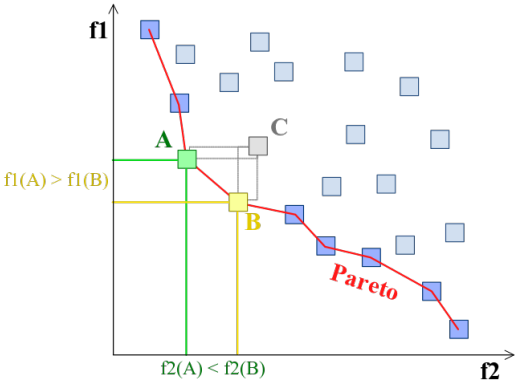


Figure 1: Frontière de Pareto

Nous avons adopté ces trois méthodes pour répondre aux problématiques traitées durant la thèse.

Dans un premier temps, nous considérons des données totalement ordonnées, c'est à dire, il existe un ordre total sur le domaine des attributs, et nous abordons l'optimisation de requêtes skylines multidimensionnelles. Dans la littérature, il existe principalement des solutions qui calculent le skyline de zéro ou matérialisent le résultat. Récemment, la structure d'indexation NSC a été proposée et a été montrée plus efficace que les solutions de l'état de l'art. NSC est de la forme clé valeur où la clé représente un sous ensemble d'attributs et la valeur est un ensemble de tuples. Quand une requête skyline par rapport à un sous ensemble d'attributs est émise, la procédure d'évaluation des requêtes balaye la structure NSC pour constituer le résultat. Cependant NSC a été conçue en supposant que les données sont statiques. Hors, dans les applications du monde réel, les données changent constamment. Ainsi, nous étudions la maintenance incrémentale de la structure d'indexation NSC. Nous considérons deux types de données: (i) données dynamiques où les tuples peuvent être inserés ou supprimés à tout moment, et (ii) données en flux où les tuples sont ajoutés à intervalle prédéfini.

Dans un contexte de données dynamiques, c'est à dire, des tuples peuvent être inserés ou supprimés à tout moment. Nous avons identifié que la suppression est plus difficile que l'insertion. En effet, il est possible de récupérer l'information qu'un tuple t est dominée par NSC mais pas la source de cette domination, c'est à dire, le ou les tuples qui dominent t. Ainsi, quand des suppressions se produisent au niveau de l'ensemble de données, il est nécessaire de reconstruire NSC. Nous proposons alors une modification de la structure pour pallier à ce problème qui est très couteux. Pour tout tuple, nous ajoutons dans NSC l'information du nombre de tuples qui le dominent par rapport à un sous ensemble d'attributs. Cette approche permet un bon compromis entre le temps de maintenance et l'espace mémoire supplémentaire utilisé. Nous montrons des expériences

réalisées pour évaluer les performances de NSC par rapport à ses compétiteurs où nous considérons plusieurs ensembles de données réels et synthétiques avec des configurations allant jusqu'à 1 million de tuples et 20 attributs. En général, nous montrons que NSC est jusqu'à 100 fois plus rapides que les méthodes qui calculent le skyline de zéro. Aussi, NSC utilise moins de mémoire (environ 16 fois moins la taille des données). Enfin, il admet une maintenance incrémentale. Par exemple, en ajoutant 10% de la taille des données de départ, le temps de maintenance est aussi 10% du temps de construction du nouvel ensemble de données de zéro. Aussi, nous montrons que pour certaines configurations, le temps moyen pour mettre à jour la structure après une suppression d'un tuple est 1000 fois moins que le temps de reconstruire toute la structure de zéro.

Dans un contexte de données en streaming, c'est à dire, des données qui s'ajoutent chaque θ unité de temps et des requêtes qui considèrent les données insérées dans un intervalle de temps de taille ω. Nous avons conçu un système de gestion des données entrantes *MSSD*. Ce système gère les données par lot. Nous stockons les données entrantes dans une mémoire tampon durant intervalle de temps de taille k. Puis nous mettons à jour la structure *NSCt* qui est une adaptation de *NSC* pour ce contexte. Cette stratégie permet un compromis entre la précision des réponses aux requêtes et la capacité du système à y répondre. En effet, la requête de l'utilisateur ne considère que les données arrivées à l'instant de la dernière maintenance. Par contre, plus la durée de mise en tampon est grande, plus long est l'intervalle de temps qu'a l'utilisateur pour exécuter ses requêtes. Nous montrons empiriquement qu'en adoptant le système de traitement par lot avec NSCt, un utilisateur peut soumettre plusieurs requêtes pendant l'intervalle de lot. Aussi, l'esapce mémoire utilisé est jusqu'à 100 fois plus petit que l'espace mémoire utilisé par une méthode qui matérialise les résultats. Enfin, nous présentons une expérience sur un flux de tweets collectés en temps réel par notre framework. Nous montrons que dans ce cas aussi, notre solution permet de filtrer les tweets des personnes influentes en un temps plus intéressant que celui des autres approches.

Après avoir démontré théoriquement et empiriquement que NSC est une structure de données rapide et fiable pour l'évaluation des requêtes skyline multidimensionnelles, nous investiguons l'optimisation des requêtes de minimisation de regret à travers NSC. Ces requêtes ont été proposées comme alternatives aux requêtes skyline et requête Top-K. En effet les requêtes skyline ne permettent pas la maîtrise de la taille du résultat. Et les requêtes Top-K requièrent une fonction de score. Les requêtes de minimisation de regret ont comme entrée un ensemble de données T, un entier K et une famille de fonctions $\mathcal{F}$. Le résultat est un ensemble S de taille K qui minimise une erreur (regret) ε. Cette erreur est calculée de plusieurs façons dans la littérature. En général, elle représente

la différence entre le meilleur score obtenu par l'ensemble S comparé à celui obtenu par l'ensemble en entrée T par rapport à la famille de fonction $\mathcal{F}$. Il a été démontré précédemment qu'il est suffisant de considérer le skyline de T, $Sky(T)$, au lieu de tout T pour calculer l'ensemble S. Dans ce manuscrit, nous investiguons l'impact d'autres ensembles candidats pour le calcul de l'ensemble de regret minimum comme le Top-K des skylines fréquents (Top-KF). Globalement, nous montrons que le résultat de la requête Top-KF est un bon ensemble candidat pour calculer un ensemble à regret minimum car (i) il est rapide à calculer par NSC et (ii) il permet de trouver un ensemble résultat de regret intéressant.

Dans un deuxième temps, nous considérons que les domaines de certains attributs sont partiellement ordonnés, par exemple, un attribut "compagnie aérienne". A prioiri, il n'existe pas d'ordre spécifique entre les compagnies. Dans ce cas, chaque utlisateur définit un ordre partiel (préférence) $\mathcal{R}$ sur les valeurs des attributs. Dans la littérature, les travaux ayant abordé ce problème ont proposé soit des solutions qui calculent le skyline de zéro soit des approches basées sur la matérialisation des résultats. Cependant, ces solutions sont inefficaces quand le nombre d'attributs à ordre partiel ou les domaines des attributs à ordre partiel grandissent. Dans ce manuscrit, nous proposons une solution qui décompose une requête en plusieurs sous-requêtes. En effet, un utilisateur définit un ordre partiel $\mathcal{R}$ sur un attribut. Cet ordre partiel est un ensemble de paires (x,y) tel que la valeur x est préféré à la valeur y. Notre méthode consiste à décomposer une requête en plusieurs sous-requêtes tel que chaque sous requête considère une seule paire $(x,y) \in \mathcal{R}$ seulement. Le résultat final n'est que l'union des résultats des sous requêtes. Du fait que ces sous-requêtes sont indépendantes, nous les évaluons en parallèle. Aussi, du fait que le nombre de sous-requêtes est limité, les résultats de ces sous-requêtes peuvent être matérialisées pour optimiser toutes les requêtes possibles. En outre, nous introduisons le problème de sélection d'un sous ensemble de sous-requêtes à matérialiser dans le but d'optimiser un ensemble de requêtes donné, en prenant en considération la contrainte d'espace mémoire disponible. Nous évaluons empiriquement nos propositions pour valider les propriétés théoriques. En général, nous montrons que notre approche est plus efficace quand la taille des données et la taille du domaine des attributs à ordre partiel grandissent. Aussi, nous montrons que le temps d'évaluation des requêtes par notre méthode décroit linéairement avec le nombre de processeurs affectés au calcul.

Abstract

Preference queries are interesting tools to compute small representatives of datasets or to rank tuples based on the users' preferences. In this thesis, we mainly focus on the optimization of *Skyline* queries, a special class of preference queries, in dynamic contexts. In a first part, we address the incremental maintenance of the multidimensional indexing structure NSC which has been shown efficient for answering skyline queries in a static context. More precisely, we address (i) the case of dynamic data, i.e. tuples are inserted or deleted at any time, and (ii) the case of streaming data, i.e. tuples are appended only, and discarded after a specific interval of time. In case of dynamic data, we redesign the structure and propose procedures to handle efficiently both insertions and deletions. In case of streaming data, we propose MSSD a data pipeline which operates in batch mode, and maintains NSCt a variation of NSC. We moreover investigate the optimization of regret minimization queries through NSC. In a second part, we address the optimization of skyline queries in presence of dynamic orders, i.e, some or all attributes of the dataset are nominal and each user expresses his/her own partial order on these attributes' domain. In that case, we propose highly scalable parallel algorithms that decompose an issued query into a set of sub-queries and process each sub-query independently. In a further step for optimization, we propose the partial materialization of sub-queries and introduce the problem of cost-driven sub-queries selection.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Nowadays, data is driving decisions and is bringing value to businesses. However, often, the amount of data and the multiple criterion make it hard to extract valuable insight directly from the input data. It becomes then imperative to have tools that filter useful data and compute small interesting representatives. *Preference queries*, for instance, are tools that allow users to extract and rank data with respect to their preferences. One concrete example for preference queries implementation is the flight booking platform *Skyscanner*. Users are given a small set of flights considered the "best" with respect to their travel information. They can then make their decision based on this set of flights.

Since the introduction of preference queries by the database community in the 90's, two main variants have been extensively studied and expanded to many applications: *Top-K* queries [1] and *Skyline* queries [2]. Both have the same objective, namely retrieving the *best* tuples, however, they diverge in their semantics. Top-K queries are combined with utility (scoring) functions that rank tuples, and return bounded results, i.e. a set of K tuples. While skyline queries depend on *order* relationship and dominance without relying on any utility function. Skyline query result contains only those tuples that are not worse than any other. Concretely, given a set of attributes $\mathcal{D}$, and two tuples t and t' sharing these attributes. We say that t dominates t' if and only if t is *better* or equal than t' on all attributes in $\mathcal{D}$ and strictly better on at least one attribute. The skyline set is then the set of non dominated tuples. By contrast to Top-K queries, the result is not bounded.

Example 1. *Consider the dataset depicted in Table 1. It represents a set of flights connecting Paris to Singapore on March* 5^{th}*. Flights are described by their price, duration and number of stops.*

Top-K queries rely on a utility function. Let us consider the monotonic utility function $f(t) = t[Price] * 10 + t[Duration] * 5 + t[\#Stops] * 100$ *such that* t *is a tuple representing a flight. Then, the utility of flight* t_1 *is* $f(t_1) = 321 * 10 + 15.25 * 5 + 1 * 100 = 3386.25$ *and that of flight* t_2 *is* $f(t_2) = 393 * 10 + 14.10 * 5 + 1 * 100 = 4100.5$*. Considering this utility function* f*, Top-3 flights is composed of flights* t_1*,* t_2 *and* t_3*.*

Skyline queries rely on order relationship. The order on numerical attributes is the natural order over $\mathbb{R}$*, i.e.* $<$ *or* $>$*.*

Tuple	Price	Duration (in hours)	# of Stops
t_1	321	15.25	1
t_2	393	14.10	1
t_3	461	12.50	0
t_4	392	14.90	1
t_5	378	15.75	1
t_6	297	20.90	2
t_7	327	19.10	1
t_8	400	16	1
t_9	367	17.80	1
t_{10}	255	23.50	2

Table 1: Flights connecting Paris to Singapore on March 5^{th}

Note that a skyline query is computed with respect the set of attributes the user is interested into. For example, a user flying on budget, is interested into the skyline set with respect to Price and Duration only, which is the set $\{t_1,t_2,t_3,t_4,t_6,t_{10}\}$. *While another user, rich enough, is interested in the skyline with respect to Duration and # of Stops, which is the set* $\{t_3\}$.

In this dissertation, we consider mainly skyline queries [2]. Although they have attracted great attention by the database community, their computation is still challenging. Given a dataset of size n, the time complexity to compute the skyline set is in the worst case $O(n^2)$.

Several works proposed optimization techniques for evaluating skyline queries. Such works can be broadly categorized into three groups. The first group uses indexes such as R-Trees [3, 4]. The second group uses preprocessing such as ranking tuples with respect to (wrt) some utility function in order to prune comparisons [5, 6]. The third group uses partitioning in order to continuously prune dominated tuples [7, 8, 9]. The later group techniques have been shown the most efficient. More details in Chapter 1.

The above techniques are adapted to systems which receive few skyline queries as they mainly compute the skyline from scratch any time a query is issued. They are not however efficient for systems with high skyline queries throughput. To cope with this limitation, [10, 11] proposed the materialization of the skyline query result and proposed techniques to update the materialized result each time the underlying data changes.

Challenges and contributions

In this dissertation, we address the challenges of efficiently answering skyline queries in dynamic contexts. Concretely, in a first part, we address the maintenance of an

Subspace	Skyline
(P,D,S)	$\{t_1,t_2,t_3,t_4,t_6,t_{10}\}$
(P,D)	$\{t_1,t_2,t_3,t_4,t_6,t_{10}\}$
(P,S)	$\{t_1,t_3,t_{10}\}$
(D,S)	$\{t_3\}$
(P)	$\{t_{10}\}$
(D)	$\{t_3\}$
(S)	$\{t_3\}$

Table 2: Subspace skylines

indexing structure NSC upon updates. This structure has been shown efficient for answering multidimensional skyline queries but was designed for static data. In this dissertation, we redesign the structure and propose procedures to deal with (i) dynamic data and (ii) streaming data. In a second part, we consider the case where data have dynamically ordered attributes and users are allowed to express their own preferences on the attributes' domain. We then propose both scalable on-the-fly algorithms and materialization techniques for efficiently evaluating skyline queries.

Multidimensional skyline queries and moving data Consider the dataset in Table 1. Users can issue a skyline query wrt any non empty combination of the three attributes (Price, Duration, # of Stops), e.g. $Sky(Price, Duration)$ or $Sky(\#of\ Stops)$. There exists $2^3-1=7$ possible subspace (subset of attributes) skyline queries where 3 is the number of attributes. Table 2 illustrates all subspace skyline queries wrt the dataset in Table 1

In the literature, works proposed the computation and the materialization of all possible subspace skyline queries in a structure called the *Skycube*, e.g. [12]. This approach ensures the minimum cost for evaluating a skyline query, however it requires a high storage cost. Other works proposed partial materialization of the *Skycube* or dedicated index structures that seek for a reasonable trade-off between the memory cost and the query answering time, e.g, [13, 14, 15]. In previous work [16], the structure NSC has been presented as an index to optimize multidimensional skyline queries. Let $\mathcal{D}$ be a set of attributes and T be a dataset, its main idea consists in comparing every tuple $t \in T$ to all remaining tuples t' in T and summarizing the subspaces where t' dominates t in a pair $\langle X|Y\rangle$. X represents the attributes where t' is *strictly better* than t and Y represents the attributes where t and t' are equal. Now given a subspace $Z \subseteq \mathcal{D}$, a tuple t belongs to $Sky(Z)$, i.e. the skyline over the subspace Z, if and only if there does not exist a pair $\langle X|Y\rangle$ associated to t that *covers* Z, i.e. $Z \subseteq XY$ and $Z \neq Y$. $cover(\langle X|Y\rangle)$ denotes the set of subspaces covered by $\langle X|Y\rangle$. For example, consider a dataset with attributes A, B and C. The pair $\langle AB|C\rangle$ covers the subspaces $\{A,B,AB,AC,BC,ABC\}$. The time complexity

of NSC is quadratic wrt the size of the dataset as well as its space complexity. However, not every pair is kept. Let $Pairs(t)$ be the set of pairs associated to t. This set can be minimized by computing a subset $Q \subseteq Pairs(t)$ such that $cover(Q) = cover(Pairs(t))$, i.e. the set of subspaces covered by $Pairs(t)$ are covered by Q as well. Q is considered an *equivalent* subset of $Pairs(t)$, $Q \equiv Pairs(t)$. The minimization problem is NP-Hard and a polynomial greedy approximate algorithm has been proposed. Experiments in [16] have shown the proposed structure NSC to be the most efficient wrt both construction and query answering time, and space consumption (cf. Chapter 2).

However, NSC's incremental maintenance has been left an open question. In This dissertation, we address its incremental maintenance in case of dynamic data, i.e., tuples are inserted/deleted at any time. Regarding insertions, we provide a procedure as well as an incremental technique for the minimization of the set of pairs. Regarding deletions, we propose a slight modification of the structure that allows the identification of *impacted* tuples by a deletion, i.e., tuples that need their respective set of pairs to be rebuilt. This enables a partial rebuild of the structure rather than a rebuild from scratch. We show trough extensive experiments that these modifications do not alter NSC's query answering performance. Moreover, we show that the maintenance cost is low. Overall, we show that (i) skyline query evaluation time is up to 100 times faster than state of the art skyline algorithm, (ii) memory usage is low (about 16 times less than input data size), and (iii) the proposed maintenance procedures are effective, e.g. adding 10% of the overall size of initial data requires 10% of the time to build the NSC from scratch (cf. Chapter 3).

In a second time, we address NSC's incremental maintenance in case of streaming data. The proposals in Chapter 3 (dynamic data) are not suited to streaming context because the maintenance latency is variable and uncontrollable. Indeed, some updates may take few milliseconds while others may last several minutes (cf. Section 3.4). Hence, we propose a buffer-based system MSSD which processes data in batch mode. MSSD is composed of (i) a data buffer, (ii) a main dataset, and (iii) NSCt a variation of NSC to deal with streaming data. This system balances the maintenance frequency with the query answering performance. One may choose a longer buffering window if he/she is interested in evaluating a large number of queries (cf Chapter 4). We show that by adopting the batch processing system with NSCt, a user can submit a large number of queries during the batch interval. Also, the memory space used is up to 100 times smaller than that used by a method which materializes the results. Finally, we carried out experiments on tweets collected by our framework. We show that in this case too, our solution filters the tweets of influential people faster than other approaches.

Finally, we leverage NSC to optimize the computation of regret minimization queries

proposed by [17]. Given a dataset T, a family of linear scoring functions $\mathcal{L}$ and an integer K. Let $f \in \mathcal{L}$ and $f_1(T)$ be the best score by considering tuples in T. The regret minimization query aims to compute a subset $S \subset T$ of size K such that for every function $f \in \mathcal{L}$, the difference between $f_1(S)$ and $f_1(T)$ is minimum. This difference is called the regret ratio. In short, the regret ratio represents how far is the user's best choice within S from the user's best choice within T. [17] proposed these queries to avoid the limitations of both skyline queries and Top-K queries, i.e. computing a bounded result without requiring a scoring function from the user. [18] proved the NP-Hardness of computing such set. In this dissertation, we investigate the improvement provided by a skyline related query, namely Top-K frequent skyline query, to computing regret minimization sets. We principally investigate the speedup of regret minimization queries when they are computed on top of the result of a Top-K frequent skyline query rather than the whole dataset. We explore this mainly because regret minimization queries are (i) time-consuming and (ii) Top-K frequent skyline queries are optimized by NSC. The empirical results show that Top-K frequent skyline query provide interesting execution time and regret ratio.

Skyline queries in presence of dynamic and partial orders Consider the dataset in Table 3. Users may want to include the attribute "Airline company" into their skyline query. However, there does not exist a predefined order over the attribute's domain. Hence, users express their preferences over companies, e.g., one user may prefer *Finnair* and *Thai* over the remaining companies. While another user may prefer *Swiss Airline* over all. Techniques and algorithms for data with static and total orders are not suitable for this configuration. In the literature, there is two major approaches to handle this situation, (i) algorithms which, given a query q, maps a nominal attribute into a set of virtual totally ordered attributes in accordance to the user preference $q.\mathcal{R}$. Then, a *traditional* algorithm, e.g. BSkyTree [7], is processed over the transformed dataset. For example, [19] uses the lattice theorem [20, 21] to transform a partially ordered attribute into a set of totally ordered attributes. (ii) Algorithms that answer the issued query through a set of cached views. For example, [22] adopts a *refinement* strategy. Let q be an issued query and let q' be a cached view then $Ans(q) \subseteq Ans(q')$ if $q'.\mathcal{R} \subseteq q.\mathcal{R}$. We say that q is a *refinement* of q'. More details about related work in Chapter 1. In this dissertation, we propose a *decomposition* technique. It consists in decomposing a query q into a set of sub-queries Q. Each sub-query can be processed independently. The result of q is simply the union of the results of the sub-queries in Q. Moreover, sub-queries result can be materialized such that further issued queries are optimized. We propose and address a cost-based problem to select the relevant sub-queries to materialize. Experiments show that our proposals

outperform those in the literature (cf Chapter 6). In general, we show that our approach is more efficient when (i) the size of the input data and (ii) the domain size of the nominal attributes grow. Also, we show that our algorithms are scalable.

Tuple	Price	Duration (in hours)	# of Stops	Airline company
t_1	321	15.25	1	Finnair
t_2	393	14.10	1	Lufthansa
t_3	461	12.50	0	Singapore Airlines
t_4	392	14.90	1	Swiss
t_5	378	15.75	1	Thai
t_6	297	20.90	2	XiamenAir
t_7	327	19.10	1	Finnair
t_8	400	16	1	Lufthansa
t_9	367	17.80	1	Eva Air
t_{10}	255	23.50	2	Norwegian

Table 3: Flights connecting Paris to Singapore on March 5^{th} with airline company name

Manuscript organization

We first recall the literature relevant to this thesis in Chapter 1. We summarize two decades of work relative to skyline queries. We detail more the aspects related to our work. In Chapter 2, we present the main definitions and notations used throughout the manuscript, and we recall the structure NSC [16] which is a building block of our work.

In Chapter 3, we present our first contribution: NSC's incremental maintenance in presence of dynamic data. We address both the cases of insertions and deletions. This work has been published in "Information Systems" Journal [23].

In Chapter 4, we address NSC's incremental maintenance in presence of streaming data. We present MSSD, a framework that handles data in batch mode and propose a new design for NSC to cope with this setting. Then we present experiments that assess our proposals performance. This work has been published in "Data and Knowledge Engineering" Journal [24] as well as "DASFAA'19" proceedings [25].

In Chapter 5, we investigate the optimization of regret minimization queries through NSC. Concretely, we evaluate Top-K frequent skyline queries results as candidates sets for regret minimization queries. This work is currently under review.

In Chapter 6, we address the optimization of skyline queries in presence of data with dynamic and partial orders. We provide scalable parallel algorithms and materialization techniques to efficiently process these queries. This work is currently under review.

Finally, we conclude the manuscript by providing perspectives for future work.

Chapter 1

Related work

Here we give a general overview of work related to this dissertation. We give skyline queries the larger share of this section as they are the main studied topic. Then we introduce the recently proposed regret minimization queries. Note that further details about related work will be presented in each chapter.

1.1 Skyline queries

The *Skyline* operator was first known as the *pareto* frontier in economics research [26]. It was as well studied beforehand in computational geometry as the maximal vector problem [27, 28]. The *pareto* frontier is composed of optimums, such that given any two optimum points p_1 and p_2, there exists at least one property f_i where p_1 is better than p_2, and at least one property $f_j, j \neq i$ where p_2 is better than p_1. Figure 1.1 illustrates the above explained property. Here, smaller values are better values. Points crossed by red line are optimums and hence belong to the Pareto frontier. E.g., observe the points A and B. B is better than A on f_1, and A is better than B on f_2. Both are optimums because no other point is better than them on both f_1 and f_2.

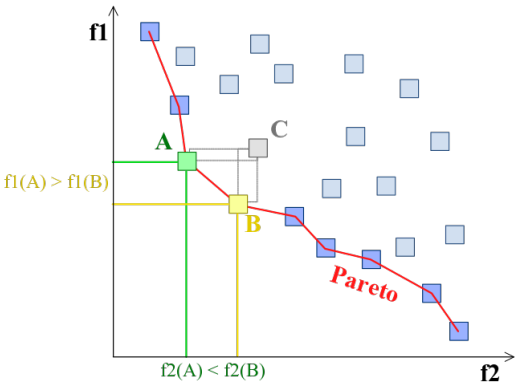


Figure 1.1: Pareto frontier

[2] introduced the skyline query as an alternative to Top-K queries. Given a dataset T over a set of attributes $\mathcal{D}$, the skyline set $Sky(T)$ is the set of the *best* tuples of T. Its computation relies on *domination* relation. We say that a tuple t dominates another tuple t' iff t is better or equal on all attributes and strictly better on at least one attribute. The set of skyline tuples is then the set of non dominated tuples.

[2] proposed to extend the SQL syntax to handle the skyline operator by DataBase Management Systems (DBMS) as shown below. The attributes wrt (with respect to) which the skyline is to be computed are listed after the term "SKYLINE OF". Moreover for each attribute, it is necessary to precise if it is to be minimized or maximized.

```
SELECT ...FROM ... WHERE ...
GROUP BY ... HAVING ...
SKYLINE OF [DISTINCT] d1 [MIN | MAX | DIFF], ..., dn [MIN | MAX | DIFF]
ORDER BY ...
```

Evaluating a skyline query on top of a relational database system can be done by converting the skyline query into a nested SQL query. Hereafter an example of a skyline query over the dataset in Table 1.

```
SELECT * FROM flight s1
WHERE NOT EXISTS (SELECT * FROM flight s2
WHERE (s2.price <= s1.price
AND s2.stop <= s1.stop )
AND (s2.price < s1.price
OR s2.stop < s1.stop);
```

This naive implementation has the disadvantage of being time-consuming. It involves a self-join over the table *flight*. For every tuple handled through $s1$, a full read of the same table is executed in order to check if the tuple is dominated. Nonetheless, the evaluation of this SQL query in a database management system is enhanced when data (columns) are indexed [29].

In the following sections, we present (i) relevant algorithms in the literature for computing the skyline from scratch and (ii) approaches for updating a materialized skyline in a dynamic context. Moreover, we present the two variants we consider in this thesis: (i) subspace or multidimensional skyline and (ii) skyline over partially and dynamically ordered dimensions.

1.1.1 Algorithms

We can divide the relevant skyline algorithms into two groups: (i) early algorithms that mainly targeted pruning comparisons, and (ii) algorithms that used intelligently

partitioning in order to speed up computation. The second group provided a significant improvement into skyline computation time.

Early algorithms

Authors of [2] which introduced the skyline operator, proposed *BNL* algorithm (*block nested-loops*). It incrementally discards dominated tuples. First, it initializes the skyline set S with some random tuple from the dataset. Then it iterates on the whole dataset. For each tuple t in the dataset, it compares it to tuples in S. If t is found dominated, it is discarded and never considered again. If t is not dominated, it is then appended to S. Moreover, tuples in S which are found dominated by t are discarded.

[2] proposed also D&C, a divide and conquer like algorithm. It naively partitions the dataset into several subsets and computes the skyline wrt each subset. The intermediate results from each subset are merged and a final skyline computation is performed.

On another side, [3, 4] proposed index based techniques. They specifically used R-Trees [30, 31]. We recall that R-Trees are used for multidimensional data indexing. Their respective algorithms, i.e. *NN* (*Nearest Neighbor*) and *BBS* (*Branch and Bound Skyline*) proceed by a recurrent nearest neighbor search. At the beginning, they select the nearest tuple to the origin o (consider smaller values are preferred). Let us call this tuple s_1. This tuple is appended immediately to the skyline set. Then they partition the dataset wrt s_1 into 2^d regions. The region delimited by $s1$ and o is empty. One region is called the dominance region, i.e., all tuples in this region are dominated by s_1, and hence are discarded. Finally the $2^d - 2$ regions are called anti-dominance regions as the tuples in these regions are incomparable with $s1$. This process (nearest neighbor selection, partitioning and pruning dominated tuple) is repeated for all remaining regions until all tuples are either found dominated or belonging to the skyline. R-trees are used by these algorithms in order to speed up the selection of the nearest neighbor. Figure 1.2 shows the first iteration of the above process. The tuple b is the nearest neighbor tuple to the origin. First, it is appended to the skyline set. Then data is partitioned wrt b. Tuples a, e, d, f are discarded as they are in the dominance region of b. The process is repeated on regions containing (i) f, and (ii) g and c.

[5, 6] proposed a pre-sorting algorithm based on the following observation: given any ascending scoring function f, a tuple t is not dominated by a tuple t' iff $f(t)$ is smaller than $f(t')$ (assuming small values are preferred). Hence, their proposed algorithm *SFS* sorts the dataset wrt a function f, and checks the dominance of a tuple only wrt tuples having better (smaller in this case) scores. More precisely, the algorithm starts by appending the tuple with the lowest score to the skyline set S. Then, it handles the remaining tuples in

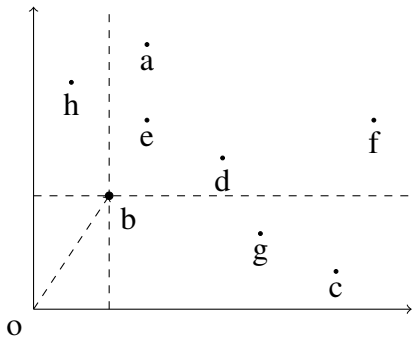


Figure 1.2: Data partitioning wrt *nearest neighbor* point

an ascending order, and it compares them to tuples in S. If a tuple is found **not** dominated by tuples in S then it is surely a skyline tuple and is appended to S. [32, 33] came up with improvements for SFS algorithm with respectively $LESS$ and $SaLSa$ algorithms but both use this notion of scoring function.

[34] proposed a dedicated index structure inspired from Bitmap [35]. It encodes data in order to identify the skyline tuples through bitwise "&" operation. However, it is not suitable for high dimensionality and has poor maintenance performance.

Partitioning and parallelization

The algorithms *NN* and *BBS* presented above have been the first to come up with a partitioning technique. Their ability to prune dominated tuples has been shown higher than other's. However, despite this performance, these algorithms lack scalability wrt the number of dimensions. Indeed, it is likely that data becomes anti-correlated when the number of dimensions grow. Hence, tuples are often incomparable and the pruning power of these methods weakens. Moreover, the partitioning generates 2^d regions (where d is the number of dimensions), each of which needs to be processed. Hence the algorithm has an exponential complexity wrt the number of dimensions.

Nevertheless, the partitioning technique remains efficient. The following algorithm BSkyTree [7, 8] adopted a different approach for pivot selection, i.e., tuple wrt which data is partitioned. Recall that for *NN* and *BBS*, the pivot tuple is always the nearest neighbor to the origin. Authors of BSkyTree have pointed out that existing techniques have weak performance in presence of high dimensionality, and proposed a cost-based selection of pivot tuple for partitioning data that balances both comparability and incomparability. Once the pivot tuple is selected, the dataset is divided into sub-regions, and dominated tuples are pruned. The process is then repeated for each sub-region. Experiments showed that BSkyTree outperforms by two orders of magnitude the existing algorithms.

[9] came up with Hybrid, a multi-core partitioning based techniques. During the process, it maintains a shared, global skyline among all threads, which is used to minimize dominance tests while maintaining high throughput. The algorithm uses an efficiently updatable data structure over the shared, global skyline, based on tuple-based partitioning. Then, [36] proposed SkyAlign an adaptation of Hybrid to GPGPU (General Purpose Graphical Processing units). To our knowledge, BSkyTree, Hybrid and SkyAlign are respectively single core, multi-core, and GPU state of art techniques for processing skyline queries.

On another side, works considered the paradigm *Map Reduce*. This paradigm has been developed for distributed architecture. It allows to distribute computation on a cluster of machine (map) and aggregate the intermediate results (reduce). For processing skyline queries, the *map* step consists in computing skyline wrt subsets of the dataset. The *reduce* consists in gathering the skylines and computing the final skyline. [37] proposed to use *Map Reduce* also in the step of data partitioning.

1.1.2 Materialization and dealing with updates

The algorithms presented in the previous section compute the skyline from scratch, i.e., every time a skyline query is issued, these algorithms run through the whole dataset. In a real world case where thousands of queries are issued simultaneously, running these algorithms for every issued query is not manageable whatever their efficiency.

Materialization is a technique in databases that provides fast query processing. It consists in storing the queries results in the disk, and retrieving them whenever the same query is issued. However materialized results need to be updated whenever the underlying data change.

Regarding skyline queries, updating materialized results is challenging, mainly because they are not monotonic [38]. The skyline set can change dramatically by both insertions and deletions. However deletions have been shown harder to deal with than insertions. Concretely, given a table T and its corresponding materialized skyline set S. A newly inserted tuple t^+ can either join S and exclude zero or more tuples from S, or be dominated by tuples in S. For a recently deleted skyline tuple $t^- \in S$, non skyline tuples in T may join the skyline set S.

We consider two types of data that change over time:

- Dynamic data: a number of tuples are inserted/deleted at any time.
- Streaming data: a stream of tuples over a window, i.e., tuples have a unique specified lifetime after which they are deleted

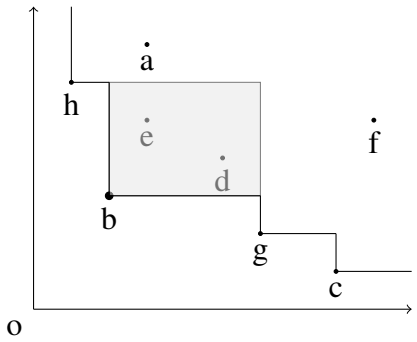


Figure 1.3: Exclusive dominance region of b

We make this categorization (dynamic data vs. streaming data) because one efficient approach wrt dynamic data may not be efficient wrt to streaming data, and vice versa.

Dynamic data Works have mainly addressed the deletion of skyline tuples as it is considered more challenging than insertion. [4] was first to introduce the notion of *exclusive dominance regions* EDR. Consider a dataset T and its skyline set S. Let $t \in S$, $EDR(t)$ consists of tuples not dominated by any other skyline tuple than t. Hence $EDR(t)$ constitutes the set of candidate tuples that will integrate the skyline set once t is deleted. Figure 1.3 shows the *exclusive dominance region* of the tuple b. $EDR(b)$ is the rectangle in gray. Observe that for a and f, despite being dominated by b, are not in $EDR(b)$ because they are dominated by respectively h, and g and c. However the idea was not developed nor implemented in that paper. A naive algorithm for computing EDR runs in time $O(s^d)$, where s is the size of the current skyline and d the number of dimensions. Later, [10] proposed an $O(s \cdot d)$ algorithm, called *DeltaSky*.

Streaming data The semantics of skyline queries in a streaming data context is: the skyline over tuples arrived in the window $(\tau - \omega, \tau]$ such that ω is the size of the window and τ the current time. When queries consider all tuples arrived so far, $\omega \to +\infty$. Continuous skyline queries results are meant to be accurate with the current state of the dataset [39]. To the best of our knowledge, [11] is the first work to address the continuous skyline query answering. It was motivated by the fact that state of the art algorithms are not efficient in presence of data streams. Their approach consists of maintaining two sets of tuples *DBsky* and *DBrest*. *DBsky* stores skyline tuples, and *DBrest* stores skyline candidates, i.e., tuples waiting some tuples from *DBsky* to expire. They proposed two approaches to maintain these sets: *Lazy* method consists of (i) storing the incoming tuple either in *DBsky* or *DBrest* (ii) discarding from *DBsky* outdated tuples and (iii) migrating tuples from *DBrest* to *DBsky* if they become skyline tuples. The second method, called *Eager*, optimizes the migration of tuples from *DBRest* to *DBSky* by storing an event list

which indicates at what timestamp a tuple in *DBRest* could integrate *DBSky*. Later, [40, 41] proposed a slightly different approach. Their algorithm called *LookOut* maintains a skyline set *Sky* and an R-tree of the database. An incoming tuple t is processed by the procedure called *isSkyline* that takes t and the R-tree as inputs and returns true if t is a skyline tuple. Yet, these methods remain unsuitable for multidimensional queries as the structures they manage and the maintenance processes need to be replicated for every subspace, hence an exponential time and space complexity wrt the number of dimensions. We note however that they guarantee an immediate query answering as the skylines are fully materialized. [42] addressed subspace skyline query answering. They proposed to maintain potential subspace skyline tuples besides the full skyline (skyline wrt all attributes). Then they answer issued queries through both sets. While this approach stores less data. The maintenance and query answering procedure is more costly.

[43] performed an empirical evaluation of the methods described above and showed that *Eager* method presented in [11] is the most efficient wrt execution time, but requires more memory due to the maintenance of the event list. It is the approach to which we compare our proposal in Section 4.4.3.

In a similar field, [44] addressed the skyline query with temporal constraints. However without considering the streaming behavior.

Hereafter we present the research lines that we consider in this manuscript, i.e. (i) multidimensional skyline and (ii) skyline over data with partially and dynamically ordered dimensions.

1.1.3 Subspace skyline answering and the SkyCube structure

Consider again the dataset shown in Table 1. One user, who travels on budget, may be interested into the skyline wrt the attributes Price and Duration, while another, richer, may be interested into the skyline wrt the attributes Duration and # of stops. This use case motivated the subspace (subset of dimensions) skyline research and the Skycube concept. The latter is the set of skylines wrt every possible subspace. [45] and [46] have independently introduced subspace skyline queries (multidimensional skylines) and the Skycube concept. They adapted existing algorithms for full skyline to subspace skyline. [47] highlighted the inefficiency of existing algorithms to deal with multidimensional skyline and proposed *SUBSKY*. It encodes multidimensional tuples into 1 dimension values, and indexes them with a B-tree. Then answers subspace skyline queries using that structure. [48] introduced the concept of Extended Skyline (Ext-SKY). These are the tuples that are not totally and strictly dominated. Those tuples not belonging to this set

do not belong to any skyline wrt any subspace, thus can be removed from the underlying data so as to simplify any subsequent computation.

[49] proposed an algorithm for computing the whole Skycube. In this line, [12] came up with QSkyCube which computes in a top-down fashion the skyline for each cuboid (subspace) using a tree-like structure. Later, [50] proposed RSkyCube, an optimization of QSkyCube. The reported experiments show up to 10 fold speed up. [13] showed how one can benefit from the functional dependencies holding in the dataset to optimize both full and partial Skycube computation. Following works propose dedicated index structure that speed up subspace skyline computation. [15] proposed the HashCube structure. It consists essentially to associate a 2^d Boolean vector to every tuple t where position i in the vector is set iff t belongs to $Sky(i)$. Here, i identifies a subspace. While query evaluation is very efficient (for a query $Sky(i)$, check position i for every tuple, hence $O(n)$) the memory consumption is large: $O(n \times 2^d)$. In order to save space, each vector is divided into subvectors (called words) of size w. Hence, each word encodes a set of subspaces. To each word ω_j, is associated a set of tuples sharing ω_j. The worst case storage is $\frac{2^d}{\omega} \cdot 2^\omega$ words. However this limit is hardly reached as a word for which no tuple is associated is not stored. In practice, this encoding reduces memory by a factor 10. Even if this space compression technique comes with a little overhead for query evaluation (for $Sky(i)$, one needs to traverse all the words, check whether the subspace i is set in a word then add its associated tuples to the result), HashCube is still remarkably fast for query evaluation. [51] extended [15] by proposing, among others, *mdmc*, an algorithm for building the HashCube structure. Regarding HashCube maintenance, inserting a new tuple t can be handled by comparing it to every other tuple in order to update the previous bit vectors and create the vector associated to t. However, deleting an old tuple requires rebuilding the HashCube from scratch and this represents a severe limitation when dealing with dynamic data. [14] proposed the Compressed Sky Cube (CSC). Its main idea consists in associating to every tuple t the smallest subspaces X, in terms of set inclusion, such that t belongs to $Sky(X)$. $Sky(X)$ query is evaluated by first computing the union of the sets of tuples t such that $Y \subset X$ is associated to t. Then a standard skyline procedure is evaluated on the so obtained tuples set. We also note that CSC, to our best knowledge, is the only structure for which an incremental maintenance procedure has been provided. Despite its advantages, the experiments conducted by [15] show that query evaluation via CSC is not efficient. [52] proposed the *closed* Skycube structure. The technique clusters all equal subspace skylines into equivalence classes so that a single copy is materialized. Even if this solution provides an optimal query response time, finding the equivalent classes is time consuming, actually, more than computing the Skycube. Moreover, the size of the

closed Skycube may reach that of the Skycube in the case where subspace skylines are all different from each others. Reference [49] also proposed a condensed representation of the Skycube. It associates to every tuple a set of pairs $\langle Top, Bottoms \rangle$ encoding the subspaces where t is in the skyline. For example, if the pair $\langle ABC, \{A, B\} \rangle$ is associated to t, then for every X such that $X \subseteq ABC$ and $X \supseteq A$ or $X \supseteq B$, t belongs to $Sky(X)$. [14] proved that CSC is smaller than this condensed structure.

In Chapter 2, we present the dedicated index structure for answering subspace skyline queries NSC, for which we study its incremental maintenance in this manuscript.

1.1.4 Skyline wrt partial and dynamic orders

Usually, data is described over nominal attributes, e.g. companies operating flights or movies genre. Initially, there does not exist any order over these attributes' domain. Users are asked to express their preferences (orders) which can be *partial*. Also, orders change from a user to another, i.e., *dynamic*. Existing algorithms for skyline query evaluation mainly consider data over attributes with static and total orders. Moreover, these existing techniques can not easily be extended to handle data over attributes with partial and dynamic orders.

We note two approaches in the literature for handling this case: (i) algorithms computing the skyline from scratch and (ii) materialization-based techniques.

Algorithms

In lattice theory, it is well known that every partial order can be embedded into a product of a set of total orders [20, 21]. This inspired [19] to propose *CPS*, a transformation technique of every partially ordered dimensions. Finding the minimal number of total orders is NP-complete. So, [19] used an approximate algorithm. A skyline algorithm for totally ordered dimensions is then applied on the transformed dataset.

[53] proposed to transform each partially ordered dimension into *two* totally ordered dimensions. The transformed dataset is then processed by any standard algorithm. However, the output may include false positives, because of the restricted number of total orders. So a filtering pass on the output is required.

[54] proposed the framework *TSS*. It transforms a partially ordered dimension into a *single* totally ordered dimension corresponding to one of its topological orders. Likewise [53], a filtering step is needed after getting a first skyline because of false negatives.

Hence, *CPS* [19] is the only accurate technique. Moreover, empirical studies showed that *CPS* combined to BSkyTree outperforms techniques in [19, 54, 53] wrt query answering time.

Materialization based techniques

[55, 22, 56] addressed skyline queries over dataset with partially and dynamically ordered dimensions with materialization-based techniques. [56] proposed a tree-like structure *Ordered Skyline Tree OST* in order to materialize the skylines wrt every total preference. A query q related to a preference $q.\mathcal{R}$ is evaluated through combining the skylines of different total preferences. The number of total preferences on one attribute is $m!$, where m is the cardinality of the nominal attribute. Hence, the memory usage of this tree can rapidly become a bottleneck. Handling several dimensions worsen this limitation, i.e., $(m!)^l$. Nonetheless a compressed version of *OST*, denoted *CST*, has been presented and whose worst case of memory usage may reach that of *OST*.

In [55] and its extension [22], authors proposed answering queries by refinement process. Let q, q' be two skyline queries and $q.\mathcal{R}$, $q'.\mathcal{R}$ be their respective preferences. We say that q is a refinement of q' iff $q'.\mathcal{R} \subseteq q.\mathcal{R}$. In such case it is easy to see $ans(q.\mathcal{R}) \subseteq ans(q'.\mathcal{R})$. Suppose that a set Q of queries are materialized and consider q as a new submitted query. Their solution consists first to find a refinement $q' \in Q$ of q and then use its materialized result to evaluate q. The authors propose an index structure to find a refinement given a query. Unfortunately, this index is not complete in that, some refinement can be missed. Hence, it cannot return the *best* refinement, i.e., the one whose result is the smallest.

Recently, [57] considered the problem of maintaining several skylines corresponding to different users preferences. Consider two users looking for Ferraris' deals on Internet. User1 prefers Ferraris with (i) red color over yellow, and (ii) yellow color over green. While user2 prefers Ferraris with red color over the yellow and green colors, and has no preference between yellow and green. The authors propose to measure the similarity between user's preferences in order to share skyline computations. Hence when a new Ferrari deal is available, it is decided whether it belongs to each user's skyline with less cost.

[58] studied skyline queries on datasets with categorical attributes, i.e., having very small domains, e.g, values are either *True* or *False*. However they considered only totally and statically ordered attributes.

1.1.5 Reducing the query output size

The skyline set becomes rapidly close to the whole input dataset when the dimensionality grows and data are anti-correlated. In that setting, the skyline set becomes of minimal interest. Works have proposed techniques to solve this counter-performance.

[4, 59] proposed Top-k skyline queries, it consists on selecting K skyline tuples with the highest score wrt to a utility function f. They showed that their algorithm handles this extension. However, it is of minimal practicality as the skyline operator was proposed in order to avoid the user to express a utility function. [60] proposed the K-dominating queries which return the tuples that dominate the largest number of other tuples. [61] proposed a similar query: the k-representative skyline tuples (Top-k RSP). They propose the function D which given a subset $S \subseteq T$, $D(S)$ represents the number of tuples dominated by tuples in S. The output of Top-k RSP is the set of K skyline tuples that maximizes D. This technique has been shown useful for reducing the output size, however it may discard interesting tuples. [62] showed that the previous technique does not output good representative and redefined the problem of identifying the k-representative skyline tuples based on a distance metric. In [63], authors proposed the epsilon-skyline. It allows to reduce the size by discarding tuples that have *bad* values in some dimensions.

The following work considered ranking tuples with respect to their behavior in subspaces. Authors in [60] proposed a new metric called skyline frequency. It represents the number of subspace skylines to which a tuple belongs. [64] proposed *skyrank* which ranks tuples based on the number of tuples it dominates by considering all subspaces.

1.1.6 Variants of skyline queries

Works have proposed variants of skyline query to deal with specific use cases. We cite few of them in this section. [4, 59] proposed the dynamic skyline. This query aims to capture tuples close to a given query tuple t. In this setting, a tuple t_1 dominates a tuple t_2 if the distance wrt some function between t_1 and t is *better* than that between t_2 and t. A use case is e.g., a user selects a house on a real estate platform and the query retrieves the similar houses. As a dual query, the reverse skyline query [65] retrieves those tuples in the database whose dynamic skylines contain a given query tuple. Authors proposed dedicated efficient algorithms for the above queries. Nonetheless they can be evaluated by traditional skyline algorithms. Finally, [66] proposed the group skyline which consists in returning a group of K tuples not dominated by any other group of K tuples as well. This technique can also be used to control the size of the output.

1.2 Regret minimization queries

[17] presented the regret minimization queries to leverage the benefits of skyline [2] and Top-k [1] queries, and exclude their limitations. Like Top-K queries, it bounds the result size and like Skyline queries, it does not require the user to provide a scoring function.

Hotels	Price	Distance
h_1	200	120
h_2	390	140
h_3	465	20
h_4	395	90
h_5	100	300

Table 1.1: Hotels

Next, we recall the skyline queries and Top-K queries and illustrate their behavior through Table 1.1. The Skyline queries are based on the dominance relation. A tuple t is said to be dominated by a tuple t' iff (i) t' is *better* or equal on all dimensions and (ii) t' is strictly *better* on at least one dimension. The Skyline result is then the set of non dominated tuples. Top-K queries are based on scoring functions given by users. Often, scoring functions are linear, e.g. $f(t) = \sum_{i=1}^{d} w[i] * t[i]$ where w is called the weight vector. In a normalized setting, $0 \leq w[i] \leq 1 \, \forall i \in [1,d]$ and $\sum_{i=1}^{d} w[i] = 1$. The result of Top-K query, by considering the scoring function f, is K tuples with the best scores.

Example 2. *Consider Table 1.1 that describes Hotels by their price and their distance from the beach. Suppose that cheaper and closer to the beach is better*

The Skyline set with respect to this dataset is illustrated in Table 1.2. Only h_2 does not belong to the Skyline set because it is dominated by t_1. Indeed, t_1 is cheaper and closer to the beach. Observe here that we can not control the result size.

Hotels	Price	Distance
h_1	200	120
h_3	465	20
h_4	395	90
h_5	100	300

Table 1.2: Skyline hotels

Table 1.3 represents the hotels' score wrt three linear scoring functions. Note that lower the score the better the hotel. Top-1 hotels score is underlined wrt every function. h_1 is Top-1 wrt $(0.5, 0.5)$, h_3 is Top-1 wrt $(0.2, 0.8)$ and h_5 is Top-1 wrt $(0.8, 0.2)$

[17] presented the regret minimization queries (RMS) to avoid the limitations of skyline and Top-k queries, i.e., the unbounded result of Skyline queries and the need of scoring functions for Top-K queries. The main idea is to select a subset S from a dataset T such that S minimizes the user regret. In a nutshell, the regret represents how far the user's *best* tuple in S is from the user's *best* tuple in T. For example and to simplify, consider the family of 3 functions $\mathcal{F} = \{f_{(0.2,0.8)}, f_{(0.5,0.5)}, f_{(0.8,0.2)}\}$. Now consider the

Hotels – Weight vector	$(0.2, 0.8)$	$(0.5, 0.5)$	$(0.8, 0.2)$
h_1	136	<u>160</u>	184
h_2	190	215	340
h_3	<u>109</u>	242.5	376
h_4	151	242.5	334
h_5	260	200	<u>140</u>

Table 1.3: Top-K hotels

set $S = \{h_3, h_1\}$. The maximum regret ratio of S wrt $\mathcal{F}$, i.e. $mrr(S, \mathcal{F})$, is 31.4%. This represents the ratio between the best score within T and the best score within S wrt the function $f_{(0.8,0.2)}$. Concretely, this means that for a user whose scoring function is in $\mathcal{F}$, the best score he can get from S is at most 31.4% less than the best score he can get from T.

[17] formalized the RMS problem as follows:

> **Problem RMS** Given a dataset T, the family of all linear scoring functions $\mathcal{L}$, an integer K, compute a set $S \subset T$ of size K that minimizes the maximum regret ratio $mrr(S, \mathcal{L})$.

Now, we present how the maximum regret ratio is computed. Let $f \in \mathcal{L}$ be a scoring function, and given a dataset T, let $f_1(T)$ be the highest score by considering tuples in T. The regret of a subset $S \subseteq T$ wrt f is $f_1(T) - f_1(S)$ and the regret ratio is $\frac{f_1(T)-f_1(S)}{f_1(T)}$. The maximum regret ratio is then $mrr(S, \mathcal{L}) = max_{f \in \mathcal{L}} \frac{f_1(T)-f_1(S)}{f_1(T)}$.

[18] proved the NP hardness of the RMS problem and [17] proposed a greedy approximate algorithm to solve it. The regret minimization set (RMS) has been shown (i) scale-invariant, i.e. the maximum regret ratio remains the same even if the values in the dataset are multiplied by the same factor, and (ii) stable, i.e. the RMS does not change when weak tuples (tuples not having the highest score wrt any scoring function) are inserted or deleted from the dataset.

Algorithms for solving RMS belong to three categories: (i) those solving it exactly and in polynomial time for 2 dimensions' dataset [18, 67, 68], (ii) heuristic-based [17, 69, 70] and (iii) those providing theoretical guarantees [17, 71, 67, 69, 72, 68]. Sphere [69] is currently the state of the art algorithm. Also, it provides theoretical guarantees on the output regret.

1.2.1 Variants of regret minimization queries

[18] proposed a relaxation of RMS, namely the k-regret minimizing set (kRMS). The k-regret represents how far the user's *best* tuple in S is from the k^{th} user's *best* tuple

in T. Concretely, let $f \in \mathcal{L}$ be a scoring function, k be an integer, then let $f_k(T)$ be the score of the k^{th} ranked point using f. The k-regret of a subset $S \subseteq T$ wrt f is $max(0, f_k(T) - f_1(S))$[1] and the regret ratio is $\frac{max(0,f_k(T)-f_1(S))}{f_k(T)}$. The maximum regret ratio is then $max_{f\in\mathcal{L}} \frac{max(0,f_k(T)-f_1(S))}{f_k(T)}$. [73] introduced the regret minimization problem wrt non-linear scoring functions such as concave and convex functions. [74] considered the average regret ratio rather than the maximum regret ratio. Finally, [75] proposed the rank regret minimization queries. Authors measure the regret based on the rank difference rather than the score. The exact semantic of rank regret minimization query is: Given a dataset T, a family of scoring functions $\mathcal{FL}$ and an integer k, compute $S \subset T$ such that $\forall f \in \mathcal{L}\ \exists t \in S$ such that t is at worst ranked k^{th} wrt f.

1.2.2 Candidate sets for RMS

[17] showed that it suffices to consider the skyline set to compute the RMS rather than the whole dataset. In other words, the optimal solution S^* is composed of skyline tuples. [70] presented an even smaller candidate set, namely *Happy* tuples. However, its computation time is a weakness. Its time complexity is $O(n^2 * d^2)$ where n is the size of the dataset and d the number of dimensions. [76, 77] showed that one can leverage from Skycube to optimize regret queries. Concretely, they proposed the Top-K frequent skyline set and Top-K priority skyline set as candidate sets for RMS. Until now, there is no theoretical guarantee on the RMS calculated from these sets. In this manuscrispt and specifically in Appendix A, we investigate the improvement provided by these candidate sets on RMS by using NSC. Moreover we empirically evaluate the output regret of these approaches compared to RMS dedicated algorithm Sphere [69].

[1] The regret is always positive

Chapter 2

Preliminaries

2.1 Global notations and definitions

Let $T(Id, \mathcal{D})$ be a relation where $\mathcal{D} = \{D_1, \ldots, D_d\}$ is a set of attributes called also dimensions. A subspace, hereafter denoted $X, Y, \ldots$ is a subset of $\mathcal{D}$. We assume now that the domain of every D_i is associated to a *total* order $<_i$, or simply $<$ expressing the preference of users.

Hereafter the main definitions for this manuscript: (i) Dominance and (ii) Skyline. Note that these definitions may slightly change in next Chapters.

Definition 1. *Dominance: Given two tuples t and t' and a subspace X, t dominates t' w.r.t. X, denoted $t \prec_X t'$, iff $\forall D_i \in X : t[D_i] \leq t'[D_i]$ and there exists $D_j \in X$ s.t $t[D_j] < t'[D_j]$. We say that t' is $\mathcal{X}$-dominated by t.*

Definition 2. *Skyline: The skyline of T w.r.t. X, denoted $Sky(T, X)$ is the set of tuples $\{t \mid \nexists t' : t' \prec_X t\}$. We sometimes write just $Sky(X)$ when T is clear from the context.*

Example 3. *Table 2.1 will be used as a running example throughout the section. Using this dataset, users may ask for the best tuples (skyline tuples) regarding every combination of the dimensions $\{A, B, C, D\}$. For instance, $Sky(AB) = \{t_1, t_2\}$ and $Sky(ABCD) = \{t_2, t_3, t_4\}$.*

Table 2.2 summarizes the notations used throughout this manuscript. Note that some notations will be redefined according to the needs of each chapter.

2.2 The Negative SkyCube

In this section, we present NSC (Negative SkyCube) [16] a concise data structure which, for every tuple t in T, summarizes the set of subspaces X such that t *does not belong*

Id	A	B	C	D
t_1	1	1	3	3
t_2	1	1	2	3
t_3	2	2	2	2
t_4	4	2	1	1
t_5	3	4	5	2
t_6	5	3	4	2

Table 2.1: Dataset T

Notation	Definition
T	input table
$t, t_1, \ldots$	tuples
n	number of tuples in T
$\mathcal{D}$	the set of all dimensions
d	number of dimensions
$A, B, \ldots$	dimensions
$X, Y, \ldots$	subspaces, i.e., subsets of $\mathcal{D}$
$Sky(X)$	skyline w.r.t. X
$topmost$	$Sky(\mathcal{D})$
$compare(t, t')$	the pair resulting from comparing t to t'
$\langle X \vert Y \rangle$	a pair of subspaces
$Pairs(t, T)$	set of pairs associated to t w.r.t. T
$\mathsf{NSC}(T)$	set of $Pairs(t)$ $\forall t \in T$

Table 2.2: Notations

to $Sky(X)$. This concept was motivated by the following observation: for a tuple t, while we need to compare it to every other tuple in order to state that it belongs to some $Sky(X)$, comparing t to just one t' can inform us about a whole set of subspaces where t is dominated, i.e., it does not belong to their respective skyline. Then, answering a subspace skyline $Sky(X)$ consists in finding through the structure the tuples that are not dominated wrt X. In the following we present how NSC is (i) constructed, (ii) optimized wrt time and space, and (iii) used for answering skyline queries.

2.2.1 NSC construction

We start by providing some preliminary definitions.

Definition 3 (Dominance subspaces)**.** *Let $t \in T$ and $X \subseteq \mathcal{D}$. X is a dominant subspace for t iff $t \notin Sky(X)$.*

Definition 4 (Negative SkyCube)**.** *Let $t \in T$ and let $Dom(t)$ denote the dominant subspaces for t. The negative skycube of T is the set $\{Dom(t) \,|\, t \in T\}$.*

In other words, the negative Skycube stores for every tuple t, the subspaces where it does not belong to their respective skyline.

Example 4. *From the running example in Table 2.1, it is easy to check that $Dom(t_1) = \{ABCD, ABC, ACD, BCD, AC, BC, CD, C, D\}$.*

Clearly, the computation of every skyline $Sky(X)$ is straightforward using NSC: for every tuple t, $t \in Sky(X)$ iff $X \notin Dom(t)$.

Actually, NSC does not store $Dom(t) \forall t \in T$ but a more concise summary, i.e., $Pairs(t) \forall t \in T$. Next we present this concept and we show how NSC can be computed.

We first show how by comparing t to some t' we obtain a set of subspaces where t is dominated.

Definition 5. *Let $t, t' \in T$. We define a comparison function* **compare** *as follows: $compare(t,t') = \langle X|Y \rangle$ such that X is the set of dimensions D_j such that $t'[D_j] < t[D_j]$ and Y is the set of dimensions D_ℓ for which $t'[D_\ell] = t[D_\ell]$.*

Example 5. *From Table 2.1, we have $compare(t_5, t_6) = \langle BC|D \rangle$ because $t_6[B] < t_5[B]$, $t_6[C] < t_5[C]$ and both tuples are equal on dimension D.*

Obviously, if $compare(t,t') = \langle X|Y \rangle$ then $X \cap Y = \emptyset$.

Definition 6 (Coverage)**.** *Let $\langle X|Y \rangle$ be a pair of disjoint subspaces and let Z be a subspace. We say that $\langle X|Y \rangle$ covers Z iff $Z \subseteq XY$ and $Z \cap X \neq \emptyset$. By $cover(\langle X|Y \rangle)$ we refer to the set of subspaces covered by the pair $\langle X|Y \rangle$*

Example 6. *$p = \langle AC|B \rangle$ covers subspaces A, AB, AC, BC and ABC. Note that B is not covered by p because even if $B \subseteq ACB$, $B \cap AC = \emptyset$.*

As stated in the following property of *cover*, the coverage of multiple pairs is the union of the sets of subspaces covered by the pairs.

Property 1. *Let $\{p_1, \ldots, p_n\}$ be a set of pairs then*
$cover(\{p_1, \ldots, p_n\}) = \bigcup_{i \in [1,n]} cover(p_i)$.

The following proposition shows that the covered subspaces by the pair we obtain when t is compared to t' are precisely the subspaces where t' dominates t. Consequently, they represent a fraction of $Dom(t)$.

Proposition 1. *Let $t,t' \in T$, $compare(t,t') = \langle X|Y \rangle$ and $Z \subseteq \mathcal{D}$. Then t' dominates t over Z iff $Z \in cover(\langle X|Y \rangle)$.*

Proof. 1) $Z \in cover(\langle X|Y \rangle) \Rightarrow t' \prec_Z t$: For every two disjoint subspaces Z_1 and Z_2 such that $Z_1 \cup Z_2 = Z$, $Z_1 \subseteq X$, $Z_2 \subseteq Y$ and $Z_1 \neq \emptyset$ we have: (i) $Z_1 \subseteq X \Rightarrow t' \prec_{Z_1} t$, and (ii) $Z_2 \subseteq Y \Rightarrow t' =_{Z_2} t$. Therefore $t' \prec_{Z=Z_1 \cup Z_2} t$.
2) $t' \prec_Z t \Rightarrow Z \in cover(\langle X|Y \rangle)$: Suppose $Z \notin cover(\langle X|Y \rangle)$, this means that $Z \cap X = \emptyset$ or $Z \supset XY$, then $\forall z \in Z\ t[z] \geq t'[z]$, therefore t' does not dominate t over Z. □

In fact, $compare(t,t')$ is a concise summary of the set of subspaces for which t is dominated by t' hence, subspaces where t does not belong to their respective skylines.

Throughout the rest of the paper we denote by $Pairs(t,T) = \{compare(t,t') \forall t' \in T\}$ the set of all pairs related to t w.r.t. T. We simply write $Pairs(t)$ when it is clear that the pairs are computed w.r.t. T.

Thereby $cover(Pairs(t)) = Dom(t)$ represents all subspaces where t is dominated, hence not in the skyline.

At this point we are ready to provide an algorithm for building the NSC. This structure can then be used to answer skyline queries. Algorithm 1 shows how this data structure is built. Its main principle consists simply in comparing every pair of tuples (t,t') and add $compare(t,t')$ to $Pairs(t)$. So a total of $n \times (n-1)$ comparisons. Each of which considers d dimensions. Hence, $O(n^2 \times d)$ comparisons. We could neglect d since $n \gg d$. From the memory point of view, in the worst case, $n-1$ *distinct* pairs are associated to every t, where $n = |T|$. In practice, this bound is hardly reached for two reasons: (i) when comparing t, we may obtain duplicate pairs while we need just a single copy of them. (ii) the number of possible distinct pairs depends on the number of dimensions. Indeed, with d dimensions this number is $N = \sum_{i=1}^{d} \binom{d}{i} 2^i$. From this expression, one can easily derive the following upper bound: $N \leq 2^{2d}$. Therefore, the NSC size is bounded by $n * min(n-1, 2^{2d})$. For example, if $d = 6$, a maximum of 2^{12} pairs can be associated to any tuple.

Note that if NSC size is $O(n * 2^d)$ then it is comparable to the skycube size which means that not only we do not summarize it in terms of memory but more importantly, we have no gain in terms of query performance. In the next section we address the problem of NSC size minimization.

Example 7. *From Table 2.1, the data structure returned by* BUILDNSC *is depicted in Table 2.3. Note that pairs $\langle X|Y \rangle$ where $X = \emptyset$ are not stored because they do not cover any subspace (see Proposition 1).*

Algorithm 1: buildNSC

Input: Table T
Output: A data structure NSC summarizing the Skycube.

```
1 begin
2     foreach t ∈ T do
3         foreach t' ∈ T do
4             Add compare(t,t') to Pairs(t)
5     return NSC(T)
```

Tuple	*Pairs*
t_1	$\langle C\|ABD\rangle, \langle CD\|\emptyset\rangle, \langle D\|\emptyset\rangle$
t_2	$\langle D\|C\rangle, \langle CD\|\emptyset\rangle, \langle D\|\emptyset\rangle$
t_3	$\langle AB\|\emptyset\rangle, \langle AB\|C\rangle, \langle CD\|B\rangle$
t_4	$\langle AB\|\emptyset\rangle, \langle A\|B\rangle, \langle A\|\emptyset\rangle$
t_5	$\langle ABC\|\emptyset\rangle, \langle ABC\|D\rangle, \langle BCD\|\emptyset\rangle, \langle BC\|D\rangle$
t_6	$\langle ABC\|\emptyset\rangle, \langle ABC\|D\rangle, \langle ABCD\|\emptyset\rangle, \langle A\|D\rangle$

Table 2.3: NSC of T

Algorithm 2 shows how the NSC structure is used to evaluate any skyline query $Sky(Z)$. For each tuple t, it scans the set of pairs associated to it. If a pair covering Z is encountered, then t does not belong to $Sky(Z)$. Otherwise, it is a skyline point.

2.2.2 Time and memory optimization for NSC

In this section we show on one hand, how the number of comparisons we must do with every tuple can be reduced, hence the overall execution time is minimized, and on the other hand, how to save space when storing the pairs associated to each tuple.

Execution time reduction

In this section, we show how the time to build NSC can be reduced by comparing every tuple to only tuples in the *topmost* skyline (See Definition 7).

Definition 7. *Topmost Skyline: $Sky(T,\mathcal{D})$ the skyline w.r.t. all the dimensions.*

As stated by Theorem 1, it is sufficient to compare the tuples to those belonging to the *topmost* instead of comparing every pair of tuples thereby avoiding a costly $O(n^2)$ comparisons. This is particularly interesting when the size of topmost skyline is small w.r.t. n.

Algorithm 2: evaluateSkyline

```
Input: NSC structure, subspace Z
Output: Sky(Z)
1  begin
2      foreach t ∈ T do
3          covered ← false
4          foreach p ∈ NSC[t] do
5              if p covers Z then
6                  covered ← true
7                  break
8          if covered=false then
9              Add t to Sky(Z)
10     return Sky(Z)
```

Theorem 1. *Let $t \in T$. Let*

- $P_T(t) = \{compare(t,t') | t' \in T\}$, *and*
- $P_{TM}(t) = \{compare(t,t') | t' \in topmost\}$.

Then $cover(P_T(t)) = cover(P_{TM}(t))$.

Proof. To simplify the notation, we omit the parameter t since it is understood from the context.

Clearly, $cover(P_T)$ can be written as

$$cover(P_{TM}) \cup cover(P_{\overline{TM}})$$

where

$$P_{\overline{TM}} = \{compare(t,t') | t' \notin topmost\}$$

That is to say, *cover* is *distributive* over the *union*. We just need to show that for all $t' \notin topmost$,

$$cover(compare(t,t')) \subseteq cover(P_{TM})$$

Let $t' \in T \setminus topmost$. By skyline definition, there must exist a tuple $u \in topmost$ such that u dominates t', i.e., $u \prec_{\mathcal{D}} t'$. Let $\langle X_1|Y_1\rangle = compare(t,u)$ and $\langle X_2|Y_2\rangle = compare(t,t')$. For every subspace Z covered by $\langle X_2|Y_2\rangle$, we have (*i*) $t' \prec_Z t$. On the other hand, $u \prec_{\mathcal{D}} t'$ implies that (*ii*) $u \preceq_Z t'$ (because $Z \subseteq \mathcal{D}$).

From (*i*) and (*ii*), $u \preceq_Z t$ thus Z is covered by $\langle X_1|Y_1\rangle$. Any subspace covered by $\langle X_2|Y_2\rangle$ is also covered by $\langle X_1|Y_1\rangle$.

Hence, for each $t' \notin topmost$, $compare(t,t')$ does not need to be considered. □

N.B: The *topmost* is computed ahead of building NSC. Any state of the art skyline algorithm can be used for that purpose.

Example 8. *From our running example, the topmost is made up of tuples t_2, t_3 and t_4. As an example, the list of pairs associated to t_1 is $\{\langle C|ABD\rangle, \langle CD|\emptyset\rangle\}$ whose size is reduced to 2 instead of a set of 3 pairs if we compare t_1 to all other tuples.*

Memory reduction

Reducing the size of NSC not only reduces memory consumption but also optimizes skyline queries evaluation. So the problem we want to solve consists in finding, for every t, a *minimal set* of pairs which covers exactly the subspaces covered by $Pairs(t)$. To give an intuition about how we proceed, let us consider the following example.

Example 9. *Let $p_1 = \langle A|BC\rangle$, $p_2 = \langle C|A\rangle$ and $p_3 = \langle BC|\emptyset\rangle$ be the pairs associated to t. From these pairs we derive $cover(p_1, p_2, p_3) = \{A, AB, ABC, AC, BC, B, C\}$. Observe that by considering just p_1 and p_3, the same subspaces are covered. Indeed, $cover(p_1) = \{A, AB, ABC, AC\}$ and $cover(p_3) = \{B, C, BC\}$. Hence, p_2 can be removed from $Pairs(t)$ without losing any information.*

We formalize the NSC size reduction problem as follows:

> **RSP Problem:** Given a tuple t and its associated set $P = Pairs(t)$. **R**educing the **S**ize of the set of **P**airs P, (**RSP**), is the problem of finding a subset $Q \subseteq P$ of minimal size such that $cover(Q) = cover(P)$.

The following theorem shows that **RSP** problem is NP-Hard.

Theorem 2. **RSP** *is NP-Hard.*

Proof. By considering all the subsets of P, one can check which are equivalent to P and which are of minimum size. Thus, the problem is in NP. The hardness proof is based on a reduction from the minimal set cover (**MSC**) problem. Given an **MSC** instance, we build a table T with a distinguished tuple t where the number of dimensions d is equal to the number of elements to be covered in **MSC** and where the number $n+1$ of tuples is equal to the initial number of sets in **MSC** in addition to the distinguished t. So, there is a bijection between the n tuples and the n sets of **MSC** instance. The n tuples form the topmost of T and distinguished tuple t is compared to each of them giving rise to a set of pairs P. We show that the minimum equivalent subset of P coincides with a solution of **MSC**.

Let $s = \{s_1, s_2, \ldots, s_n\}$ be the input set of sets in the **MSC** instance. W.l.o.g, we assume that there is no inclusion between these sets and none of them does contain all the elements to cover. For every set $s_j \in s$, we add to T a tuple t_j such that $t_j[i] = 0$ iff $i \in s_j$ otherwise $t_j[i] = 1$. In addition, we add to T a tuple $t = (1, 1, \ldots, 1)$ be a d-tuple.

For example, let $s = \{s_1 = \{1,2\}; s_2 = \{2,3\}; s_3 = \{1,3\}\}$ be the **MSC** instance. The number of elements to cover is $d = 3$ and the number of sets $n = 3$. So, we get a table T with $n+1 = 4$ tuples, including t, and 3 dimensions. This table is depicted below.

Id	**1**	**2**	**3**
t	1	1	1
t_1	0	0	1
t_2	1	0	0
t_3	0	1	0

Clearly, every t_j dominates t and $t_i \not\prec t_j$. Hence, $\{t_1, \ldots, t_n\}$ is the topmost. By comparing t to the topmost, we obtain $P(s) = \{p_1, \ldots, p_n\}$. There is a one to one correspondence between $s_i \in s$ and $p_i = compare(t, t_i)$. For example, $compare(t, t_1) = \langle 12|3\rangle$ corresponds to $s_1 = \{1,2\}$. Let $u \subseteq s$ and let $P(u)$ be the set of pairs p_j such that $p_j = compare(t, t_j)$ where t_j corresponds to some $s_j \in u$. Let $cover(P(u))$ denote the set of subspaces covered by the pairs in $P(u)$. We show that $\cup_{s_j \in u} s_j = \cup_{s_i \in s} s_i$ iff $P(u) \equiv P(s)$ and this proves the claim.

(i) $P(u) \equiv P(s) \Rightarrow u \equiv s$: Every $p_j \in P(u)$ is of the form $\langle X_j|Y_j\rangle$ thus it covers, among others, the subspace X_j which actually corresponds to the content of $s_j \in u$. As $P(u) \equiv P(s)$, $\forall p_i = \langle X_i|Y_i\rangle \in P(s)$, $P(u)$ covers X_i and the union of the X_i's is the union of the s_i's. Hence $u \equiv s$.

(ii) $u \equiv s \Rightarrow P(u) \equiv P(s)$: Assume, for the sake of contradiction, that $P(u) \not\equiv P(s)$. There must exist a subspace Z s.t $P(s)$ covers Z but not $P(u)$. Thus, there exists $p_i \in P(s)$ such that $p_i = \langle X_i|Y_i\rangle$ s. t $Z \subseteq X_i Y_i$ and $Z \cap X_i \neq \emptyset$. Note that every p_i is of the form $\langle s_i|\mathcal{U} \setminus s_i\rangle$ where $\mathcal{U} = \cup_{s_j \in s} s_j$. Therefore, to cover Z, a pair $\langle s_j|\mathcal{U} \setminus s_j\rangle$ needs just to satisfy $Z \cap s_j \neq \emptyset$. Such an s_j is necessarily in u because otherwise $u \not\equiv s$, i.e., there exists $k \in \mathcal{U}$ such that there is no $s_i \in u$ s.t $k \in s_i$ and thic contradicts the fact that $u \equiv s$.

We conclude that every (minimum) solution of the set cover problem corresponds to a (minimum) solution to **RSP** problem regarding the distinguished tuple t of the table T above which terminates the proof. □

[78] proposed a greedy polynomial time approximation of ***MSC*** (Minimum Set Cover) problem that chooses at each step the set that covers the highest number of uncovered elements. The adaptation of this algorithm to solve **RSP** problem is described in Algorithm 3.

Algorithm 3: compressByGreedy

Input: Set of pairs P
Output: Set of pairs $P' \equiv P$ with $|P'| \leq |P|$

```
1  for p ∈ P do
2      p.covers ← set of subspaces p covers
3  SubspacesToCover ← ∪_{p∈P} p.covers
4  while SubspacesToCover ≠ ∅ do
5      q ← argmax_{p∈P} |p.covers|
6      Add q to P'
7      Remove q.covers from SubspacesToCover
8      Remove q from P
9      for p ∈ P do
10         p.covers ← p.covers \ q.covers
11         if p.covers = ∅ then
12             Remove p from P
13 Return P'
```

Example 10. *The minimization of Table 2.3 is depicted in Table 2.4. Note that the number of pairs decreases from 20 to 9.*

Tuple	Associated list of pairs
t_1	$\langle C\|ABD\rangle, \langle CD\|\emptyset\rangle$
t_2	$\langle CD\|\emptyset\rangle$
t_3	$\langle AB\|C\rangle, \langle CD\|B\rangle$
t_4	$\langle AB\|\emptyset\rangle$
t_5	$\langle ABC\|D\rangle, \langle BCD\|\emptyset\rangle$
t_6	$\langle ABCD\|\emptyset\rangle$

Table 2.4: List of pairs synthesizing dominance subspaces sets

The following proposition states the time complexity and the approximation guarantee provided by Algorithm 3.

Proposition 2. *Let P be a set of pairs, P_{opt} be a minimal equivalent subset of P and P_G be the output of CompressByGreedy(P). Then (i) the time complexity of CompressByGreedy is $O(|P|^2)$ and (ii) $|P_G| \leq |P_{opt}| \times d$.*

Proof. For the time complexity, note that the first **for** loop (Line 1) is linear in $|P|$. At each iteration of the **While** loop (Line 4), we first select a pair q which covers the maximal number of subspaces. This can be done in linear time. Since such a q is removed from P, the **for** loop (Line 9) is executed $O(|P|-1)$ times. At worst, $|P|$ decreases by just

Algorithm 4: buildNSC_index

Input: Table T (set of tuples)
Output: NSC_index

```
1 begin
2     foreach tuple t in T do
3         foreach pair ⟨X|Y⟩ in Pairs(t) do
4             NSC_index[XY] = NSC_index[XY] ∪ {(t,Y)};
5     return NSC_index
```

one unit at each **While** iteration, hence the **while** loop is executed at most $|P|$ times, and each iteration executes a **for** loop with a decreasing size of P. Therefore, the global time complexity is $O(|P|^2)$.

Regarding the approximation multiplicative factor, let $p \in P$ such that p covers the maximum number of subspaces and let ℓ be this number. Then by [78], we have $|P_G| \leq |P_{opt}| \times \log \ell$. Since $\ell < 2^d$, we obtain $|P_G| \leq |P_{opt}| \times d$.

□

2.2.3 NSC index and query answering

Performing a query related to a given subspace Z requires to check for every tuple t, whether Z is covered by $Pairs(t)$, i.e., there exists $\langle X|Y\rangle \in Pairs(t)$ such that

- Z a subspace of XY, and
- $Z \cap X$ different from empty set ($\emptyset$)

Since when submitting a query $Sky(Z)$, only pairs $\langle X|Y\rangle$ such that Z is a subspace of XY are relevant, we use a subspace-based index to optimize the query evaluation process. More precisely, we use a table which associates to every subspace, a list of pairs of the form $\langle tuple|subspace\rangle$ as follows: let $\langle X|Y\rangle \in Pairs(t)$, then the pair $\langle t|Y\rangle$ is added to the list of XY. According to this structure, evaluating $Sky(Z)$ needs just to check those subspaces XY such that $Z \subseteq XY$. Algorithm 4 shows how to build this index structure. Example 11 illustrates the query evaluation process and Algorithm 5 depicts the procedure.

Example 11. *From our running example, the corresponding index structure is shown in Table 2.5. Note that subspaces with no tuples are removed.*

Let us show how $Sky(AB)$ is evaluated using Table 2.5. Only the lists associated to the supersets of AB i.e., AB, ABC, and ABCD are scanned. With AB, we find $\langle t_4|\emptyset\rangle$ meaning that to t_4 we associate $\langle AB|\emptyset\rangle$ hence t_4 is dominated on AB. The same holds with ABC

Subspace	*Pairs*
AB	$\{\langle t_4 \mid \emptyset \rangle\}$
ABC	$\{\langle t_3 \mid C \rangle\}$
CD	$\{\langle t_1 \mid \emptyset \rangle, \langle t_2 \mid \emptyset \rangle\}$
BCD	$\{\langle t_3 \mid B \rangle, \langle t_5 \mid \emptyset \rangle)\}$
ABCD	$\{\langle t_1 \mid ABD \rangle, \langle t_5 \mid D \rangle, \langle t_6 \mid \emptyset \rangle\}$

Table 2.5: NSC index

and t_3. From the list associated to ABCD, we deduce that for t_1 we have $\langle C \mid ABD \rangle$ and this pair does not cover AB. Hence, t_1 is not dominated. The pairs $\langle ABC \mid D \rangle$ and $\langle ABCD \mid \emptyset \rangle$ are respectively associated to t_5 and t_6. They both cover AB meaning that t_5 and t_6 are dominated w.r.t AB. Thus, only t_1 and t_2 belong to $Sky(AB)$[1].

Algorithm 5: evaluateSkyline_Index

Input: NSC_index, subspace Z, table T
Output: $Sky(Z)$

```
1 begin
2     NotSkylinePoints = ∅
3     foreach subspace W such that W ⊇ Z do
4         foreach pair (t, Y) in NSC_index[W] do
5             X = W \ Y
6             if Z is covered by ⟨X|Y⟩ then
7                 Add t to NotSkylinePoints
8     return T \ NotSkylinePoints
```

[1] In the concrete implementation, the table is sorted w.r.t. subspaces to avoid visiting useless subspaces.

Part I

Multidimensional skyline queries and moving data

In this part, we address the maintenance of NSC, an efficient indexing structure for answering subspace skyline queries, upon underlying data updates. We consider two types of moving data: (i) dynamic data in Chapter 3 and (ii) streaming data in Chapter 4. Moreover we investigate the optimization of regret minimization queries through NSC in Chapter 5.

Chapter 3

Maintenance of NSC with dynamic data

3.1 Introduction

In Section 2.2, we presented NSC the auxiliary compact data structure capable of answering skyline queries wrt any subspace. It consists in storing for each tuple t a set of pairs which summarize the subspaces where t is dominated. We have presented how (i) NSC is built, (ii) time and space optimized, and (iii) used for answering skyline queries. In [16], NSC has been shown time and space efficient compared to its competitors. However no incremental maintenance procedure has been proposed.

In the present chapter, we address NSC incremental maintenance, precisely with dynamic data, i.e tuples can be deleted/inserted at any time. Indeed, an index structure which needs to be computed from scratch each time an update occurs, is not usable. Hence, we investigate the ability of NSC to handle deletions/insertions. We came up with slight modifications of the data structure and we designed algorithms for both deletions and insertions. We show theoretically and experimentally that these modifications do not highly impact both construction and query answering times, and space consumption of NSC. Moreover we show that incrementally maintaining NSC is many folds faster than rebuilding it from scratch.

Table 3.1 provides a preview of the performance of NSC compared to its competitors, described in Chapter 1 Section 1.1.3. The higher the score of a technique S wrt a criterion c, the better is S wrt c. As it can be observed, there is no clear winner wrt to the four criteria. However, NSC seems providing a reasonable trade off.

Technique	Build time	Memory consumption	Query time	Mainte--nance time
NSC	**3**	**3**	**3**	**4**
HashCube[15]	2	2	4	1
CSC[14]	4	4	1	3
Skycube[12]	1	1	5	1
BSkyTree[8]	5	5	1	5

Table 3.1: Solutions scores

3.2 Preliminaries

In addition to the definitions and notations presented previously, Table 3.2 gives the notations used throughout this chapter.

Notation	Definition
t^+	inserted tuple
t^-	deleted tuple
Δ^+	inserted transaction
Δ^-	deleted transaction

Table 3.2: Notations

Organization The next section describes the approaches applied on NSC to handle insertions and deletions. Then we present the experiments we performed.

3.3 Managing NSC updates

In this section, we present our approach to update NSC structure after inserting/deleting either a single or a set of tuples. We first start with the insertion case which does not require any modification of the original NSC structure. Then we address the deletion which turns to be harder to deal with making us to slightly extend the NSC structure.

3.3.1 Insertions

Inserting a single tuple

When a tuple t^+ is inserted into table T, the naïve solution is to restart the computation of NSC from scratch by providing $T \cup \{t^+\}$ as input to Algorithm 1. To avoid this solution, we first identify a situation where the insertion of t^+ does not change the content of NSC.

Lemma 1. *Let* $S^+ = topmost(T \cup \{t^+\}) = Sky(T \cup \{t^+\}, \mathcal{D})$. *If* $t^+ \notin S^+$ *then* $\forall t \in T, Pairs(t,T) \equiv Pairs(t, T \cup \{t^+\})$.

The above lemma simply says that the insertion of t^+ which is $\mathcal{D}$-dominated will not change the structure of the previous $\mathrm{NSC}(T)$. All what we need to do is to compute the pairs of t^+, add the so obtained pairs to NSC and eventually compress this single set of pairs.

Algorithm 6 is the procedure we use to maintain NSC after the insertion of a single tuple. It first compares t^+ to the tuples previously belonging to the topmost (For loop in line 3). While computing the pairs of t^+(line 7), we check if t^+ is $\mathcal{D}$-dominated and identify tuples $\mathcal{D}$-dominated by t^+. If none of these comparisons show that t^+ is $\mathcal{D}$-dominated, then t^+ belongs to the new topmost, therefore every $t \in T$ needs to be compared to t^+(line 9). We compute $compare(t,t^+)$, we append it to $Pairs(t,T)$ and we compress the new set of pairs.

Algorithm 6: insertTuple

Input: T, $topmost(T)$, $\mathsf{NSC}(T)$, t^+
Output: $\mathsf{NSC}(T \cup \{t^+\})$, $topmost(T \cup \{t^+\})$

```
1  NotTopmostAnymore ← ∅
2  Pairs(t+, T ∪ {t+}) ← ∅
3  for t ∈ topmost(T) do
4      if t+ ≺_D t then
5          Add t to NotTopmostAnymore
6      else
7          Pairs(t+, T ∪ {t+}) ← Pairs(t+, T ∪ {t+}) ∪ compare(t+, t)
8  Compress(Pairs(t+, T ∪ {t+}))
9  if t+ not D-dominated then
10     for t ∈ T do
11         Pairs(t, T ∪ {t+}) ← Pairs(t, T) ∪ compare(t, t+)
12         CompressByGreedy(Pairs(t, T ∪ {t+}))
13     topmost(T ∪ {t+}) ← topmost(T) ∪ {t+} \ NotTopmostAnymore
14 return NSC(T ∪ {t+}), topmost(T ∪ {t+})
```

Example 12. *Let* t^+*=(1,1,2,2) to be inserted into Table 2.1. Recall that* $topmost(T) = \{t_2, t_3, t_4\}$. *We first compare* t^+ *to* t_2, t_3 *and* t_4 *and obtain respectively the pairs* $\langle \emptyset | ABC \rangle, \langle \emptyset | CD \rangle$ *and* $\langle CD | \emptyset \rangle$. *None of these pairs covers ABCD meaning that* t^+ *belongs to the new topmost. Note that while the first two pairs do not cover any subspace, hence they can be removed from* $Pairs(t^+)$*, they do respectively imply that* t_2 *and* t_3 *are* $\mathcal{D}$*-dominated by* t^+*, e.g.,* $compare(t^+, t_2) = \langle \emptyset | ABC \rangle \Rightarrow compare(t_2, t^+) = \langle D | ABC \rangle$. *All the remaining tuples need to be compared to* t^+*, i.e.,* t_1, t_5 *and* t_6.

Table 3.3 shows the new pairs obtained by comparing every t to t^+ beside the existing list of pairs. It also shows the computed pairs of t^+. The pairs to be kept after the compression are underlined.

Table 3.3: Updating pairs of the tuples of T

Tuple	New pair(s)	Existing pairs
t_1	$\underline{\langle CD\|AB\rangle}$	$\langle C\|ABD\rangle, \langle CD\|\emptyset\rangle$
t_2	$\underline{\langle D\|ABC\rangle}$	$\underline{\langle CD\|\emptyset\rangle}$
t_3	$\underline{\langle AB\|CD\rangle}$	$\langle AB\|C\rangle$, $\underline{\langle CD\|B\rangle}$
t_4	$\langle AB\|\emptyset\rangle$	$\underline{\langle AB\|\emptyset\rangle}$
t_5	$\langle ABC\|D\rangle$	$\underline{\langle ABC\|D\rangle}$, $\underline{\langle BCD\|\emptyset\rangle}$
t_6	$\langle ABC\|D\rangle$	$\underline{\langle ABCD\|\emptyset\rangle}$
t^+	$\langle\emptyset\|ABC\rangle, \langle\emptyset\|CD\rangle, \underline{\langle CD\|\emptyset\rangle}$	

Complexity analysis t^+ is compared to $topmost(T)$. If it is not $\mathcal{D}$-dominated then every t is compared to t^+ hence n comparisons, and every tuple calls $CompressByGreedy(Pairs(t, T \cup \{t^+\}))$ whose complexity is $O(\pi^2)$ if π denotes the number of pairs per tuple. Hence, $O(n \times \pi^2)$ operations just for the compression. Actually, *CompressByGreedy* presents two negative points: (i) it is not incremental and (ii) the polynomial complexity of the greedy procedure hides an exponential term as it is shown in Algorithm 3. **while** loop (Line 4) is iterated at most π times ($\pi = |P|$) and **for** loop (Line 9) is executed at most π times too. Hence, $O(\pi^2)$. Note however that the sets of covered subspaces we manipulate may have an exponential size w.r.t. the number of dimensions. One may wonder whether it is possible to implement *CompressByGreedy* by just operating on the pairs. It is unfortunately impossible because not every set of subspaces can be summarized by a pair.

Example 13. *Let $P = \{p_1 = \langle A|B\rangle, p_2 = \langle A|C\rangle\}$ we want to summarize with CompressByGreedy. Suppose p_1 is first chosen to be added to the solution. Now we need to update the set of still uncovered subspaces associated to p_2. The only subspace covered by p_2 and not by p_1 is AC. There exists no pair which covers AC and only AC.*

Therefore we trade the compression ratio guaranteed by *CompressByGreedy* by a less compressing procedure based just on the inclusion we detect between pairs.

Definition 8. *p_1 is included into p_2, noted $p_1 \sqsubseteq p_2$, iff the set of subspaces covered by p_1 is included into the set of subspaces covered by p_2.*

The following lemma characterizes pairs inclusion without generating their respective covered subspaces.

Lemma 2. *Let* $p_1 = \langle X_1|Y_1\rangle$ *and* $p_2 = \langle X_2|Y_2\rangle$. $p_1 \sqsubseteq p_2$ *iff* $X_1 \subseteq X_2$ *and* $X_1Y_1 \subseteq X_2Y_2$.

For example $\langle AB|C\rangle \sqsubseteq \langle ABC|D\rangle$ but $\langle AB|C\rangle \not\sqsubseteq \langle A|BCD\rangle$.

Thanks to this fast inclusion test, we propose a compression procedure whose complexity is $O(\pi^2)$ and which does need to manipulate sets of covered subspaces. It is described by Algorithm 7.

Algorithm 7: compressByInclusion

Input: Set of pairs P
Output: Set of pairs $P' \equiv P$ with $|P'| \leq |P|$

```
for p ∈ P do
    for q ∈ P and q ≠ p do
        if q ⊑ p then
            Remove q from P
Return P'
```

The following property shows that *CompressByInclusion* returns a summary that is larger than that returned by the greedy algorithm.

Property 2. *Let* $P_g = CompressByGreedy(P)$ *and* $P_i = CompressByInclusion(P)$ *for some set of pairs P. Then* $|P_g| \leq |P_i|$.

Proof. We prove that $P_g \subseteq P_i$. Let $p \in P$. Then $p \in P_i$ iff $\nexists p' \in P$ such that $p \sqsubset p'$. Now, we show that $\forall q \in P_g$ there is no q" $\in P_g$ such that $q \sqsubseteq q'$. We do this by induction on the iterations of the greedy algorithm. The base case is the first iteration where a pair q_1 covering the maximal number of subspaces is chosen. There will be no pair $q \in P_g$ covering q_1 otherwise, q is chosen first. Suppose that at iteration i, every selected pair has no so super pair in P_g. Let q_{i+1} be the selected pair at step $i+1$ then among the not already selected pairs, there remains no q such that $q_{i+1} \sqsubseteq q$ otherwise q is selected instead of q_{i+1} thus there will be no $q \in P_g$ s.t $q_{i+1} \sqsubseteq q$ which concludes the proof. □

Besides its lower complexity, compressing by inclusion test is amenable to an incremental implementation by contrast to the greedy algorithm. Indeed, adding a new pair p to an already compressed set of pairs P can be done by just comparing p to the elements of P leading to a linear procedure while, to the best of our knowledge, there is no incremental version of the greedy algorithm.

Distinct values property

Interestingly, when all tuples have distinct values on every dimension, Algorithm 7 is not only as good as the *CompressByGreedy* algorithm, but it returns the optimal solution.

Proposition 3. *Let P be a set of pairs s.t $\forall \langle X|Y\rangle \in P$, $Y = \emptyset$. Let $P' = CompressByInclusion(P)$. Then, for every Q, $Q \equiv P \Rightarrow |Q| \geq |P'|$.*

Proof. The maximal subspace covered by a pair $p = \langle X|\emptyset\rangle$ is X. For p to be removed from P, there should be another pair $p' = \langle X'|\emptyset\rangle$ covering X. Thus, X should be included into X' meaning that $p \sqsubseteq p'$. □

When T satisfies the distinct values property, i.e., $\forall t_1, t_2 \in T$ and $\forall d_i \in \mathcal{D}$ we have $t_1[d_i] \neq t_2[d_i]$, the pairs we obtain satisfy the condition of Proposition 3. Thus, the compression of *CompressByInclusion* is optimal in this case.

Inserting a set of tuples

In the case we have a set of tuples Δ^+ to be inserted, we can iterate Algorithm 6 over each $t^+ \in \Delta^+$. Some of the comparisons we do with this technique can be avoided. For example, let $\Delta^+ = \{t_1^+, t_2^+\}$ such that $t_2^+ \prec_{\mathcal{D}} t_1^+$. Clearly, comparing the tuples with t_1^+ is useless w.r.t. skyline semantics as stated by the following lemma.

Lemma 3. *Let $t \in T$ and $\{t_1^+, t_2^+\} \subseteq \Delta^+$, $t_2^+ \prec_{\mathcal{D}} t_1^+ \Rightarrow compare(t, t_1^+) \sqsubseteq compare(t, t_2^+)$*

Algorithm 8 takes benefit from the previous lemma. First, it computes the new *tompost* by considering not the whole previous T but just its *topmost*(line 1). Then every $t \in T$ is compared with each $t^+ \in \Delta^+$ which belongs to the new topmost (**Forall** loop, lines 2-5). The new set of pairs is compressed (line 5). Then, every $t^+ \in \Delta^+$ is compared to the elements of the new topmost and every set of pairs is compressed (**Forall** loop, line 6-9).

Note that the compression procedure can be implemented either via *CompressByGreedy* or *CompressByInclusion*.

3.3.2 Deletions

Likewise insertions, we consider single and set oriented deletions separately.

Deleting a single tuple

The impact on a tuple t when deleting a tuple t^- is that the set of subspaces where t is dominated can decrease. In NSC structure, this set of subspaces is encoded by the pairs

Algorithm 8: batchInsertSetOfTuples

Input: T, $topmost(T)$, $\mathsf{NSC}(T)$, Δ^+
Output: $\mathsf{NSC}(T \cup \Delta^+)$, $topmost(T \cup \Delta^+)$

```
1  NewTopmost ← Sky(Δ⁺ ∪ topmost(T)), 𝒟)
2  forall t ∈ T do
3      for t⁺ ∈ (Δ⁺ ∩ NewTopmost) do
4          Pairs(t) ← Pairs(t) ∪ compare(t, t⁺)
5      Compress(Pairs(t))
6  forall t⁺ ∈ Δ⁺ do
7      for t ∈ NewTopmost do
8          Pairs(t⁺)) ← Pairs(t⁺) ∪ compare(t⁺, t)
9      Compress(Pairs(t⁺))
10 topmost(T ∪ Δ⁺) ← NewTopmost
11 return NSC(T ∪ Δ⁺), topmost(T ∪ Δ⁺)
```

associated to t. The brute force approach to maintain this set of pairs is to rebuild it from scratch, i.e., executing Algorithm 4 by providing $T \setminus \{t^-\}$ as input parameter.

In this section, we identify some properties which allow us to avoid this heavy computation. We start by observing that when $t^- \notin topmost$ then we do not need to recompute NSC.

Lemma 4. *if* $t^- \notin topmost(T)$, *then* $\forall t \in T, Pairs(t,T) \equiv Pairs(t, T \setminus \{t^-\})$

This comes from the fact that the tuples are compared just to $topmost(T)$.

Suppose all tuples have the same probability to be deleted, then the probability to not to have to update NSC when a tuple t^- deleted, is greater than $1 - \frac{|topmost(T)|}{|T|}$.

The following example illustrates other situations where the whole NSC maintenance is note required. Indeed, we identify tuples whose associated pairs need not to be recomputed.

Example 14. *Table 3.4 depicts the* NSC *associated to the running example where we add to each pair, the tuple(s) used to obtain it. Recall that* $topmost(T) = \{t_2, t_3, t_4\}$. *The existing pairs depend only on these tuples, which means that the deletion of e.g.,* t_1 *has no impact on the other tuples.*

Note that deleting the topmost tuple t_3 *has no impact on e.g.,* t_6 *since there is no pair associated to* t_6 *and obtained from* t_3. *Moreover, deleting* t_3 *has no impact on* t_1 *either. This is because* $\langle CD|\emptyset\rangle$ *will still be associated to* t_1 *via tuple* t_4.

From the example above, we see that the deletion of a tuple t^- may impact a tuple t if t^- contributes effectively to derive the subspaces where t is dominated. This is formalized by the following proposition.

Table 3.4: List of pairs synthesizing dominance subspaces sets

Tuple	Associated list of pairs
t_1	$(t_2, \langle C\|ABD\rangle), (\{t_3, t_4\}\langle CD\|\emptyset\rangle)$
t_2	$(t_4, \langle CD\|\emptyset\rangle)$
t_3	$(t_2, \langle AB\|C\rangle, (t_4, \langle CD\|B\rangle)$
t_4	$(t_2, \langle AB\|\emptyset\rangle)$
t_5	$(t_3, \langle ABC\|D\rangle), (t_4, \langle BCD\|\emptyset\rangle)$
t_6	$(t_4, \langle ABCD\|\emptyset\rangle)$

Proposition 4. *Let $t \in T$ and $t^- \in topmost(T)$ be the tuple to be deleted. Let P, resp. P^-, be the compressed list of pairs associated to t in $\mathsf{NSC}(T)$, resp. in $\mathsf{NSC}(T \setminus \{t^-\})$. The following implications hold:*

1. $compare(t,t^-) \notin P \Rightarrow P \equiv P^-$.
2. $compare(t,t^-) \in P$ *and* $\exists t' \in topmost(T)$ *s.t* $compare(t,t^-) = compare(t,t') \Rightarrow P \equiv P'$

Proof.
1. If $compare(t,t^-) \notin P$ then all subspaces covered by $compare(t,t^-)$ are also covered by P. Hence, by removing t^-, these subspaces remain covered.
2. All subspaces covered by $compare(t,t^-)$ remain covered thanks to other tuples t' in topmost such that $compare(t,t^-) = compare(t,t')$.

□

It follows that for a tuple $t \in T$, its set of subspaces where it is dominated may change only if the tuple to be deleted t^- is the unique tuple producing a pair p^-, i.e., $compare(t,t^-) = p^-$, and p^- belongs to $Pairs(t,T)$.

Example 15. *From Table 3.4, for t_1, the deletion of either t_3 or t_4 has no impact as the pair $\langle CD|\emptyset\rangle$ is produced by both of them. However $\langle C|ABD\rangle$ is produced uniquely by t_2. Thus, deleting t_2 may have an impact on t_1 and actually it does.*

To take advantage from the properties stated in Proposition 4, we extend the NSC structure by associating a counter to every pair $p \in Pairs(t,T)$. This counter represents the number of tuples which contribute to this pair.

Example 16. *The* NSC *in Table 3.4 is actually stored as follows.*

This additional information increases linearly NSC memory consumption. Indeed, instead of using pairs as physical memory units, we rather store triples of the form $\langle X|Y|counter\rangle$. The number of memory units is kept unchanged. To not disturb the

Table 3.5: Pairs with counters

Tuple	Associated list of pairs
t_1	$(1,\langle C\|ABD\rangle),(2\langle CD\|\emptyset\rangle)$
t_2	$(1,\langle CD\|\emptyset\rangle)$
t_3	$(1,\langle AB\|C\rangle),(1,\langle CD\|B\rangle)$
t_4	$(1,\langle AB\|\emptyset\rangle)$
t_5	$(1,\langle ABC\|D\rangle)$, $(1,\langle BCD\|\emptyset\rangle)$
t_6	$(1,\langle ABCD\|\emptyset\rangle)$

reader, in the remaining sections, we shall still use the term *pair* to designate the basic stored information.

The construction of the new structure of NSC follows slightly the same procedure as in Algorithm 1. For completeness, we describe it in Algorithm 9.

Algorithm 9: buildNSC_with_counters

Input: T, $topmost(T)$
Output: NSC(T)

```
1  Pair p
2  for t ∈ T do
3      for t' ∈ topmost(T) do
4          p ← compare(t, t')
5          if p ∈ Pairs(t, T) then
6              p.counter ← p.counter + 1
7          else
8              p.counter ← 1
9              Pairs(t, T) ← Pairs(t, T) ∪ {p}
10     Compress(Pairs(t))
11 return NSC(T)
```

Remark: The main difference between Algorithms 9 and 1 is the membership test (Line 5). Actually, this can be done in $O(1)$ by organizing the set of pairs as hash table. Thus, the new algorithm adds little overhead w.r.t. the original one. Furthermore, the insertion algorithms presented so far are adapted accordingly to cope with the counters associated to the pairs.

Now we are ready to present our approach to maintain NSC after a single tuple deletion. It is described in Algorithm 10 and we illustrate its execution with various t^- in the following example.

Example 17. *Let $t^- = t_1$, t^- is not in $topmost(T)$ so the deletion of t^- requires no change. We only delete it from the dataset.*

Algorithm 10: deleteTuple

Input: t^-,T,NSC(T),$topmost(T)$
Output: NSC($T \setminus \{t^-\}$), topmost($T \setminus \{t^-\}$)

```
1  if t⁻ ∈ topmost(T) then
2      topmost(T \ {t⁻}) ← Sky(T \ {t⁻}, 𝒟)
3      for t ∈ (T \ {t⁻}) do
4          p ← compare(t, t⁻)
5          if p ∈ Pairs(t, T) then
6              if p.counter = 1 then
7                  Pairs(t) ← ∅
8                  for t' ∈ topmost(T \ {t⁻}) do
9                      q ← compare(t, t')
10                     if q ∈ Pairs(t) then
11                         q.counter ← q.counter + 1
12                     else
13                         q.counter = 1
14                         Add q to Pairs(t)
15                 Compress(Pairs(t))
16             else
17                 p.counter ← p.counter − 1
18 return NSC(T \ {t⁻}), topmost(T \ {t⁻});
```

Now let $t^- = t_2$, and we use the initial dataset. t^- belongs to $topmost(T)$ so we need to compute $topmost(T \setminus \{t^-\}) = \{t_1, t_3, t_4\}$. Every tuple t needs to be compared to t^-. It turns that t^- impacts t_1 because $compare(t_1, t_2) = \langle C|ABD\rangle$ and (i) this pair is in $Pairs(t_1)$ and (ii) its counter is set to 1. Therefore, $Pairs(t_1)$ needs to be recomputed. The same situation holds for t_3 and t_4 whose respective pairs need to be recomputed. Note however that t_2 does not impact neither t_5 nor t_6.

Deleting a set of tuples

When deleting a subset $\Delta^- \subset T$, one solution could be to call Algorithm 10 for each $t^- \in \Delta^-$. One problem with this procedure is that the set of pairs associated to a tuple t is computed as many times as there are tuples $t^- \in \Delta^-$ which affect it. Moreover, a new topmost skyline is computed for every t^- belonging to the previous topmost. Therefore, we propose a batch procedure to avoid the above limitations. Our solution is described in Algorithm 11. It first checks whether there is a deleted tuple belonging to the present topmost (line 1). In this case, the new topmost is computed (line 3). Then for every t remaining in T, it checks whether it is impacted by Δ^-, i.e., there is a pair $p \in Pairs(t, T)$

having its counter set to 0 (line 10). In this case, a new set $Pairs(t, T \setminus \Delta^-)$ is computed by comparing t to all the elements of the new topmost and compressing the so obtained set of pairs (line 12-21).

Algorithm 11: batchDeleteSetOfTuples

Input: Δ^-,T,NSC(T),$topmost(T)$
Output: NSC($T \setminus \Delta^-$), topmost($T \setminus \Delta^-$)

```
1  TopDel ← Δ⁻ ∩ topmost(T)
2  if TopDel ≠ ∅ then
3      topmost(T \ Δ⁻) ← Sky(T \ Δ⁻, 𝒟);
4      for t⁻ ∈ TopDel do
5          //Find the impacted tuples
6          for t ∈ T \ Δ⁻ do
7              p ← compare(t, t⁻)
8              if p ∈ Pairs(t, T) then
9                  p.counter ← p.counter − 1
10                 if p.counter = 0 then
11                     Add t to ImpactedTuples
12     for t ∈ ImpactedTuples do
13         //Recompute the pairs of impacted tuples
14         for t' ∈ topmost(T \ Δ⁻) do
15             p ← compare(t, t')
16             if p ∈ Pairs(t, T \ Δ⁻) then
17                 p.counter ← p.counter + 1
18             else
19                 p.counter ← 1
20                 Pairs(t, T \ Δ⁻) ← Pairs(t, T \ Δ⁻) ∪ p
21         Compress(Pairs(t))
22 return NSC(T \ Δ⁻), topmost(T \ Δ⁻);
```

Complexity analysis

The first parameter affecting the deletion procedure is the probability that the set Δ^- intersects the topmost, i.e., at least one of the deleted tuples belongs to the topmost. Recall that if the intersection is empty, the only thing to do is to remove the deleted tuples. Under a uniform hypothesis for deleting any tuple, the probability that the intersection is not empty by $|\Delta^-| \times \frac{|topmost|}{|T|}$. Hence, for a fixed $|\Delta^-|$, the larger the topmost, the larger is this probability.

The second parameter is the number of impacted tuples by $\Delta^- \cap topmost$. There are two extreme cases: the one where no tuple is impacted and the other where all tuples

are impacted. The later worst case makes our algorithm degenerate to the naïve solution since we need to compare all tuples to the new topmost. The formal analysis of the average number of affected tuples by Δ^- is hard, if not impossible, because we need to estimate the probability that a pair is (i) not removed due to the compression process and (ii) all the tuples used to obtain it belong to Δ^-. In Section 3.4 we empirically analyze this behavior and we will see that, e.g., the number of impacted tuples by a topmost one is far from being uniform.

An interesting situation is that of correlated data where the size of the topmost is small, possibly equal to one. In this case, all tuples are compared to the unique tuple of the topmost. Deleting the later will impact all the remaining tuples. However, the probability of deleting this specific tuple is $\frac{1}{|T|}$ which is rather small.

3.4 Experiments

In this section we present the comparative experimental results we obtain with NSC and its principal competitors. For this, we consider the following four criteria: (i) construction time, (ii) memory consumption, (iii) skyline query processing time and (iv) maintenance upon updates. All the implementations we used but CSC are those provided by their respective authors.

First we compare NSC to CSC, HashCube and QSkyCube wrt construction time and space consumption. QSkyCube builds the whole skycube so its output is considered as a baseline to assess the compression ratio of each solution.

As for query evaluation, we use CSC, HashCube and BSkyTree. The later is the baseline since it is the state of the art algorithm for skyline evaluation when no precomputation is available. So it serves to evaluate the optimization ratio provided by each technique against a solution where no extra storage/computation is performed.

Regarding updates, to our knowledge, the only structure for which incremental maintenance algorithms have been provided is CSC. Because, as we will show, it has poor query performance compared to its "materialization based" competitors, i.e. HashCube and NSC, we decided to not include it in this part of the experiments.

Datasets

For the purpose of evaluating NSC, we perform experiments on both real and synthetic datasets. For real datasets, we use commonly cited datasets in skyline literature. In table 3.6 we present the datasets, their cardinality n, the cardinality of the set of dimensions d, and the size of the *topmost*. For synthetic datasets, we generate data through the

framework presented in [2], with different distributions (Independent (INDE), Correlated (CORR), Anti-correlated (ANTI)). For each, we vary n in $\{10^4, 10^5, 10^6\}$ and d in $\{4, 8, 12, 16, 20\}$.

Dataset	d	n	$\|topmost\|$
NBA	17	20493	3
MBL	18	92797	78
IPUMS	10	75836	3852
HOUSE	6	127931	127931
INSEE	22	2628433	58
POKER	11	1000000	14131

Table 3.6: Real datasets

Hardware and implementation

All the experiments are conducted on a Linux machine equipped with two 2.6 ghz hexacore CPUs and 32GB RAM. We implemented NSC as well as CSC in c++ together with OpenMP library to parallelize some parts of the algorithms. CSC proceeds levelwise and each subspace of one level is treated independently of the others of the same level, so they can all be processed in parallel. As for NSC, every tuple is processed independently so this is the parallel granularity we used for it. To test HashCube, BSkyTree and QSkyCube we used their respective authors versions which are in c++ too. HashCube implements several algorithms to compute the hashcube structure. As it is shown in [51], *mdmc* is the most efficient so it is the one we use. *mdmc* can share computation on both CPU and GPU. For the present experimentation, only CPU is used. NSC Source code is available on GitHub[1]. This repository contains as well CSC and BSkyTree implementations. HashCube and QSkyCube implementations are available on GitHub as well [2].

In the remainder, a missing value means that the respective solution could not terminate either because of memory saturation or excessive execution time (stopped after 24 hours).

3.4.1 Constructing the structures

The aim of this experiment is to compare NSC structure to (i) CSC, (ii) HashCube and (iii) QSkyCube wrt construction time and space consumption. We evaluate the behavior

[1] https://github.com/karimalami7/NSC
[2] https://github.com/sean-chester/skycube-templates

of these techniques by varying n and d.

Regarding memory consumption, we report the number of memory units used by each structure: for NSC we count the total number of pairs. CSC and skycube store for each tuple t respectively the smallest and all subspaces where t belongs to the skyline, hence for both of them we count the total number of subspaces that need to be stored. HashCube stores for each word ω encoding a set of subspaces, a list of tuples that share this word. Recall that the same tuple may be associated to several words. So, we count the total number of tuples that need to be stored. Physically, each storage unit used by CSC or the skycube corresponds to a subspace which can be encoded by a Boolean vector of size d. A pair of subspaces, as used by NSC can be encoded by a $2d$ vector. Regarding HashCube, a tuple Id can be encoded by a $log(n)$ bit vector. The values of n used in our experiments are sufficiently large to make $\log(n) \geq d$. Hence, by counting the number of bit vectors used by HashCube, the comparison to NSC or CSC is fair. It is worth to notice that because QSkyCube cannot terminate its execution in some situations, e.g., $d = 16, n = 10^5$ and anticorrelated data, we obtained the size of the skycube by just evaluating all skyline queries and summing their respective sizes.

Figures 3.1, 3.2, 3.3 and 3.4 show the results we obtain with respectively independent, correlated, anticorrelated and real datasets.

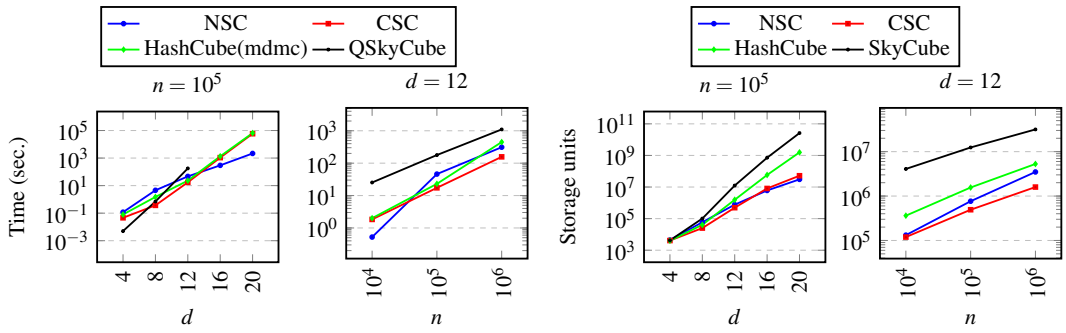
Synthetic data

With respect to construction time, all techniques have almost the same behavior wrt varying n even though QSkyCube is in general the slower. When varying d, we observe an advantage for NSC when d increases, e.g. in Figure3.1(a), NSC construction time grows by a factor smaller than 10 between both $d = 12$ and $d = 16$, and $d = 16$ and $d = 20$ while for CSC and HashCube the factor is approximatively 60. Regarding QSkyCube, it is performing well with small d, e.g. $d = 4$, however it does not terminate in a reasonable time or saturates the available memory, e.g., with $d = 16$ for independent and anticorrelated datasets Fig. 3.1 Fig. 3.3, and even with correlated data when $d = 20$ Fig. 3.2.

With respect to memory consumption, we observe the same behavior for all techniques when varying n, i.e., a *linear* increase wrt n. By varying d, we note that NSC and CSC scale better with large d whatever is the data correlation type. For example, when $d = 20$, NSC uses about $100\times$ less memory than HashCube and even $1600\times$ with correlated data (Fig. 3.1(b),3.2(b) and 3.3(b)).

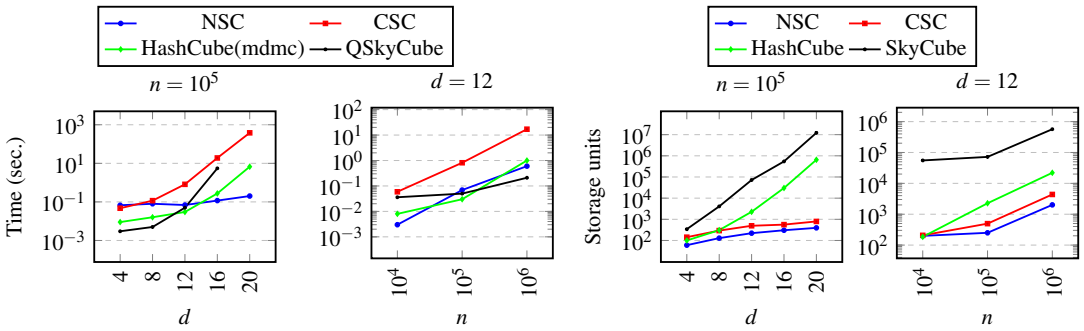
Remark: When d is set to 20, the larger NSC is obtained with anti-correlated data and $n = 10^5$ (Figure 3.3(b)). In that case, the average number of pairs per tuple is 740. By

contrast, the average number of subspaces where a tuple belongs to the skyline is $\simeq 4 \cdot 10^5$. So, for the same information NSC stores $\frac{4 \cdot 10^5}{740} \simeq 540$ less memory than the skycube.



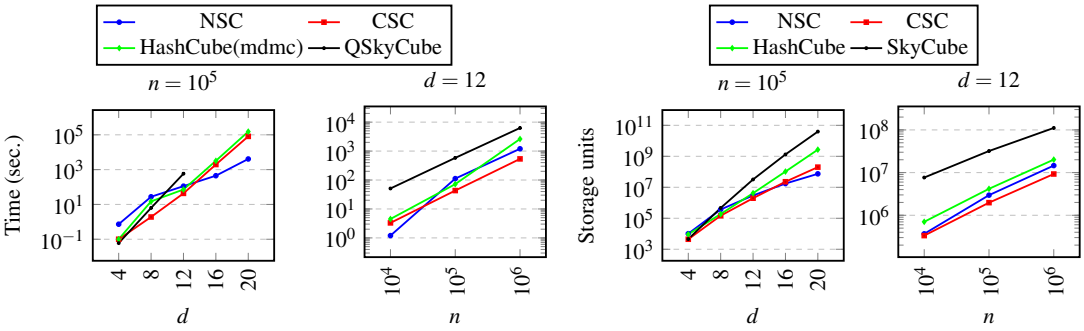
(a) Build time (b) Memory consumption

Figure 3.1: Build time and memory consumption with independent data



(a) Build time (b) Memory consumption

Figure 3.2: Build time and memory consumption with correlated data



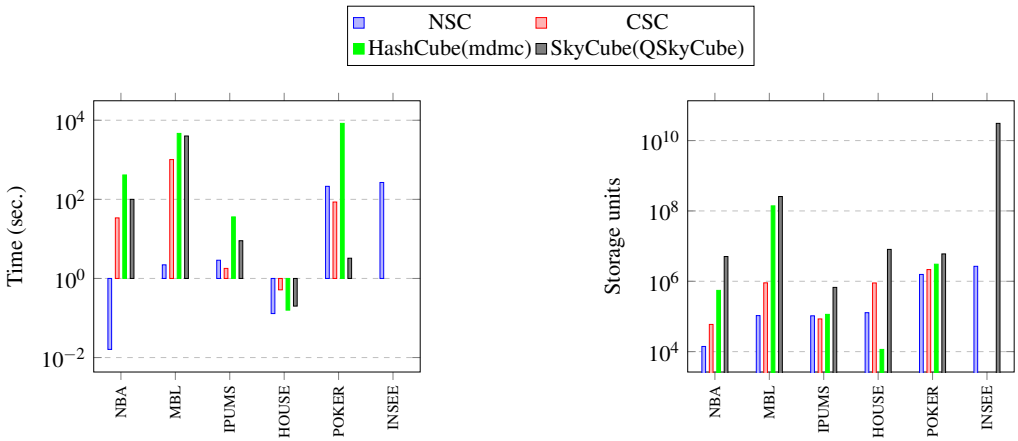
(a) Build time (b) Memory consumption

Figure 3.3: Build time and memory consumption with anticorrelated data

Real data

The same experiments as above have been performed on real data sets so that to avoid the biases introduced by the (non) correlation of synthetic data. The results are depicted on Figure 3.4. These data sets exhibit different configurations wrt n and d as well as the size of their respective topmost skylines. In terms of build time, we observe that in general, NSC is the fastest. It is worthwhile to notice that only NSC is able to process INSEE data set. The other algorithms were stopped either due to memory saturation (CSC and QSkyCube) or excessive time (after 24 hours for HashCube).

Regarding the memory consumption, globally, NSC requires less storage space. IPUMS and HOUSE are the exception: IPUMS has a small n and for HOUSE, d is rather small. Here too, and for the sake of assessing the compression ratio, we report the size of the skycube of every data set even if QSkyCube did not terminate.



(a) Build time (b) Memory consumption

Figure 3.4: Build time and memory consumption with real data

3.4.2 Answering skyline queries

For this experiment we compare the performance of NSC to those of (i) CSC, (ii) HashCube and (iii) BSkyTree wrt query answering execution time. The first three methods use pre-computation while the latter evaluate skyline queries directly from row data.

For each structure we evaluate *all* possible skyline queries, i.e., the $2^d - 1$ queries, and report the total execution time. We do so to avoid the impact of dimensionality. Moreover, this total time divided by the total number of queries gives an idea about the average query execution time. Here too, we vary n, d and correlation.

Synthetic data

Figures 3.5, 3.6 and 3.7 depict the results we obtained with independent, correlated and anticorrelated data.

Globally, with respect to d, NSC and HashCube have the same performance and outperform CSC and BSkyTree by more than two orders of magnitude. However HashCube scales remarkably better with increasing n. Its query answering time is almost constant. This shows that the number of distinct words used by HashCube remain almost constant when d is fixed.

Interestingly, when d and n are relatively small, BSkyTree, i.e., no materialization, seems to be the best solution. Indeed, with anticorrelated data and $d = 4$ or even $d = 8$, answering a single skyline query takes less than half a second when $n = 10^5$.

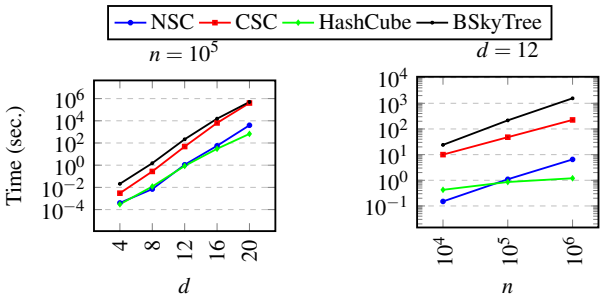


Figure 3.5: Query answering with independent data

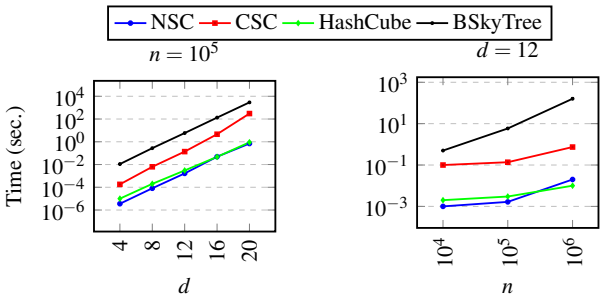


Figure 3.6: Query answering with correlated data

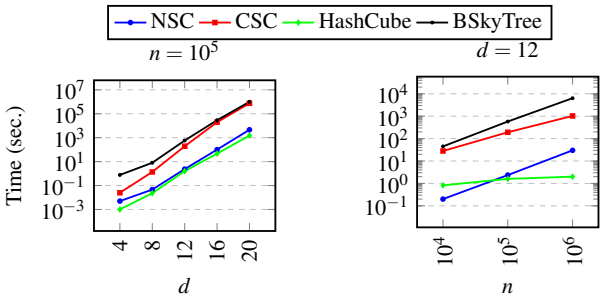


Figure 3.7: Query answering with anticorrelated data

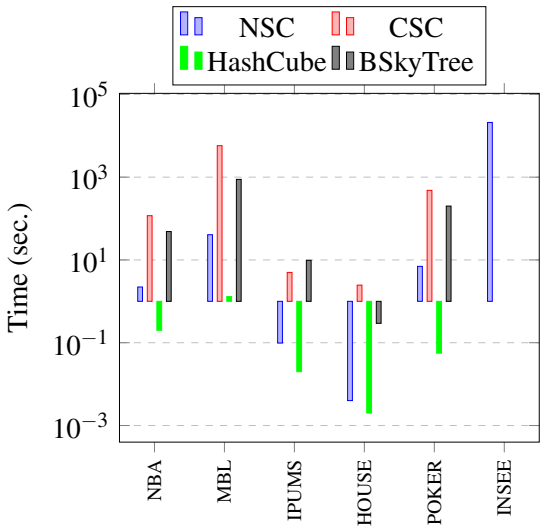


Figure 3.8: Query answering with real data

Real data

The obtained results are shown in Figure 3.8. The first noticeable remark is that in most cases, CSC is slower than BSkyTree which makes it definitively not a viable solution. The second observation is that HashCube is always the best solution. The only exception is with INSEE dataset where the HashCube itself cannot be constructed.

3.4.3 Maintenance upon updates

The aim of this section is to assess the effectiveness of the proposed solutions to maintain the NSC structure upon updates. We compare our proposals of incrementally updating the structure against the process of rebuilding the structure from scratch.

Evaluating insertions

Compression procedures In Section 3.3.1, we have presented the incremental compressing procedure *CompressByInclusion* as an alternative to *CompressByGreedy*. This first experiment consists in analyzing the memory increase versus the execution time decrease we obtain when using the compression procedure based on pairs inclusions rather than the greedy algorithm. To the sake of completeness, we also consider the case where no compression is used. We present the results we obtained with an independent data set with $n = 10^5$, build its NSC using the greedy algorithm then we evaluate the effect of inserting a set of tuples[3]. We repeat the experiment by varying d and $|\Delta^+|$. For both

[3]We emphasize the fact that we performed the same experiment with correlated and anti-correlated data and we obtained similar results.

memory usage and execution time, we consider greedy as the reference. More precisely, Figure 3.9 depicts the values taken by the formula

$$\frac{\text{Size of the new NSC} - \text{Size of initial NSC}}{\text{Size of new NSC with greedy} - \text{Size of initial NSC}}$$

Intuitively, this formula represents the loss ratio in compression when we use *CompressByInclusion* or *NoCompression* over *CompressByGreedy*. For the execution time, Figure 3.10 shows the speed up of an execution method (without compression and *CompressByInclusion*) over *CompressByGreedy*, i.e.,

$$\frac{\text{Time to get the new NSC with Greedy}}{\text{Time to to get the new NSC with other}}$$

We observe that, the size of NSC obtained with *CompressByInclusion* gets closer to that returned by *CompressByGreedy* when d increases. By contrast, *CompressByInclusion* speedup gets higher. We also observe that for a fixed d, the speedup decreases wrt $|\Delta^+|$.

To conclude, these experiments show that summarization is worthwhile (it divides NSC size by a factor of almost 100). *CompressByInclusion* does provide an interesting trade off between execution time and memory usage. Therefore, *CompressByInclusion* is the procedure we use in the next experiments.

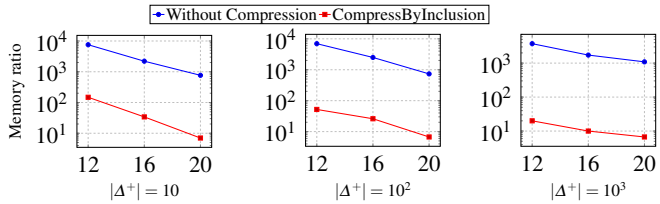


Figure 3.9: Memory growth ratio: varying d and $|\Delta^+|$

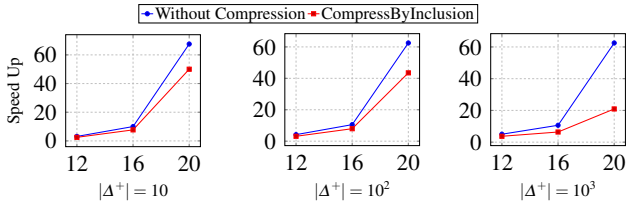


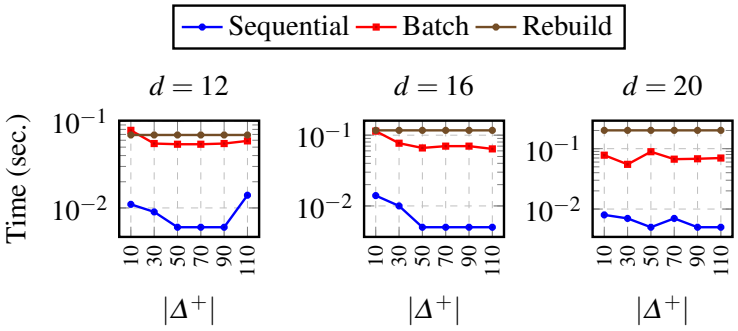
Figure 3.10: Speedup evolution ratio varying d and $|\Delta^+|$

Execution time analysis In this section we analyze the insertion methods presented in Section 3.3.1. We focus on the insertion of a set of tuples by considering both batch and sequential procedures. The later provides also information about single tuple insertion since it consists in just iterating single insertions. Therefore, we do not report on the execution times we get when a single tuple is inserted.

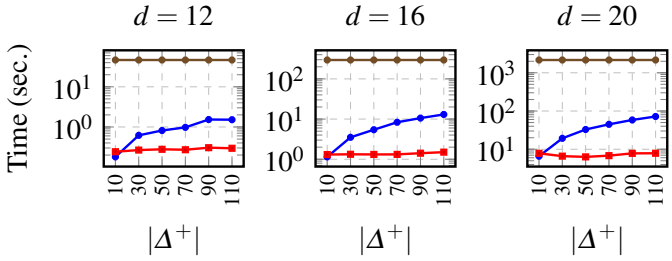
We suppose that NSC is already built for a data set T and we generate a set of tuples Δ^+ with the same correlation type that we append to T.

To be sure that the inserted tuples imply effective update of NSC we select them in such a way that they will be part of the new topmost skyline. More precisely, when $|\Delta^+|$ is set to 10, we keep generating new tuples until we get 10 that are not dominated on D by any of the previous topmost tuples.

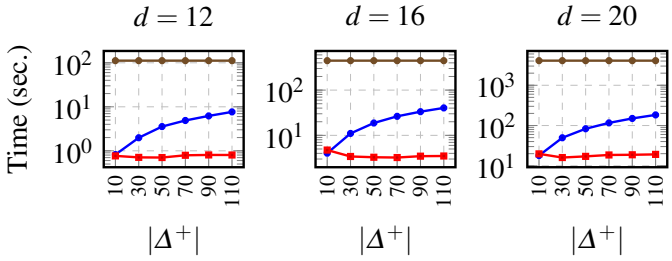
Figures 3.11 and 3.12 plot the results of inserting Δ^+ of sizes (10,30,50,70,90,110), with respectively varying d and n.



(a) CORR



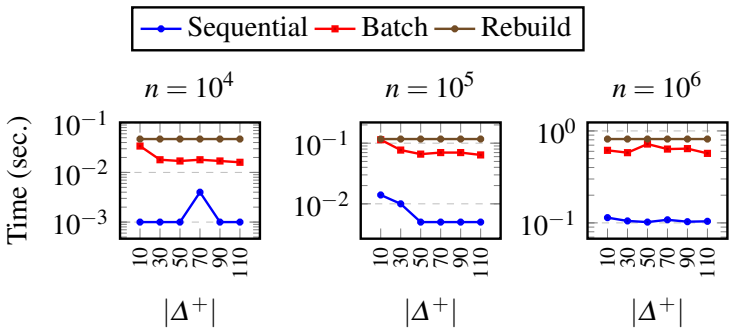
(b) INDE



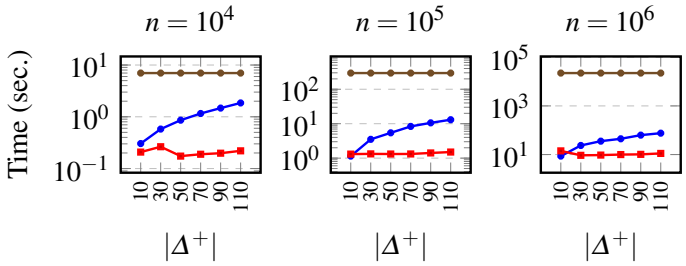
(c) ANTI

Figure 3.11: Small Δ^+ insertion by varying d ($n = 10^5$)

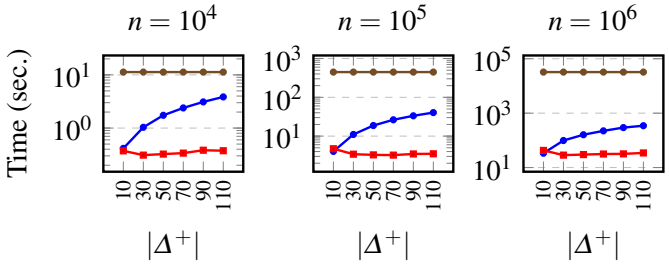
Sequential and Batch insertion methods are faster than rebuilding NSC from scratch in all cases. Note however that when $|\Delta^+|$ is quite small (typically, 10) the sequential procedure is better than the batch one. Recall that the former consists simply in iterating the insertion of a single tuple over Δ^+. This shows that the batch method is worthwhile when Δ^+ gets large (typically, more than 10). An exception to this behavior is the case of



(a) CORR



(b) INDE



(c) ANTI

Figure 3.12: Small Δ^+ insertion by varying n ($d = 14$)

correlated data where even rebuilding NSC from scratch is a viable solution since it takes about $100msec$.

We also performed some experiments to see whether our incremental solutions degenerate to the naïve solution, i.e., build from scratch, when the amount of inserted tuples is large. Figure 3.13 shows the results for an independent data set with $n = 10^6$ and $d \in \{8, 12, 16, 20\}$. Observe that the execution time of the batch method seems to be linear w.r.t. the size of Δ^+. Its speed up w.r.t. the rebuild procedure is correlated with the inserted tuples ratio independently of the dimensionality even if we observe a slight gain when d increases. More precisely, when $|\Delta^+| = 10^3 = 0.1\% \times |T|$ the speed up $Time(Rebuild)/Time(Batch) \simeq 1000$ when $d = 16$ and it falls to about 10 when $|\Delta^+| = 10^5 = 10\% \times |T|$. So even with large insertions, our solution is still competitive compared to the rebuild method.

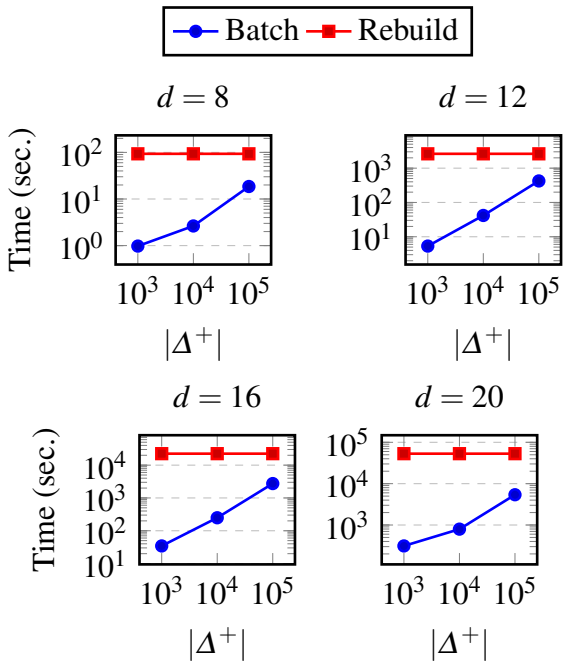


Figure 3.13: Inserting large Δ^+ by varying d ($n = 10^6$)

Evaluating deletions

Impact analysis As we have seen, an important parameter influencing the efficiency of handling the deletion of a tuple t^- is the number of tuples it impacts, i.e., those for which we need to recompute their associated new set of pairs. We conduct some experiments to analyze the distribution of this parameter. To this aim, we compute the NSC associated to a table T, then for every tuple $t \in topmost$, we compute the number of tuples it impacts whenever t is to be deleted. Note that we do not consider tuples not belonging to $topmost$ since they have no impact. For the three types of data (CORR, INDE and ANTI), we generate a data set with $n = 10^5$ and $d = 16$. The characteristics of these data are depicted below. #max represents the maximal number of impacted tuples by an element of the topmost.

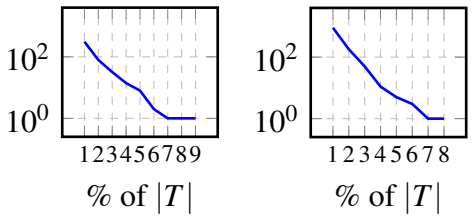
CORR		INDE		ANTI	
$\|topmost\|$	#max	$\|topmost\|$	#max	$\|topmost\|$	#max
26	4	63091	9916	97847	8462

As it can be noted, the maximal number of tuples impacted by a deletion represents a small portion of the dataset whatever is the correlation nature of data. Interestingly, this number is larger for anti-correlated than independent data. This can be explained by the fact that in the former case, the tuples tend to belong to more skylines hence, they are dominated in less subspaces. Therefore, less pairs are needed to summarize them. By contrast, when the dimensions are correlated, the topmost is small thus many tuples are

totally dominated by most tuples in this small set. In consequence, very few pairs are associated to a single topmost tuples.

Moreover, not all topmost tuples have an impact on T, e.g. for independent data, only 28730 (about 45% of the topmost) are impacting at least one tuple and 68213 for anti-correlated data (about 70% of the topmost) which represent respectively 28 and 68% of T.

We are also interested by the distribution of this number of impacted tuples among the elements of topmost. To this aim, for each data set and each $X\%$ ratio of the input table T ($X\% = 1\%, 2\%, \ldots, \frac{\#max}{|T|}\%$), we compute the number of topmost tuples impacting *more* than $X\%$ of T. Figure 3.14 depicts these results. We observe that most topmost tuples impact very few tuples. Said another way, the probability that deleting a tuple, or even a set of tuples, will incur a large amount of work is quite small.



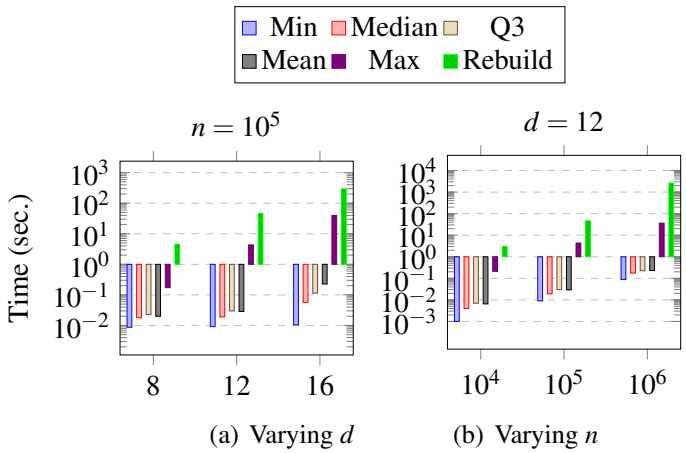
(a) INDE (b) ANTI

Figure 3.14: Evaluating impact with $n = 10^5$ and $d = 16$

To make this observation more concrete, we conducted an experiment to collect some statistics about the execution time required to maintain NSC upon deleting a topmost tuple. We report the min, max, mean, median and Q3 (third quartile) execution times and we contrast these values with the time required to build NSC from scratch. Figure 3.15 shows the results we obtained with an independent dataset by varying both n and d. We observe, among others, that in most cases, half of the topmost tuples (median) need an execution time which is about two orders of magnitude lower than that for rebuilding NSC.

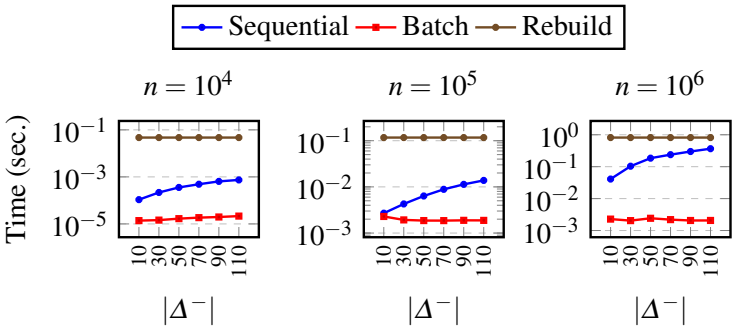
Execution time analysis We investigate also the execution time of updating NSC upon a deletion of a subset $\Delta^- \subset T$. We reiterate the operation with different Δ^- of increasing size $\{10, 30, 50, 70, 90, 110\}$. Figures 3.16 and 3.17 show the results of maintaining NSC upon deleting Δ^-, respectively, by varying n and by varying d.

Sequential and Batch methods overtake rebuilding NSC in all the experiments. The gap is even larger when n increases, e.g. in Figure 3.16 with $d = 16$ and $n = 10^6$, for both independent and anti-correlated datasets, the gain is at least 100. However, Batch

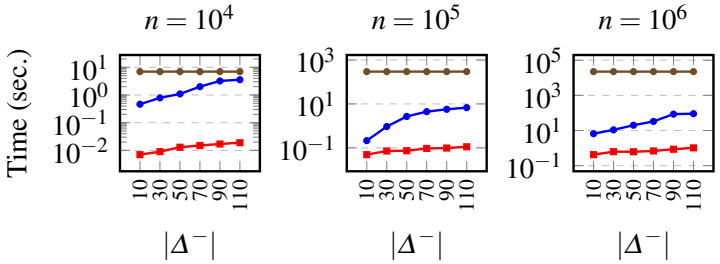


(a) Varying d (b) Varying n

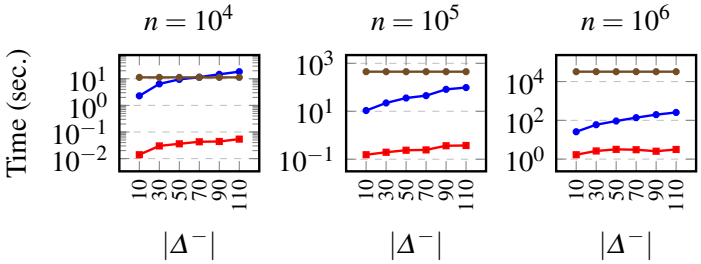
Figure 3.15: Deletion time of topmost tuples



(a) CORR



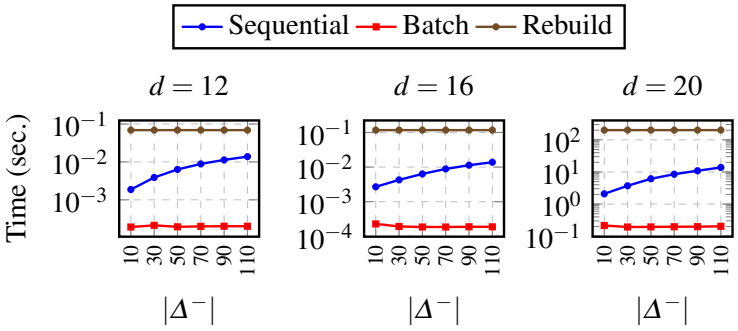
(b) INDE



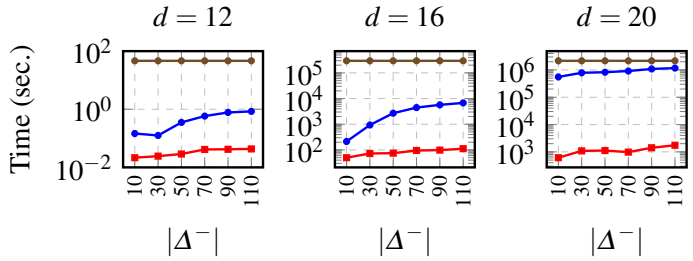
(c) ANTI

Figure 3.16: Small Δ^- deletion by varying n ($d = 16$)

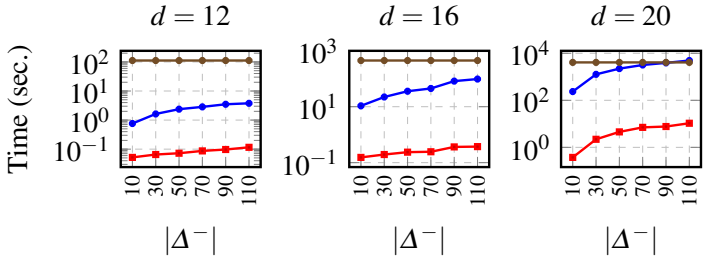
tends to be better with larger Δ^-, due to the fast evolution of sequential execution time, e.g. in Figure 3.16, for anti-correlated data with $d = 16$ and $n = 10^4$, sequential deletion



(a) CORR



(b) INDE



(c) ANTI

Figure 3.17: Small Δ^- deletion by varying d ($n = 10^5$)

oversteps rebuilding NSC from scratch.

To push even more those experiments, we delete until 10% of an initial data of 10^6 tuples and 20 dimensions. The previous experiments have already shown that sequential is not scalable w.r.t. $|\Delta^-|$. Therefore, we compare only Batch with rebuilding NSC. Figure 3.18 shows the results with an independent dataset. We observe that Batch outperforms rebuilding NSC for all configurations. We remark also that the more dimensions we add, the higher the gap between Batch and Rebuild when deleting 10^5 tuples. The main reason is that when $d = 8$, topmost is smaller than with $d = 20$, which makes a large portion of topmost included into Δ^-, and this leads to a high number of impacted tuples.

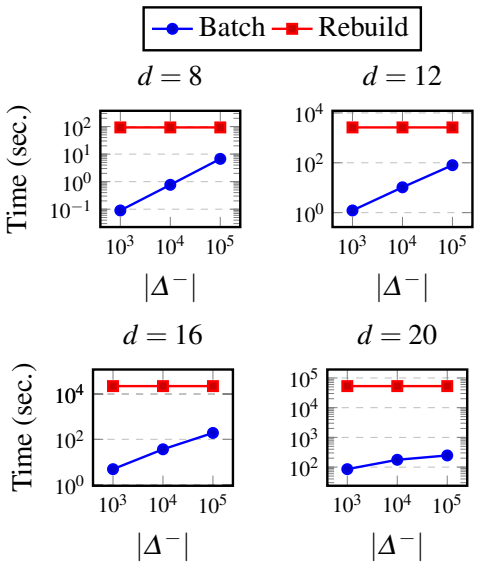
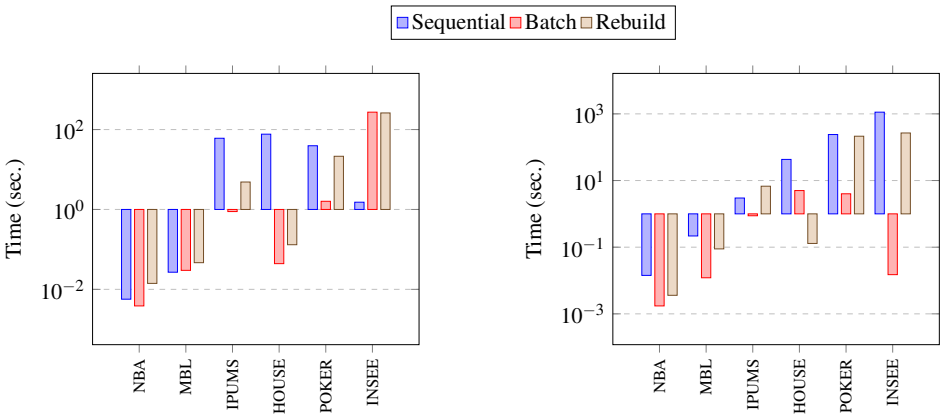


Figure 3.18: Deleting large Δ^- by varying d ($n = 10^6$)

Real data

To evaluate insertions, we compute NSC with 95% of the dataset randomly chosen, and then we insert the remaining 5%. Note that here, the inserted tuples are not guaranteed to belong to the new topmost skyline. We make this choice in purpose so that the experiment becomes closer to realistic situations: users do not insert just not dominated tuples. Likewise for deletions, we compute NSC for the whole data set, and we delete 5% tuples chosen randomly. The obtained execution times are depicted in Figure 3.19.



(a) Inserting 5% (b) Deleting 5%

Figure 3.19: Inserting and deleting 5% of real data

In general, we observe the same behavior as with synthetic data, that is Batch method is the fastest for insertions as well as for deletions. We note however some exceptions with the insertion experiments, e.g. for NBA and MBL, Sequential method is as good as Batch method, while for INSEE, Sequential method is clearly faster. We explain this

behavior by the fact that the *topmost* skylines of these three data sets are quite small (see Table 3.6), therefore due to the random selection of the 5% that we insert, Δ^+ is likely to be composed of dominated tuples, consequently the *topmost* will not change. Sequential method checks the tuples of Δ^+ one by one whether they are dominated, thus it performs at most $|\Delta^+| * |topmost|$ comparisons. While Batch method computes *topmost* of $T \cup \Delta^+$, i.e. computes skyline of $T \cup \Delta^+$ over $\mathcal{D}$. Note that this behavior is similar to that observed with correlated synthetic datasets.

Experiments conclusion We learn from NSC maintenance experiments is that the harder the computation of NSC from scratch, the more efficient incremental methods. When NSC computation is already fast, the lack of the incremental methods gain is not crippling. Moreover, experiments suggest that one should prefer Sequential insertions when the *topmost* skyline is small. For deletions, Batch method is always the best choice.

3.5 Conclusion

In this chapter, we studied the incremental maintenance of the structure NSC in presence of dynamic data. In a previous work, this structure has been shown efficient for answering subspace skyline queries, however no incremental maintenance procedure has been provided. Through slight modifications in the structure design and efficient algorithms, we have shown that NSC can efficiently handle updates. Moreover we have shown that these modifications do not alter its efficiency with respect to both construction and query answering times, and space consumption.

We considered in this chapter data changing in unpredictable way, i.e. a set of unknown size can be deleted/inserted at any time. However in some real world situations, data are appended only and queries consider a window. In such cases, this chapter's proposals are not suitable. For example, consider a dataset where N tuples are inserted every k units of time. Then NSC's update should occur in the interval between every two batches. Otherwise NSC would never provide accurate results for skyline queries. In the next chapter, we give examples of such situation and address the maintenance of NSC in presence of streaming data.

Chapter 4

Maintenance of NSC with streaming data

4.1 Introduction

Computing the skyline in a streaming context has been investigated in e.g., [79, 11, 40, 41]. They consider a data set extended every θ units of time by a new tuple. All tuples may have a specified common lifetime ω, i.e., they are valid during a period of size ω starting from their arrival time, then they become obsolete and can be removed. Since the underlying data set is changing every θ units of time, i.e. a new tuple is appended and an old one is discarded, the answer to a skyline query may change at the same frequency. Because the complexity of skyline queries evaluation is, in the worst case, quadratic in the data size, there is a need of incremental procedures to maintain the skyline up to date. Previous works that tackled the issue have mainly considered the problem of maintaining a *single* skyline. In the present chapter, we investigate the problem of answering multidimensional skyline queries over streaming data. More precisely, we address the incremental maintenance of NSC in a streaming context.

As discussed in Chapter 1 Section 1.1.2, none of the previous solutions to monitor a single skyline can be naturally and efficiently adapted to the context of multidimensional skylines. As a motivating scenario, consider a data analytics agency which receives a live stream of statistical data about tweets. Each tuple represents a tweet statistics of the form (`UserId`, `TweetId`, `#retweets`, `#likes`, `#comments`, `retweet_depth`, `#followers`, `#shares_on_other_social_nets`). The agency is interested by the skyline tweets wrt several subsets of attributes in a 24 hours sliding window. This information can be useful for, e.g., identifying the k-most influential tweets by counting the number of subspace skylines they belong to. Considering the last

6 attributes representing statistics, there are 63 distinct skyline queries ($2^6 - 1$) that can be submitted to this multidimensional data stream. Because of data velocity, monitoring the Top-K elements requires to refresh the results as frequently as possible: if each second a new tuple is received and an old tuple is outdated, then each of the 63 queries must be re-evaluated to keep the Top-K tweets wrt to the last 24 hours up to date. Observe that if $\theta = 1sec.$ and the evaluation of these queries takes more than 1 second, then the Top-K query answer will never reflect the actual data. One solution to cope with this problem is to reduce the size of input data. This can be done by reducing the tuples validity time, e.g., considering just the last hour instead of the last 24 hours divides the input size by 24. Notice that this may not reflect the business needs of the company. Another solution would be to reduce the number of skyline queries, e.g., select "most representative" 10 queries among the 63 possible ones. Again, this could bias the result.

In this chapter, we present the framework MSSD (Multidimensional Skylines over Streaming Data) that handles (i) a buffer B where tuples are first collected during k units of time, (ii) a main dataset T that stores tuples arrived in a *window* of size ω and (iii) a variant of NSC called NSCt i.e., NSC with timestamps.

We adopt a *micro-batch* processing approach: the stream source emits one tuple every θ units of time[1]. Our framework collects the tuples into a buffer during k units of time. Thereafter, the buffered tuples are inserted into T and the outdated ones are removed from T. Simultaneously, the maintenance of the index structure NSCt is triggered. When a subspace skyline query is issued, NSCt is used in order to compute the skyline. Continuing with the analytics agency example, suppose that it is interested in querying a 24 hours window, i.e., ω, and sets the batch interval, i.e., k, to 15 min with a processing at {HH:00, HH:15, HH:30, HH:45}. Then, for example at 13:40, T covers the window $(13:30(-1\,day), 13:30]$. Note that tuples arrived during the interval $(13:30, 13:40]$ do not belong to T and are not considered for queries. In addition, those arrived during $(13:30(-1\,day), 13:40(-1\,day)]$ still belong to T despite the fact that they are no more valid. So the exact semantics of the queries our framework answers is: the skyline over the data that *were* valid at the last maintenance time.

We balance the maintenance frequency with the query answering performance. A user interested in querying a more close window will choose to reduce k. However someone who is interested in processing a big number of queries will delay the maintenance process.

[1]This limitation of the number of tuples per θ units of time is set just for the ease of the presentation. Without any change, our framework can handle the case of multiple tuples per time unit.

Organization The next section gives the additional definitions and notations used throughout the chapter. We then describe our proposed framework. We present NSCt, how it (i) is maintained and (ii) is used to answer skyline queries. Afterwards, we present the experiments we performed.

4.2 Preliminaries

We begin by presenting the definitions and notations used throughout the chapter.

Notations and definitions

In addition to the general definitions and notations, in this chapter we consider data appending to a data repository S in a streaming mode. We consider that all tuples share the same validity period of size ω which starts once the tuple is integrated into the data repository S. Every tuple t has a timestamp corresponding to the starting time of its validity period denoted $TS(t)$. To simplify, we consider time as isomorphic to the set of natural numbers which means $TS(t) \in \mathbb{N}$. At timestamp $TS(t)+\omega$, the tuple t is considered outdated, therefore deleted from the data repository. We also consider the natural order between timestamps, i.e., $TS(t_1) < TS(t_2) \Leftrightarrow t_1$ has been integrated before t_2. By convention, the current timestamp, denoted ts_c corresponds to the timestamp of the most recent tuple in the data repository. That is, $ts_c = \underset{t \in S}{\operatorname{argmax}}\, TS(t)$.

In this chapter, the skyline is defined over a window as follows,

Definition 9 (Subspace skyline over a window). *Let X be a subspace, $[a,b]$ be a time interval and S be a data repository. Let $S[a,b] = \{t \in S | TS(t) \in [a,b]\}$. Then, the subspace skyline of S wrt X over $[a,b]$, denoted $Sky_{[a,b]}(X,S)$, is the set $\{t \in S[a,b] | \nexists t' \in S[a,b]$ s.t $t' \prec_X t\}$.*

To simplify the notation, we sometimes write just $Sky(X)$ when the underlying S is understood and we omit $[a,b]$ because we focus on the time interval $(ts_c - \omega, ts_c]$, i.e., the valid tuples wrt the current timestamp.

Example 18. *Let S be the following set of tuples:*

Id	$Timestamp$	A	B	C
t_1	11	1	2	1
t_2	12	1	1	2
t_3	13	2	2	2
t_4	14	2	3	1

Assume that $\omega = 2$, i.e., a tuple is still valid 2 units of time after its arrival. Because the most recent tuple in S has a timestamp equal to 14, all tuples which arrived at timestamp $14 - \omega = 12$ or before, are considered to be outdated and hence removed. In S, this is the case for t_1 and t_2. Hence, e.g., $Sky(AB) = \{t_3\}$ and $Sky(BC) = \{t_3, t_4\}$. Now, let $\omega = 8$, i.e., all tuples are valid. Then $Sky(AB) = \{t_2\}$ and $Sky(BC) = \{t_1, t_2\}$.

Table 4.1 summarizes the additional notations for this chapter.

Term	Meaning
$ts_c, i, j, \ldots$	timestamps
ω	size of sliding window = validity duration of tuples
k	batch interval = frequency of batch updates
θ	streaming delay = time separating two successive tuples
B	buffer = set of tuples waiting to be inserted
$TS(t)$	timestamp of tuple t
Transaction δ	set of tuples

Table 4.1: Notations

4.3 MSSD framework

In this section, we present the architecture of our framework, the index structure we propose to maintain the subspaces where a tuple is dominated and the process of answering issued subspace skyline queries.

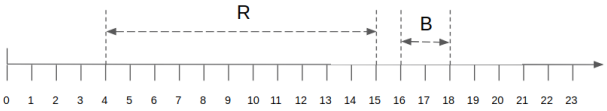
4.3.1 MSSD architecture

MSSD consists of three data structures, (i) a buffer B, (ii) a main dataset T and (iii) an index structure NSCt. MSSD integrates a *micro-batch* processing: (i) during a time interval of size k, tuples are first inserted into the buffer B, afterwards (ii) the content δ^+ of B is inserted into the dataset T, (iii) the outdated tuples are deleted from T, and finally (iv) the update of NSCt is triggered. The framework is clocked by the parameter θ which determines the delay between two timestamps t_i and t_{i+1}, i.e., the delay between two successive tuples. For the ease of the presentation, we consider that every θ units of time, one and only one tuple is buffered by our framework. Moreover, we assume that k is a divisor of ω, and both are multiples of θ. Hence, at each time, the number of tuples belonging to B is at most equal to $\frac{k}{\theta}$. On another hand, and after warm up, i.e., current timestamp t_c greater than ω, T continuously contains exactly $\frac{\omega}{k}$ transactions which corresponds to a total of $\frac{\omega}{\theta}$ tuples[2].

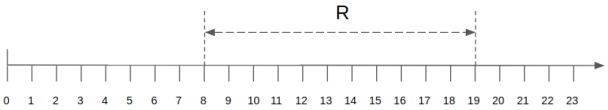
[2]If at most ℓ tuples can arrive at the same time instead of just 1, then $|B| \leq \ell \cdot \frac{k}{\theta}$ and $|R| \leq \frac{\omega}{\theta} \cdot \ell$.

Note here that throughout the paper, the timestamp of a tuple t, i.e. $TS(t)$, corresponds to the timestamp when the tuple t has been inserted into T. Hence, all tuples of a single transaction share the same timestamp.

Example 19. *In Figure 4.1, we depict two timestamps of the framework timeline. We consider the following configuration: window size $\omega = 12 \cdot \theta$ and maintenance frequency $k = 4 \cdot \theta$. Therefore B and T contains respectively 4 and 12 tuples.*



(a) Framework state at 18



(b) Framework state at 19

Figure 4.1: Framework timeline with $\theta = 1, \omega = 12$ and $k = 4$

In Figure 4.1(a), the current timestamp is $t_c = 18$, T contains tuples arrived during the window $[4,15]$. From timestamp 16, tuples are appended to the buffer. The queries issued during $[16,19)$ target tuples in T, i.e, the window $[4,15]$. In Figure 4.1(b), $t_c = 19$ and tuple t_{19} is appended to the buffer, thereafter transaction $\delta^+ = \{r_{16}, r_{17}, r_{18}, r_{19}\}$ is inserted into T and $\delta^- = \{r_4, r_5, r_6, r_7\}$ is discarded from T. The window covered by T is henceforth $[8,19]$. Note that T can be seen as a sequence $R[i]_{1 \leq i \leq \omega/k}$ of transactions where $R[i]$ corresponds to a set of tuples inserted at the same time. Here T is a sequence of $\omega/k = 3$ transactions. At $t_c = 19$, $R[3] = \delta^+$.

Remark 3. *As illustrated in the previous example, considering S as the set of all tuples seen until current time t_c and for given ω and k, T contains the tuples arrived during the interval $(t_e - \omega, t_e]$ where $t_e = t_c - ((t_c + 1) \, modulo \, k)$. Thus, the exact queries evaluated at t_c are $Sky_{(t_e-\omega,t_e]}(X,S)$ (c.f Definition 9).*

4.3.2 NSCt index structure

In this section, we present our framework index structure NSCt, (Negative SkyCube with timestamps) which is inspired from NSC presented earlier.

We recall NSC and explain why it is not suitable for streaming data. NSC stores for each tuple, a set of pairs which summarize the set of subspaces where the tuple is dominated. The pairs are computed wrt each tuple in the dataset. However, not all

pairs are stored. We compress the initial set of pairs P into an *equivalent* set of pairs P'. While this compression step improves the space consumption and query answering time, it makes deletion harder.

Let us consider the set of tuples depicted below:

Id	A	B	C
t_1	2	2	2
t_2	0	0	3
t_3	5	1	3
t_4	1	1	3
t_5	1	0	4

Comparing t_5 to the other tuples returns the following set of pairs

$Compare(r_5, r_1)$	$Compare(r_5, r_2)$	$Compare(r_5, r_3)$	$Compare(r_5, r_4)$
$\langle C \vert \emptyset \rangle$	$\langle AC \vert B \rangle$	$\langle C \vert \emptyset \rangle$	$\langle C \vert A \rangle$

Table 4.2: Pairs of t_5

For example, observe in Table 4.2 that subspaces covered by both $\langle C|A\rangle$ and $\langle C|\emptyset\rangle$ are likewise covered by $\langle AC|B\rangle$, hence NSC keeps only one single pair, i.e., $\langle AC|B\rangle$ computed wrt t_2. Now assume that t_2 is deleted from the dataset, hence t_5 has to be compared to all the remaining tuples in the dataset to recover its associated pairs. In a streaming context where the flow of insertions/deletions is high, this approach of pairs sets maintenance is not suitable because it is too time consuming.

We adapt this structure in order to handle efficiently streaming data without giving up much performance of NSC. More precisely, given a dataset T, for a tuple t in T, we organize its set of pairs $Pairs(t)$ as a sequence of buckets where each bucket $Pairs(t).Buck_i$ contains the pairs computed wrt a transaction $R[i]$ in T.

The following example illustrates the update procedure.

Example 20. *Let $\theta = 1$, the window size $\omega = 6 \cdot \theta$ and a batch interval $k = 2 \cdot \theta$. If the current time t_c is 6 then seven tuples $t_0, r_1, \ldots, r_6$ are supposed to have arrived so far. Accordingly, at this timestamp, T is composed of 3 transactions $\{r_0, r_1\}$, $\{r_2, r_3\}$ and $\{r_4, r_5\}$, and t_6 is just buffered into B. Table 4.3 represents the current status of T. It shows the projection of the tuples on the dimensions A, B and C and their arrival time, which corresponds to the timestamp where the framework received the tuple. At the current timestamp, we consider that the pairs of $\{r_0, \ldots, r_5\}$ have already been computed. At timestamp 7, a new tuple t_7 is buffered. The content of B is then inserted into T. In addition, the first two tuples t_0 and t_1 are no more valid so they are removed from T as it*

Transaction	*Id*	*A*	*B*	*C*	*Arrival time*
$R[1]$	r_0	5	4	1	0
	r_1	3	4	2	1
$R[2]$	r_2	5	1	3	2
	r_3	1	1	3	3
$R[3]$	r_4	1	0	4	4
	r_5	0	1	5	5

Table 4.3: Dataset T at timestamp 6

Transaction	*Id*	*A*	*B*	*C*	*Arrival time*
$R[1]$	r_2	5	1	3	2
	r_3	1	1	3	3
$R[2]$	r_4	1	0	4	4
	r_5	0	1	5	5
$R[3]$	r_6	2	0	6	6
	r_7	2	1	1	7

Table 4.4: Dataset T at timestamp 7

is depicted in Table 4.4. At the same time, NSCt *maintenance is triggered in order to (i) compute the pairs of* t_6 *and* t_7 *wrt* $\{r_2,\ldots,r_7\}$ *(ii) update the pairs of* $\{r_2,\ldots,r_5\}$ *wrt* t_6 *and* t_7.

In the following, we detail our approach (i) to compute and organize the pairs of a newly inserted tuple into T and (ii) to update the set of pairs of an existing tuple.

Handling a new tuple

Let δ^+ be a transaction. Let T be the set of tuples from where the outdated tuples are removed and those in δ^+ are inserted. Let t be a newly inserted tuple into T, i.e., $r \in \delta^+$. We compute its pairs wrt the tuples in T and organize them as follows: we allocate to t a **sequence of buckets** that we call $Pairs(t)$ where each bucket $Pairs(t).Buck_i$ contains the pairs computed wrt a transaction $R[i]$ in T. Since there exists $\frac{\omega}{k}$ transactions in T, then each tuple has $\frac{\omega}{k}$ buckets. The timestamp of a bucket, denoted by $TS(Buck_i)$, is the timestamp of the tuples to which it is related.

We describe the process of computing the pairs associated to a newly inserted tuple in Algorithm 12 and illustrate it in example 21. Algorithm 12 is called for every tuple in the transaction δ^+.

Example 21. *We report in Table 4.5 the pairs of tuples* t_6 *and* t_7 *from the previous example. Recall that from the values of* $\omega = 6$ *and* $k = 2$*, the number of transactions in* T *is* $\frac{6}{2} = 3$ *which is the number of buckets we associate to each tuple. The first bucket* $Buck_1$ *is obtained by comparing* t_6 *and* t_7 *to the tuples belonging to the oldest transaction in* T*, i.e.,* $\{r_2, r_3\}$ *and the second by comparing them to* $\{r_4, r_5\}$*. The last bucket corresponds to the pairs obtained by comparing the new tuples to each others, i.e.,* t_6 *to* t_7 *and vice versa.*

Algorithm 12: computePairs

Input: tuple t, T
Output: $Pairs(t)$

```
1 Pairs(t) ← ∅
2 Buck_i ← ∅ ∀i ∈ [1, ω/k]
3 begin
4     for i ∈ [1, ω/k] do
5         foreach r' ∈ transaction R[i] do
6             // we iterate over the tuples belonging to the i^th transaction of T
7             Buck_i ← Buck_i ∪ {compare(r, r')}
8         Pairs(t) ← Pairs(t) ⊎ Buck_i
9 return Pairs(t)
```

Id	$Buck_1$	$Buck_2$	$Buck_3$
t_6	$\langle A\|C\rangle, \langle\emptyset\|C\rangle$	$\langle A\|\emptyset\rangle, \langle A\|B\rangle$	$\langle C\|A\rangle$
t_7	$\langle A\|B\rangle, \langle\emptyset\|B\rangle$	$\langle A\|B\rangle, \langle AB\|\emptyset\rangle$	$\langle B\|A\rangle$

Table 4.5: Pairs of t_6 and t_7

Complexity analysis

Given the parameters ω, k and θ, the size of the dataset T is $\frac{\omega}{\theta}$. Moreover, the size of a transaction δ^+ to be inserted is $\frac{k}{\theta}$. Let $n = |R|$ and $\ell = |\delta^+|$. Each tuple in δ^+ is compared to tuples in T (except itself) hence the process of computing the pairs of a transaction has a time complexity $O(\ell \cdot n)$. Likewise space complexity is $O(\ell \cdot n)$ as from each comparison, one pair is stored. Observe however that each bucket is a *set* of pairs. Hence, $\ell \cdot n$ is the maximal number of pairs.

Minimization of Pairs(t).

We show in this section the minimization process of NSCt which is shaped for streaming data.
Let us first recall the notion of set of pairs *equivalence*.

Definition 10 (Equivalence)**.** *Let P and Q be two sets of pairs. Then P and Q are equivalent, $P \equiv Q$, iff $cover(P) = cover(Q)$*

Example 22. *Let $P = \{\langle A|BC\rangle, \langle B|C\rangle, \langle AB|C\rangle\}$, P covers the subspaces $\{ABC, AB, AC, BC, A, B\}$. Then both $P_1 = \{\langle A|BC\rangle, \langle B|C\rangle\}$ and $P_2 = \{\langle AB|C\rangle\}$ are equivalent to P.*

Now, given a sequence of buckets $Pairs(t) = [Buck_1, \ldots, Buck_m]$, for each $Buck_i$ we compute a subset $s_i \subseteq Buck_i$ such that $\forall p \in s_i$ the set $s_i \setminus p \not\equiv s_i$, i.e., s_i is a minimal

equivalent subset of $Buck_i$. We illustrate in the following example the buckets of t_6 and t_7 where each bucket is replaced by a minimum equivalent subset.

Example 23. *Table 4.6 shows the new set of pairs of t_6 and t_7 (cf. Table 4.5) after summarizing their respective buckets. For example, the pair $\langle A|\emptyset\rangle$ is removed from the $Bucket_2$ of t_6 because it is covered by $\langle A|B\rangle$ belonging to the same bucket.*

Id	1	2	3
t_6	$\langle A\|C\rangle$	$\langle A\|B\rangle$	$\langle C\|A\rangle$
t_7	$\langle A\|B\rangle$	$\langle AB\|\emptyset\rangle$	$\langle B\|A\rangle$

Table 4.6: Pairs of t_6 and t_7 minimized by equivalence

The intra-bucket size minimization as described above can be extended to inter-buckets minimization to further reduce the memory storage. Intuitively, a pair belonging to $Buck_i$ is *redundant* if the subspaces it covers are covered by pairs in more recent buckets. Let us illustrate this observation.

Example 24. *Consider $Pairs(r_6)$ and $Pairs(r_7)$ depicted in Table 4.6. Observe that for tuple t_7, the subspaces that the pair $\langle A|B\rangle$ in $Buck_1$ covers ($\{AB, A\}$) are covered by $\langle AB|\emptyset\rangle$ in $Buck_2$ ($\{AB, A, B\}$). Therefore, we discard $\langle A|B\rangle$ from $Buck_1$. We report in Table 4.7 the minimized set of pairs of t_6 and t_7.*

Id	1	2	3
t_6	$\langle A\|C\rangle$	$\langle A\|B\rangle$	$\langle C\|A\rangle$
t_7		$\langle AB\|\emptyset\rangle$	$\langle B\|A\rangle$

Table 4.7: Pairs of t_6 and t_7 minimized

Remark 4. *From the example above, one may wonder why pair $\langle B|A\rangle$ is not removed from $Buck_3$ of t_7 since it is covered by $\langle AB|\emptyset\rangle$ in $Buck_2$. We make the choice to keep it for update optimization considerations. Indeed, while the deletion of that pair reduces memory consumption and preserves skyline semantics, it makes the update procedure harder: as pairs in $Buck_2$ become outdated before those in $Buck_3$, more precisely the tuples which served to obtain them, then as soon as $Buck_2$ becomes outdated we need to recover $\langle B|A\rangle$ because the reason of its removal becomes no more valid. So our choice to not minimizing the buckets wrt to older ones can be seen as trade off between memory minimization and update efficiency.*

We combine the two minimization processes explained above (intra and inter buckets) and formalize the problem of the global minimization of $Pairs(t)$ for a given tuple t as follows:

Problem 1 Let $Pairs(t) = [Buck_1, \ldots, Buck_m]$. Then $\forall i \in [1,m]$, find $s_i \subseteq Buck_i$ s.t $s_i \cup \bigcup_{j=i+1}^{m} Buck_j \equiv \bigcup_{j=i}^{m} Buck_j$ and s_i is of minimum size.

The problem above addresses the minimization of $Pairs(t)$ by both *intra* and *inter* buckets minimization. Indeed, for every bucket $Buck_i$ in $Pairs(t)$, we look for a subset $s_i \subseteq Buck_i$ such that the set of pairs $s_i \cup \bigcup_{j=i+1}^{m} Buck_j$ and $\bigcup_{j=i}^{m} Buck_j$ are equivalent. The resulting set s_i contains then pairs not *covered* by pairs in the union of buckets following $Buck_i$, i.e., $\bigcup_{j=i}^{m} Buck_j$

Theorem 5. *Problem 1 is NP Hard.*

Proof. For the special case where $Pairs(r)$ contains only one bucket $Buck_1$, hence we look for $s_1 \subseteq Buck_1$ such that $cover(s_1) \equiv cover(Buck_1)$ and s_1 with minimum size. The time complexity is $O(2^{|Buck_1|})$.

Let $P = Buck_1$, by considering all the subsets of P, one can check which are equivalent to P and which are of minimum size. Thus, the problem is in NP. The hardness proof is based on a reduction from the Minimal Set Cover (**MSC**) problem. Given an **MSC** instance, we build a table T with a distinguished tuple t where the number of dimensions d is equal to the number of elements to be covered in **MSC** and where the number $n+1$ of records is equal to the initial number of sets in **MSC** in addition to the distinguished t. So, there is a bijection between the n records and the n sets of **MSC** instance. The n records form the topmost of T and distinguished tuple t is compared to each of them giving rise to a set of pairs P. We show that the minimum equivalent subset of P coincides with a solution of **MSC**. Let $s = \{s_1, s_2, \ldots, s_n\}$ be the input set of sets in the **MSC** instance. W.l.o.g, we assume that there is no inclusion between these sets and none of them does contain all the elements to cover. For every set $s_j \in s$, we add to T a tuple t_j such that $t_j[i] = 0$ iff $i \in s_j$ otherwise $t_j[i] = 1$. In addition, we add to T a tuple $t = (1,1,\ldots,1)$ be a d-tuple. For example, let $s = \{s_1 = \{1,2\}; s_2 = \{2,3\}; s_3 = \{1,3\}\}$ be the **MSC** instance. The number of elements to cover is $d = 3$ and the number of sets $n = 3$. So, we get a table T with $n+1 = 4$ records, including t, and 3 dimensions. This table is depicted below.

Id	**1**	**2**	**3**
t	1	1	1
t_1	0	0	1
t_2	1	0	0
t_3	0	1	0

Clearly, every t_j dominates t and $t_i \not\prec t_j$. Hence, $\{t_1, \ldots, t_n\}$ is the topmost. By comparing t to the topmost, we obtain $P(s) = \{p_1, \ldots, p_n\}$. There is a one to one

correspondence between $s_i \in s$ and $p_i = Compare(t, t_i)$. For example, $Compare(t, t_1) = \langle 12|3\rangle$ corresponds to $s_1 = \{1,2\}$. Let $u \subseteq s$ and let $P(u)$ be the set of pairs p_j such that $p_j = Compare(t, t_j)$ where t_j corresponds to some $s_j \in u$. Let $Cover(P(u))$ denote the set of subspaces covered by the pairs in $P(u)$. We show that $\cup_{s_j \in u} s_j = \cup_{s_i \in s} s_i$ iff $P(u) \equiv P(s)$ and this proves the claim.
(i) $P(u) \equiv P(s) \Rightarrow u \equiv s$: Every $p_j \in P(u)$ is of the form $\langle X_j|Y_j\rangle$ thus it covers, among others, the subspace X_j which actually corresponds to the content of $s_j \in u$. As $P(u) \equiv P(s), \forall p_i = \langle X_i|Y_i\rangle \in P(s)$, $P(u)$ covers X_i and the union of the X_i's is the union of the s_i's. Hence $u \equiv s$.
(ii) $u \equiv s \Rightarrow P(u) \equiv P(s)$: Assume, for the sake of contradiction, that $P(u) \not\equiv P(s)$. There must exist a subspace Z s.t $P(s)$ covers Z but not $P(u)$. Thus, there exists $p_i \in P(s)$ such that $p_i = \langle X_i|Y_i\rangle$ s. t $Z \subseteq X_iY_i$ and $Z \cap X_i \neq \emptyset$. Note that every p_i is of the form $\langle s_i|\mathcal{U} \setminus s_i\rangle$ where $\mathcal{U} = \cup_{s_j \in s} s_j$. Therefore, to cover Z, a pair $\langle s_j|\mathcal{U} \setminus s_j\rangle$ needs just to satisfy $Z \cap s_j \neq \emptyset$. Such an s_j is necessarily in u because otherwise $u \not\equiv s$, i.e., there exists $k \in \mathcal{U}$ such that there is no $s_i \in u$ s.t $k \in s_i$ and thic contradicts the fact that $u \equiv s$. We conclude that every (minimum) solution of the set cover problem corresponds to a (minimum) solution to our problem regarding the distinguished tuple t of the table T above which terminates the proof. □

A Polynomial time greedy algorithm for pairs minimization

We present in this section a polynomial time greedy algorithm for solving Problem 1. We establish the theoretical guarantees of its solution wrt an optimal solution as well as its time complexity. For the ease of the presentation, in the following we denote by $P1$ the problem we address. We transform an instance of our problem $P1$ into an instance of a problem $P1^*$ and show that a given sequence $[s_1, \ldots, s_m]$ that is a solution for $P1$ coincides with S^* a solution of $P1^*$.

We begin by defining a function T which takes as input a pair p and the index of the bucket p belongs to, and returns the set of subspaces it covers duplicated i times. More specifically:

Definition 11. *Let t be a tuple and $Pairs(t) = [Buck_1, \ldots, Buck_m]$ be its associated sequence of buckets. Let $p \in Buck_i$, then $T(p,i) = \{Z_1, \ldots, Z_i | Z \in cover(p)\}$*

Example 25. *Consider $Pairs(r_7)$ depicted in Table 4.7 and $\langle AB|\emptyset\rangle \in Buck_2$. Then the transformation of the pair $\langle AB|\emptyset\rangle$ is $T(\langle AB|\emptyset\rangle, 2) = \{A1, B1, AB1, A2, B2, AB2\}$.*

Now we define a function $\mathcal{T}$ that transforms an instance of $P1$.

Definition 12. *Let $D = \{D_1, \ldots, D_d\}$. Let t be a tuple and $Pairs(t) = [Buck_1, \ldots, Buck_m]$ be its sequence of buckets. Then $\mathcal{T}(Pairs(t))$ is the following set of subspaces $\{T(p) | \forall Buck_i, \forall p \in Buck_i\}$. The domain of $\mathcal{T}$ is a sequence of set of pairs and $Im(\mathcal{T}) = \{T(\langle X|Y\rangle, i) | i \in \mathbb{N}, X \subseteq D, Y \subseteq D$ and $X \cap Y = \emptyset\}$.*

In the following we formalize the problem $P1^*$

Problem 1^* Let $Pairs(t) = [Buck_1, \ldots, Buck_m]$. Find $S^* \subseteq \mathcal{T}(Pairs(t))$ such that S^* covers the same set as $\mathcal{T}(Pairs(t))$ and S^* is of minimum size.
Observe that $P1^*$ is equivalent to the **MSC** problem. Now we show that a solution for $P1^*$ is also a solution for $P1$.

Theorem 6. *Let $[s_1, \ldots, s_m] \subseteq Pairs(t)$, then $[s_1, \ldots, s_m]$ is a solution for $P1$ iff $\mathcal{T}([s_1, \ldots, s_m])$ is solution of $P1^*$.*

In order to prove the above theorem, we first have to prove that the function $\mathcal{T}$ is bijective.

Lemma 5. *$\mathcal{T}$ is bijective.*

Proof of Lemma 5. Observe in the definition that $Im(\mathcal{T})$ is composed of the image by the function T of all possible pairs according to a set of dimensions D, hence $\mathcal{T}$ is surjective. Now we show that $\mathcal{T}$ is injective. Let seq_1 and seq_2 two sequences of sets of pairs. We prove by contradiction that if $\mathcal{T}(seq_1) = \mathcal{T}(seq_2)$ then $seq_1 = seq_2$. Suppose $\mathcal{T}(seq_1) = T(seq_2)$ and $seq_1 \neq seq_2$. We first prove that the two sequences have the same size, i.e. the same number of sets of pairs. Suppose the size of the sequences is different between seq_1 and seq_2, e.g. $n = |seq_1| > |seq_2| = m$, let X be a subspace covered by a pair in the n^{th} set of seq_1. Hence X_n does belong to a element in $\mathcal{T}(seq_1)$ which is impossible because $\mathcal{T}(seq_1) = \mathcal{T}(seq_2)$. Therefore seq_1 and seq_2 contain the same number of sets of pairs. Let $seq_1 = [s_1, \ldots, s_m]$ and $seq_2 = [s'_1, \ldots, s'_m]$, we now prove that $s_i = s'_i \forall i \in [1, m]$. Suppose that $s_i \neq s'_i$, more particularly suppose $p \in s_i$ and $p' \in s'_i$ such that $p \neq p'$, we show that it's impossible that $cover(p)$ equals $cover(p')$. Let $p = \langle X_1|Y_1\rangle$ and $p' = \langle X_2|Y_2\rangle$.

- if $X_1 \subset X_2$ then $\forall Y_1, Y_2\ \exists Z \in X_2 \setminus X_1 \notin cover(p)$.
- if $X_2 \subset X_1$ then $\forall Y_1, Y_2\ \exists Z \in X_1 \setminus X_2 \notin cover(p')$.
- if $X_1 = X_2$ then
  - if $Y_1 \subset Y_2$ then $X2Y2 \notin cover(p)$
  - if $Y_2 \subset Y_1$ then $X1Y1 \notin cover(p')$

Hence in all cases, $cover(p) \neq cover(p')$ therefore $\mathcal{T}(seq_1)$ is not equal to $\mathcal{T}(seq_2)$ which contradicts our first assumption. We conclude that $\mathcal{T}$ is injective and therefore bijective. □

Next we show that a solution $[s_1,\ldots,s_m]$ for $P1$ coincides with a solution S^* for $P1^*$, that is proving the previous theorem.

Proof of Theorem 6. $\Rightarrow$ Let $[s_1,\ldots,s_m]$ be a solution for $P1$. We prove by contradiction that $\mathcal{T}([s_1,\ldots,s_m])$ is a solution of $P1*$. Suppose that $\mathcal{T}([s_1,\ldots,s_m])$ is not a solution of $P1*$. So there must exist $X_i \in \bigcup_{Z\in\mathcal{T}(Pairs(r))} Z$ such that $X_i \notin \mathcal{T}([s_1,\ldots,s_m])$ and i the highest index. As $\mathcal{T}$ is bijective, there exists a pair $p \in Buck_i$ such that $X_i \in T(p,i)$. As no other $S_{i+1},\ldots,S_m$ covers X then S_i should cover X, which is not the case as p not in S_i. Then $[s_1,\ldots,s_m]$ is not a solution, which contradicts our assumption.

$\Leftarrow$ Suppose $[s_1,\ldots,s_m]$ is not a solution of $P1$ such that s_i is the one that does not satisfies $Buck_i$. Let X be a subspace covered by pairs in $Buck_i$, however not covered by pairs in S_i, then as $\mathcal{T}$ bijective, X_i will not belong to $\mathcal{T}([s_1,\ldots,s_m])$. But $X_i \in \mathcal{T}(Pairs(r))$ because $X \in Buck_i$. Hence $\mathcal{T}([s_1,\ldots,s_m])$ is not a solution for $P1*$.

This is true for a minimum solution as well.

□

We present in Algorithm 13 the steps to find a minimal solution for an instance of $P1$. It first transforms the instance of $P1$ to an instance of $P1^*$ by computing the union of a pairs sequence (Lines 3 to 5), then according to Theorem 2 it solves $P1^*$ by using a greedy algorithm solving **MSC** [78] (Line 6). Finally, for every element in the solution of the second problem S, we keep the corresponding pair in $Pairs(t)$ (lines 7-10).

Algorithm 13: minimizingNSCt

Input: $Pairs(t) = [Buck_1,\ldots,Buck_m]$
Output: $Pairs(t)$

```
1  begin
2      I ← ∅
3      for i ∈ [1,m] do
4          for p ∈ Buck_i do
5              I ← I ∪ T(p,i)
6      S ← MSC(I)
7      Pairs(t) ← ∅
8      for s ∈ S do
9          Pairs(t) ← Pairs(t) ∪ T^{-1}(s)
10     return Pairs(t)
```

Time complexity and size guarantee Given ω, k, θ and d. Let r be a tuple, $Pairs(t) = \{Buck_1, \ldots, Buck_{\frac{\omega}{k}}\}$ be its set of buckets. Each bucket contains at most $\frac{k}{\theta}$ pairs (duplicate pairs are stored once). To simplify, let $m = \frac{\omega}{k}$, $n = |R| = \frac{\omega}{\theta}$ and $l = |\delta^+| = \frac{k}{\theta}$. Regarding the time complexity, computing the set I takes $O(m \cdot l) = O(n)$ time. I contains n sets at most, hence computing a solution by *MSC* greedy algorithm presented in [78] takes $O(n^2)$ time. The final step (line 8-9) is linear in the size of the solution S. This size is guaranteed by [78] to be $|S| \leq |S_{opt}| \cdot log(e)$ such that e is the size of the largest element in I and which is bounded by $2^d \cdot m$. Hence $|S| \leq |S_{opt}| \cdot (d + log(m))$

The batch interval k impacts the minimization process in a way that the resulting minimized set is smaller when k is larger. The following proposition describes this behavior and we process a set of experiments in section 4.4 in order to measure the impact of k on NSCt size.

Proposition 5. *Let k and k' be two batch intervals such that $k = c \cdot k'$ with $c \geq 2$. Given ω, let t be a tuple, P_k and $P_{k'}$ be its sets of pairs with respectively k and k'. Then $|P_k| \leq |P_{k'}|$.*

Proof. Let k and k' be two batch intervals such that $k = c \cdot k'$ with $c \geq 2$, we prove that $|P_k| > |P_{k'}|$ is impossible.
Suppose $|P_k| > |P_{k'}|$, then there exists a record r' such that $compare(r, r') \in Buck_i$ and $Buck_i \in P_k$, this means that $p = compare(r, r')$ is not covered by other pairs in $Buck_i$, more precisely $cover(p) \notin cover(Buck_i \setminus p)$. Let $Buck_{i1} \cdots Buck_{ic} \in P_{k'}$ representing the same interval as $Buck_i$. The supposition implies that p is not in any bucket $Buck_{i1} \cdots Buck_{ic}$, which implies that either $cover(p) \in cover(Buck_{i1})$ or $cover(p) \in cover(Buck_{i2}) \cdots$ or $cover(p) \in cover(Buck_{i2})$. This is impossible as $\bigcup_{v=1 \cdots c} cover(Buck_{iv}) \equiv cover(Buck_i)$. Hence $|P_k| \leq |P_{k'}|$. □

Updating pairs of an existing tuple

So far we presented the computation, organization and minimization of the pairs of a tuple newly inserted into T. In the following, we explain the update process for a tuple inserted beforetime.

Let t_c, the current timestamp, be a maintenance timestamp. Let δ^+ be the transaction to be inserted into T. To simplify the comprehension, we explain in a first time the maintenance process for a valid tuple $r \in R$ inserted at the previous maintenance time, then we generalize for tuples inserted at any time. The sequence of buckets $Pairs(t)$ at t_c is $[Buck_1, \ldots, Buck_m]$. The maintenance process consists on two steps, on one hand, the pairs computed wrt outdated tuples must be deleted, on the other hand, new pairs are computed wrt the newly inserted transaction. Regarding the first step, i.e. deletion of pairs, the

pairs computed wrt outdated tuples are located in $Pairs(t).Buck_1$. Therefore, it suffices to delete $Buck_1$ from $Pairs(t)$. This step takes $O(1)$ time. For the ease of presentation, the oldest bucket is always denoted $Buck_1$. Hence at this step, the sequence of buckets of t is $Pairs(t) = [Buck_1, \ldots, Buck_{m-1}]$. Now regarding the second step, i.e. computation of pairs wrt newly inserted tuples, let P_{new} be the set of the new pairs computed wrt tuples in δ^+. We merge this set with the head of the sequence, i.e. $Pairs(t).Buck_{m-1}$. We proceed like this as the *lifetime* of pairs in P_{new} is the same as the *lifetime* of pairs in $Buck_{m-1}$. Indeed, observe that pairs in $Buck_{m-1}$ which are computed wrt to tuples having the same timestamp as t, remain in $Pairs(t)$ during the whole validity period of t. Hence, $Buck_{m-1}$ is discarded at the m^{th} maintenance period after the integration of t into T, which coincides with the timestamp t is discarded from T. Therefore, pairs in P_{new} and $Buck_{m-1}$ are discarded at the timestamp where t gets outdated.

We generalize the update process for tuples inserted into T at anytime before the current timestamp. Let t be such tuple, let $Pairs(t) = [Buck_1, \ldots, Buck_v]$ be its sequence of buckets such that $v < m$. Let δ^+ be the new transaction, then the update process is as follows, (i) delete $Pairs(t).Buck_1$ (ii) compute pairs wrt δ^+ and insert them into $Pairs(t).Buck_{v-1}$, and (iii) minimize $Pairs(t)$ by running Algorithm 13 with $Pairs(t)$ as input. We illustrate the update process in the following example.

Example 26. *Let us continue with tuples t_6 and t_7 for which the sequence of buckets has been computed at timestamp 7, see Example 24. Now, suppose the current timestamp is 9 and a new transaction $\delta^+ = \{r_8, r_9\}$ is inserted. We illustrate the update of $Pairs(r_6)$ and $Pairs(r_7)$ wrt to δ^+. Table 4.8 shows the running dataset T at timestamp 9, Tables 4.9 and 4.10 display the updated sequence of buckets of t_6 and t_7 at timestamp 9 respectively before and after the minimization process. Hereafter, We explain the steps for the update process for both tuples. First, observe that tuples t_2 and t_3 are removed from T; they are outdated. $Buck_1$ is deleted from $Pairs(r_6)$ and $Pairs(r_7)$ so both of them contain just two buckets. Then t_6 and t_7 are compared to t_8 and t_9 which produces respectively the pairs $\langle AC|\emptyset\rangle$ and $\langle C|A\rangle$, and $\langle A|\emptyset\rangle$ and $\langle \emptyset|AB\rangle$. These pairs are appended to $Buck_2$ which is the most recent bucket in the sequence, see Table 4.9. Then the minimization process is triggered which leads to the sequence of buckets depicted in Table 4.10. For t_6, the pair $\langle C|A\rangle$ in $Buck_2$ is discarded because it is covered by $\langle AC|\emptyset\rangle$ in $Buck_2$ as well. For t_7, the pair $\langle \emptyset|AB\rangle$ is discarded as it covers no subspace. In addition, $\langle AB|\emptyset\rangle$ is deleted from $Buck_1$ because all the subspaces it covers ($\{A, B, AB\}$) are jointly covered by more recent pairs $\langle B|A\rangle$ and $\langle A|\emptyset\rangle$ in $Buck_2$.*

Now that the data structures used in our framework and their maintenance are explained, we complete the presentation by showing in the next section, the query

Transaction	*Id*	*A*	*B*	*C*	*Arrival time*
$R[1]$	r_4	1	0	4	4
	r_5	0	1	5	5
$R[2]$	r_6	2	0	6	6
	r_7	2	1	1	7
$R[3]$	r_8	1	2	5	8
	r_9	2	1	4	9

Table 4.8: Dataset T at timestamp 9

Id	1	2
t_6	$\langle A\|B\rangle$	$\langle C\|A\rangle,\langle AC\|\emptyset\rangle$
t_7	$\langle AB\|\emptyset\rangle$	$\langle B\|A\rangle,\langle A\|\emptyset\rangle,\langle\emptyset\|AB\rangle$

Table 4.9: Pairs of t_6 and t_7 at timestamp 9 before minimization

Id	1	2
t_6	$\langle A\|B\rangle$	$\langle AC\|\emptyset\rangle$
t_7		$\langle B\|A\rangle,\langle A\|\emptyset\rangle$

Table 4.10: Pairs of t_6 and t_7 at timestamp 9 after minimization

answering process.

4.3.3 Query answering

Likewise NSC, we apply the subspace index presented in Section 2.2.3 to NSCt.

Example 27. *Let us take the pairs of t_6 at timestamp 9 as depicted in Table 4.10. The resulting index is illustrated in the following table.*

Subspaces	Pairs
AB	$\langle r_6\|B\rangle$
AC	$\langle r_6\|\emptyset\rangle$

Table 4.11: Indexation of pairs of t_6 at timestamp 9

Suppose a skyline query $Sky(AB)$ is issued then the pair $\langle r_6|B\rangle$ is processed. As $AB \neq B$ we deduce that $t_6 \notin Sky(AB)$. If instead query $Sky(ABC)$ is submitted then t_6 belongs to the result because there is no entry in the map related to a superset of ABC and where we can find t_6.

4.4 Experiments

We consider the following scenario in order to evaluate our proposal: a data analytics agency collects data from a stream provider and continuously issues subspace skyline queries for further processing. The stream configuration (θ, d) are imposed upstream. We

evaluate the ability of our proposal in responding to the agency needs in terms of subspace skyline answering, i.e., does our framework allow to answer subspace skyline queries with low query execution and maintenance times, and lightweight memory consumption? To assess the performance of our framework, we compare it (i) to a baseline approach which computes the skyline using state of the art algorithm BSkyTree [8, 7] and (ii) to *DBSky* together with its *Eager* algorithm [11], an approach for maintaining a *single* skyline over streaming data. The goal of this comparison is to show that (i) without any index structure, the best skyline algorithm known so far is unable to handle multidimensional skyline queries when the dimensionality is moderately large in a streaming context and (ii) streaming solutions targeting a single skyline cannot be generalized to multidimensional skyline queries.

The ability of our solution to handle streaming data is reflected by its throughput per time unit. More specifically, the number of queries it can answer between two consecutive batches. There are mainly four parameters that affect this throughput: (i) the flow of the insertions θ, (ii) the size of the sliding window ω, (iii) the batch interval k and (iv) the number of dimensions d. We vary the values of these parameters as shown in Table 4.12.

Parameters	Values
θ	$\{0.1sec, 1sec\}$
ω	$\{6h, 12h, 24h\}$
k	$\{5mn, 10mn, 20mn\}$
d	$\{8, 12, 16\}$

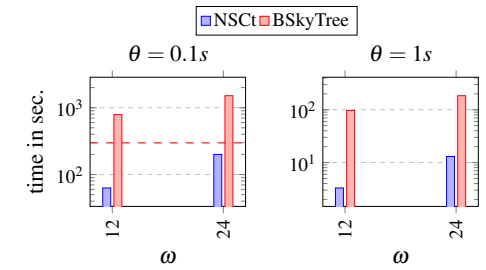
Table 4.12: Parameters values

Datasets: We generate synthetic independent (INDE) and anti-correlated (ANTI) data types using the framework of [2]. The generated tuples have either 8, 12 or 16 dimensions as depicted in Table 4.12. Moreover we consider a real stream of tweets where each tweet is described by five numerical attributes. More details in Section 4.4.5.

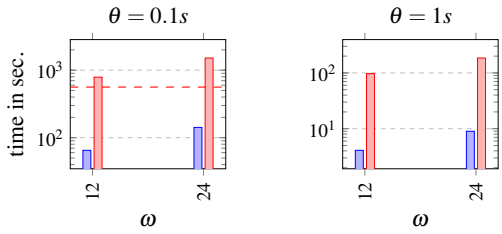
Implementation and hardware: All algorithms are implemented in C++. Source code is available on GitHub[3]. Experiments are performed on a Linux machine equipped with two 2.6GHz hexa-core processors and 32 Gb RAM.

First, we evaluate NSCt query answering performance and compare it to that of BSkyTree. The goal is to show that despite its maintenance process, NSCt is much more efficient. Second, we report the comparison to *DBSky* on memory consumption and maintenance time. Finally, we evaluate the impact of parameter k (batch interval) on both the maintenance time and the memory consumption of NSCt.

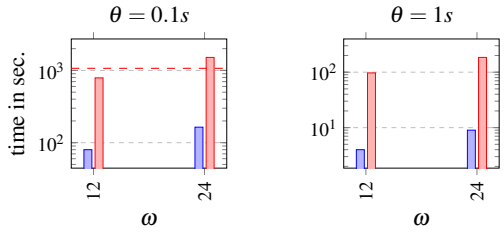
[2] https://twitter.com/
[3] https://github.com/karimalami7/MSSD



(a) $k = 5min$

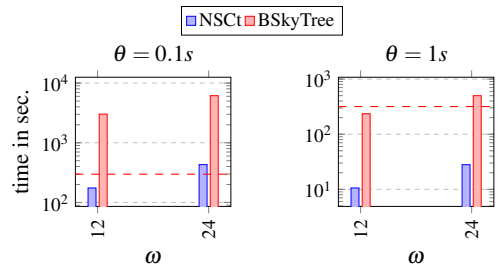


(b) $k = 10min$

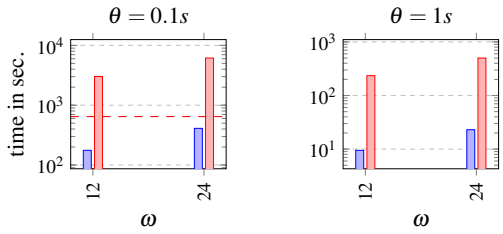


(c) $k = 20min$

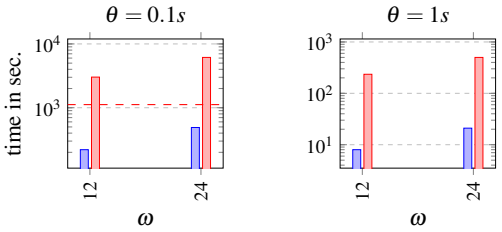
Figure 4.2: Execution time to answer $2^{12} - 1$ queries with independent data



(a) $k = 5min$



(b) $k = 10min$



(c) $k = 20min$

Figure 4.3: Execution time to answer $2^{12} - 1$ queries with anticorrelated data

For all experiments, we report the measures after *warm up*, .i.e., at a timestamp greater than ω so that the size of T becomes stable.

4.4.1 Query evaluation

The goal of this experiment is to evaluate the compatibility of NSCt maintenance delay when coping with streaming data. Indeed, if between two consecutive batches, most or all of the time is devoted to the maintenance, then NSCt becomes useless. To this aim, we compare NSCt to BSkyTree in term of query answering during a batch interval of size $k = \{5mn, 10mn, 20mn\}$. To fairly compare them, we report the time to answer all possible skylines with $d = 12$, i.e. 4095 queries.

Figures 4.2 and 4.3 depict the results with respectively independent and anticorrelated data. For both data types, we vary θ in $\{0.1s, 1s\}$ and ω in $\{12h, 24h\}$. Red dashed lines represent the value of k. When it is exceeded, it means that the approach cannot answer all issued queries during the batch interval. Recall that BSkyTree does not require any

maintenance so query evaluation can start as soon as a new transaction is inserted into T, while for NSCt we include the maintenance time. We point out two observations from this experiment:

1. NSCt is faster with more than one order of magnitude in all cases despite the fact that its maintenance time is also included.

2. BSkyTree is unable to answer all the issued queries for several scenarios, e.g. in Figure 4.2(a), with $\theta = 0.1$ sec., $k = 5mn$ and $\omega = 12$ hours, BSkyTree takes more than 5 minutes to answer all queries.

4.4.2 Time ratio

For this experiment, we consider the scenario where we have a workload set Q of random queries with $|Q| = \{10, 100, 1000, 10000\}$. All these queries are intended to be evaluated between two consecutive updates. We want to compare NSCt to BSkyTree. More precisely, our aim is to identify the situations where using an auxiliary structure like NSCt, which needs to be updated before starting query evaluation, is worthwhile. For this purpose, we report the following ratio $TR(Q)$:

$$TR(Q) = \frac{Maintenance\,Time\,of\,NSCt + NSCt\,Processing\,Time\,of\,Q}{BSkyTree\,Processing\,Time\,of\,Q}$$

When $TR(Q)$ is greater than one, BSkyTree is the best solution, otherwise one would prefer NSCt. We set $k = 5mn$ which reflects the number of inserted/deleted tuples (5×60 when $\theta = 1$sec. and 5×600 when $\theta = 0.1$ sec.) during NSCt maintenance. Figures 4.4 and 4.5 show the obtained results with respectively independent data and anticorrelated data. The general observation is that when $|Q|$ increases, $TR(Q)$ decreases. With approximately 100 queries, $TR(Q)$ is close to 1. Notice that this behavior is rather the same independently of the data correlation, θ and ω. For example, Figure 4.5 shows for $\theta = 1s$ that starting from $|Q| = 100$, $TR(Q)$ is less than 1 which means NSCt is $\frac{1}{TR(Q)}$ times faster than BSkyTree. However, with small $|Q|$, BSkyTree is faster. This indicates that using NSCt with its update delay is worthwhile when the number of queries is sufficiently large.

4.4.3 NSCt versus *DBSky*

We compare the memory consumption and maintenance time of NSCt to the approach described in [11] that consists of maintaining the skyline on D called *DBSky* and potential skyline tuples *DBrest*, i.e., those that have the potential to enter the skyline some time

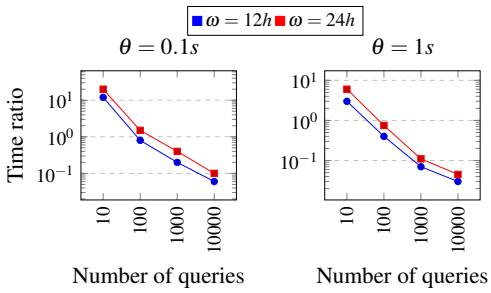


Figure 4.4: Time ratio with independent data

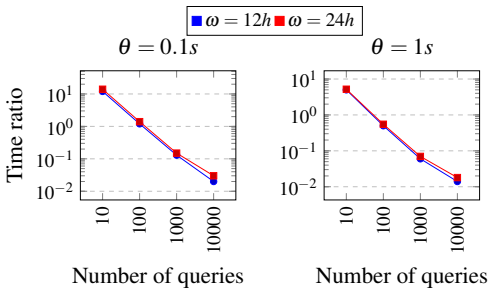
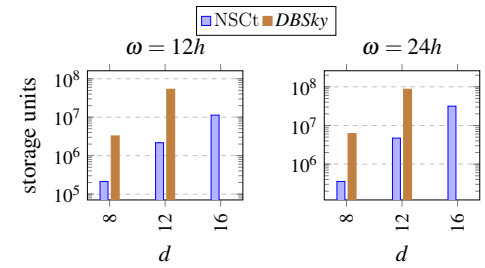


Figure 4.5: Time ratio with anticorrelated data

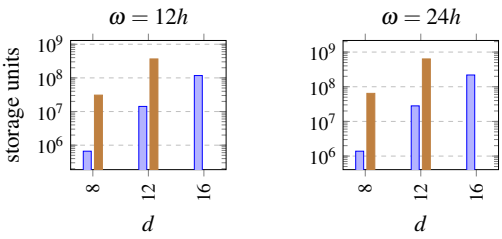
in the future once the current skyline tuples which dominate them expire. Note that meanwhile, new tuples can be inserted, hence a *DBrest* element is not guaranteed to become a skyline point. This proposal's target is to deal with just a single skyline. Obviously, its adaptation to the multidimensional setting we address with NSCt, consists in maintaining a *DBSky* and a *DBrest* for every subspace. For NSCt the memory usage corresponds to the number of pairs while for *DBSky* and *DBRest*, it represents the number of tuples stored. We set k to $20mn$, θ to 1 sec. and 0.1 sec., and repeat the measures by varying d in $\{8,12,16\}$ and ω in $\{12h,24h\}$. Figures 4.6 ,4.7 and 4.8, and 4.9 show the obtained results for respectively memory consumption and maintenance time. We note that we do not report some *DBSky* performances as it exceeded a reasonable execution time. One can observe that NSCt consistently uses less memory (see figures 4.6 and 4.7). However the memory consumption growth wrt to d is quite the same. A notable information here is that the growth wrt d of the set *DBSky* is higher than that of *DBRest* because the greater is d the bigger is the skyline set. In parallel, NSCt maintenance time is faster than *DBSky* on all configurations (see figures 4.8 and 4.9). We recall that for this experiments, the batch interval time k is set to 20 minutes (1200 seconds), therefore the maintenance time should be less than k in order to allow the user to issue queries. However, the maintenance time of *DBSky* is less than k for two configurations only, e.g. in figure 4.8, when $d = 8$, $\theta = 1sec$ and $\omega = 12h$. It even attains unreasonable execution time, e.g. in figure 4.8, when $d = 12$, $\theta = 0.1sec$ and $\omega = 12h$, the execution time is more than 2.10^5 seconds (55 hours). This make *DBSky* a non viable solution to deal with multidimensional skylines over streaming data.

4.4.4 NSCt maintenance time vs. memory consumption

We consider a stream with 12 dimensions and a delay $\theta = \{0.1s, 1s\}$. We are interested in querying a window of size $\omega = \{6h, 12h, 24h\}$. Hence, we evaluate the framework performance with respect to NSCt maintenance time and memory consumption with different values of $k = \{5mn, 10mn, 20mn\}$. Two observations can be made from Figures

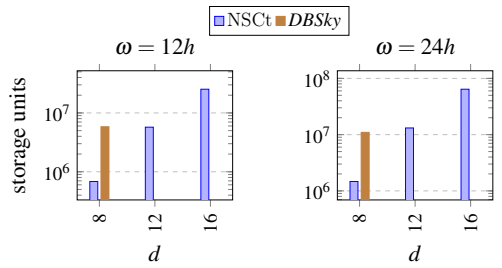


(a) $\theta = 1sec$

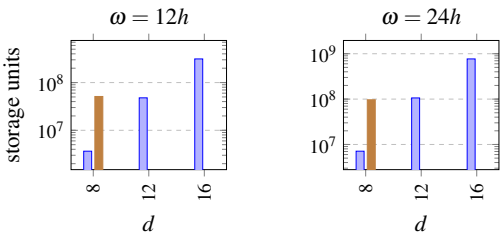


(b) $\theta = 0.1sec$

Figure 4.6: Memory usage with INDE data

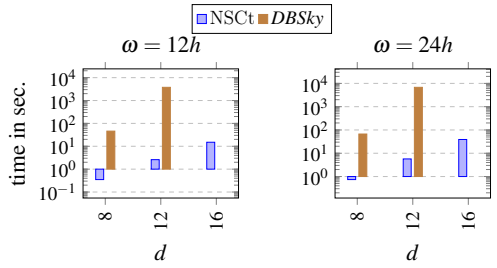


(a) $\theta = 1sec$

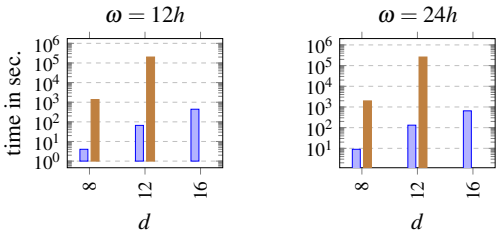


(b) $\theta = 0.1sec$

Figure 4.7: Memory usage with ANTI data

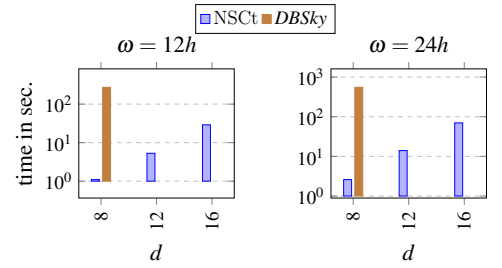


(a) $\theta = 1sec$

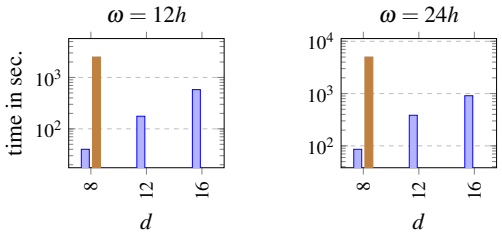


(b) $\theta = 0.1sec$

Figure 4.8: Maintenance time with INDE data



(a) $\theta = 1sec$



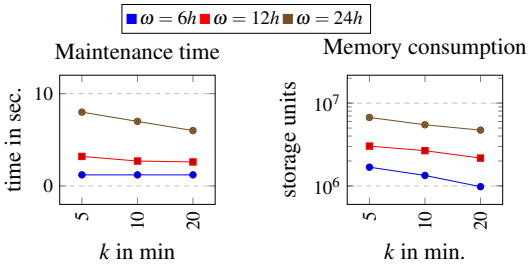
(b) $\theta = 0.1sec$

Figure 4.9: Maintenance time with ANTI data

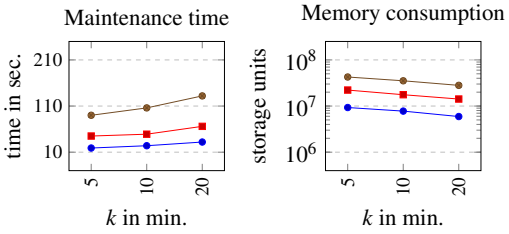
4.10 and 4.11.

1. On all cases, the memory used by NSCt decreases when selecting a bigger batch interval k. This behavior was expected by proposition 5.

2. The maintenance time ratio wrt k decreases. Let us take the hardest case depicted in Fig 4.11(b) and consider the result wrt $\omega = 24h$. For k=5 min, the maintenance lasts 210 seconds which represents two third of the batch interval while for k=20 min, the maintenance lasts 330 seconds, which represents a quarter of the batch interval.

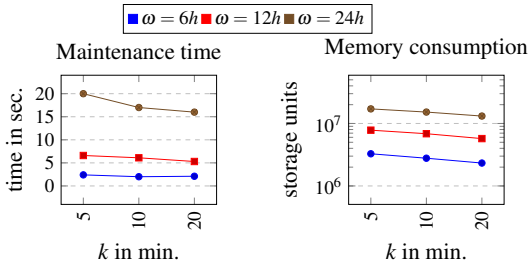


(a) $\theta = 1sec$

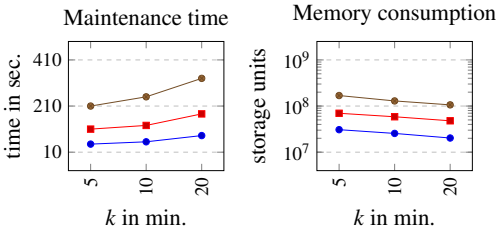


(b) $\theta = 0.1sec$

Figure 4.10: NSCt with INDE data



(a) $\theta = 1sec$



(b) $\theta = 0.1sec$

Figure 4.11: NSCt with ANTI data

This difference is due to a longer minimization process induced by a higher number of pairs when k is smaller.

4.4.5 Experiments with real data

In this section, we report on some experiments we conducted with real data describing tweets sent during a certain period. We obtained these data from an archive website. The archive is a temporal sequence of Json files each of which contains a description of a set of tweets sent during one minute. We parsed these files and, in addition to its timestamp, we retrieved for each tweet 7 attributes: TweetId, UserId, #followers, #following, #tweets, #likes, #lists. The five last attributes are describing users who sent the tweet: (i) #followers: # of people following UserId, (ii) #following: # of people UserId is following, (iii) #tweets: # of tweets the user has issued, (vi) #likes: # of tweets the user has liked, and (v) #lists: # of twitter lists the user is subscribed to. For all attributes, higher values are preferred. The goal is to retrieve at each time interval the *best* tweet users wrt any subset of these last five dimensions.

In the following experiments, we consider the batch interval $k = 1min$, i.e. we process a batch of tweets every one minute, and windows of different sizes $\omega = \{30min, 120min, 480min\}$, i.e, queries are evaluated over tweets tweeted in the last 30, 120, or 480 minutes.

[3] https://archive.org/

Remark 7. *Batches size vary between* 2000 *and* 3000 *tweets, i.e., the number of tweets in every processed file is not static. Hence, the number of tweets is less than* 90*k when* $\omega = 30min$*,* 360*k when* $\omega = 120min$*, and* 1.5*M when* $\omega = 480min$*. Moreover, we observed that the tweets data are highly correlated, hence skylines are rather small.*

Comparison to BSkyTree

Here we evaluate, as in Section 4.4.1, the compatibility of NSCt maintenance delay with streaming data. We compare NSCt to BSkyTree in term of query answering during the batch interval. To that purpose, we report the time to answer all possible skylines. In this case, $d = 5$ thus 31 queries. Figure 4.12 depicts the results. Globally NSCt is 10 times faster. Nevertheless, in the worst case, BSkyTree answers all possible queries in less than 1 second which is less than the batch interval k.

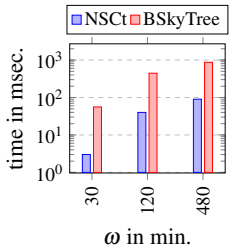


Figure 4.12: NSCt vs. BSkyTree with real data

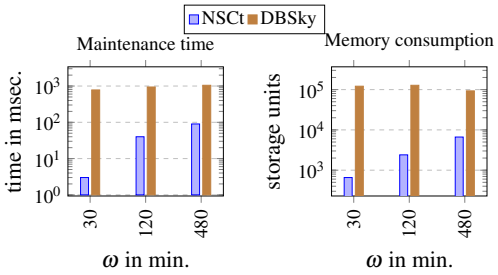


Figure 4.13: NSCt vs. DBSky with real data

Comparison to DBSky

Regarding the materialization aspect, we compare the memory consumption and maintenance time of NSCt to that of *DBSky*, as we did in Section 4.4.3. Figure 4.13 depicts the results. Firstly, we see that NSCt outperforms *DBSky* in both maintenance time and memory consumption. However, we see that *DBSky* results are not impacted by the growing ω. This is highly due to the fact that data is correlated, hence skylines are small and have same sizes even with larger input data.

4.4.6 Concluding remarks

As the previous experiments have shown and turning back to our motivating example concerning tweets, one may observe that our framework is capable to monitor Top-K influential tweets even with a relatively high frequency (e.g., 50 tweets per second), a large sliding window (e.g., 24h) a reasonable update frequency (e.g., every 5 minutes) and a data dimensionality not too small.

4.5 Conclusion

We have proposed a framework for processing subspace skyline queries on streaming data with a validity time window. The proposed approach consists of an index structure whose maintenance is triggered at regular time intervals. Since the queries evaluation is performed using the indexed data, their semantic is relative to the last update not the instant where the query is submitted. This introduces a kind of approximation regarding the results which is in conformance with standard streaming data evaluation algorithms [80, 39]. The conducted experiments demonstrate the effectiveness of our solution in terms of both memory consumption and its ability to speed up the queries evaluation in such a way that it can be considered as a viable technique in a streaming context.

Chapter 5

Optimization of regret minimization queries with NSC

5.1 Introduction

In this chapter, we conduct an experimental study on the optimization of the evaluation of regret minimization queries (RMS) by considering skyline related candidate sets. As presented in Chapter 1 Section 1.2, [17] proposed regret minimization queries to overcome the limitation of skyline queries and Top-K queries. However, their computation is challenging. One way to speed up their computation is by providing small candidate sets as input rather than the whole dataset. The challenge when providing smaller candidate sets is to guarantee the same quality of the output (regret) as if RMS were computed on top of the entire input dataset. [17] proved that the skyline constitutes a good candidate sets as the optimal solution of RMS is inevitably a subset of the skyline set. In this chapter, we investigate specifically the impact of providing the result of either Top-K frequent skyline (Top-KF) or Top-K priority skyline (Top-KP) queries as candidate sets for RMS algorithm *sphere*. Given D a set of attributes and T a dataset:

- Let $t \in T$, $Frequency(t) = |\{X \subseteq D \text{ s.t. } t \in Sky(T,X)\}|$. Top-K frequent skyline is then the K tuples with the highest frequency.
- Let $t \in T$, $Priority(t) = min_{X \subseteq D | t \in Sky(X)}(|X|)$. Top-K priority skyline is then the K tuples with the lowest priority.

We consider these queries because they are efficiently evaluated through NSC. Algorithm 14 describes the procedure to compute Top-KF through NSC. We compute the subspaces where a tuple t is dominated by computing the *cover* of all pairs related to t (line 4-7). We then deduce the frequency of each tuple and put it in list *Score* (line 8).

We sort *Score* and select Top K tuples (line 9-11). Algorithm for Top-KP is similar to Algorithm 14 with a difference in computing the score (line 8).

Algorithm 14: top-K_frequent

```
   Input: NSC, T, K, D
   Output: Top − KF
1  begin
2      Top − KF ← ∅
3      Score ← []
4      foreach t ∈ T in parallel do
5          E ← ∅
6          foreach p ∈ NSC[t] do
7              E ← E ∪ cover(p)
8          Score.append(t, 2^|D| − |E|)
9      sort(Score)
10     foreach i ∈ [0, K) do
11         Top − KF ← Top − KF ∪ Score[i].first
12 return Top − KF
```

Thus, the experiments we carry out in this chapter are: we calculate candidate sets of size K by either Top-KF or Top-KP then we compute a set of minimum regret of size r on top of these candidate sets. Our hypothesis is that by considering these candidate sets, we evaluate the regret minimization query faster, and the output regret will be close to if the regret minimization query is evaluated on top of the entire input dataset.

5.2 Experiments

In this section, we perform experiments to evaluate the impact of different candidate sets on computing the RMS. We proceed in three steps:

1. We evaluate the speed up of RMS computation by considering the skyline set as a candidate set.

2. We investigate the speed up and output regret of RMS algorithm *sphere* by considering Top-K Frequent and Top-K priority sets as candidate sets.

3. Given an integer K, we evaluate the output regret of sets computed by (i) Top-K frequent, (ii) Top-K priority sets and (iii) *sphere*.

Parameters	Values
distribution	ANTI, INDE
n (dataset size)	**100K**, $1M$
d(number of dimensions)	$4, \mathbf{8}, 12$
r(output size)	$20, \mathbf{30}, 40, 60, 80, 100$

Table 5.1: Datasets parameters

Hardware and software We consider the state of the art algorithm *sphere* [69] for computing regret minimizing sets and the structure NSC [23] for computing skyline related queries, i.e., (i) skyline, (ii) Top-K frequent and (iii) Top-K priority sets. All the experiments are conducted on a Linux machine equipped with two 2.6 ghz hexacore CPUs and 32GB RAM. Software is in C++ and available on GitHub[1].

Datasets We consider synthetic datasets generated through the framework in [2]. The parameters considered for these experiments and their values are illustrated in Table 5.1. Bold values are default values.

5.2.1 Speed up with skyline set

Here, we evaluate the speed up of *sphere* by considering the skyline set as input, i.e., given a dataset T, we run *sphere* on top of the whole dataset T and the skyline of T. Note that the output set and regret are the same whether we consider the skyline set or the whole dataset (Refer [17]). Hence we do not report the output regret. Figures 5.1 and 5.2 depict the results. We can see that the skyline enables faster computation of the minimum regret set on all cases. However, its benefit decreases with growing dimensions. Note that the computation time of the skyline set through NSC in negligible. The reported time is mostly the execution time of *sphere*. The main cause of the increasing computation time is that the skyline size grows rapidly with growing dimensions. For example, in Figure 5.1 for a dataset with 1 million tuples and independent distribution (blue curve), the skyline set goes from 418 tuples with 4 dimensions to 237726 tuples with 12 dimensions. Hence the speed up of *sphere* goes from more than 10^3 times to only 5 times faster.

We conclude that considering the skyline set as candidate set has a limitation, even if its computation time is negligible because its size is not controllable. In the next section, we investigate the impact of skyline related ranking queries, i.e. Top-KF and Top-KP, on *sphere*. The main motivation behind using these queries is that they provide (i) a controllable size of the output and (ii) their computation is optimized by NSC.

[1] https://github.com/karimalami7/NSC

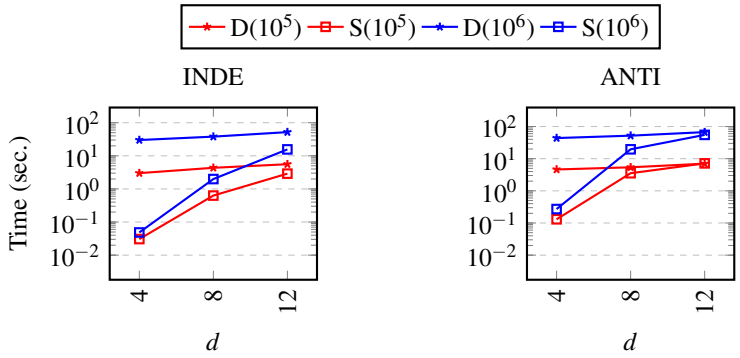


Figure 5.1: Speedup of *sphere* with skyline set as candidate set by varying dimensionality d

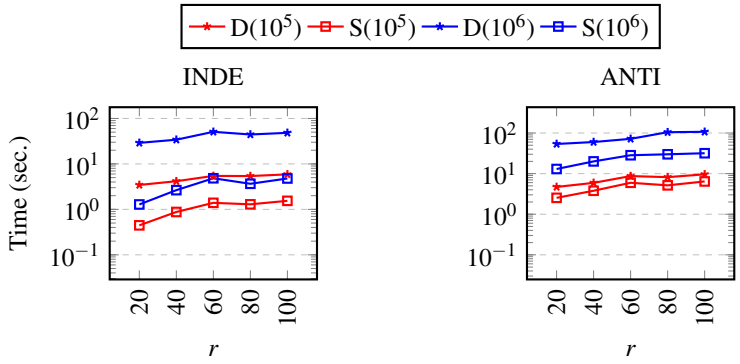


Figure 5.2: Speedup of *sphere* with skyline set as candidate set by varying the output size r

5.2.2 Speed up and regret of sphere with multidimensional skyline metrics as candidate sets

In this section, we evaluate the speedup of *sphere* by providing Top-KF and Top-KP sets as candidate sets. Figures 5.3, 5.4 depict the execution times. Figures 5.5 and 5.6 depict the output regrets.

Regarding computation time, we do not observe an apparent improvement by providing candidate sets Top-1% frequent tuples and Top-1% priority tuples. Indeed, in these settings *sphere* computation time is improved because the candidate sets are smaller and have constant sizes (1000 tuples). However the computation time of Top-1%F and Top-1%P is higher than that of skyline. For example, in Figure 5.3 with anti-correlated data and 12 dimensions, RMS computation takes 7 seconds on top of the skyline while it takes 5.5 seconds on top of Top-1%F. Regarding the first case, *sphere* alone takes approximately 6.9 seconds because the skyline approaches 95% of the whole dataset. While for the second case, *sphere* takes only few milliseconds. We note however that using these candidate sets is interesting for medium dimensionalities, i.e. $d \in [6, 12]$. For smaller d, the skyline is small, hence is a good candidate set. For higher d, Top-KF and Top-KP computation is high even with NSC.

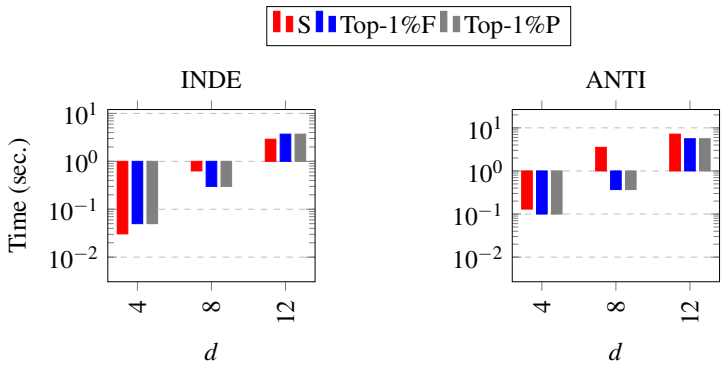


Figure 5.3: Computation time of *sphere* with candidate sets (i) skyline (ii) Top-K frequent and (iii) Top-K priority by varying d

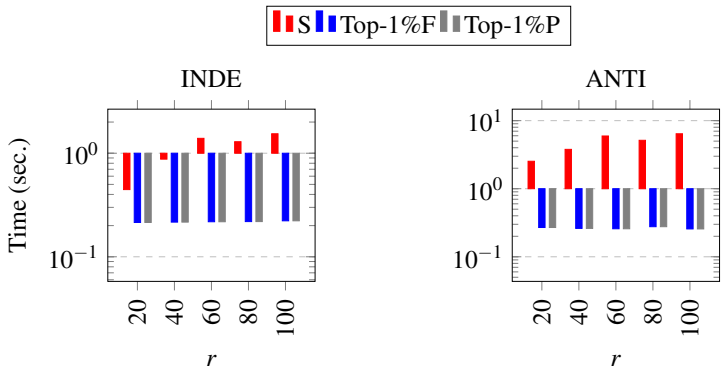


Figure 5.4: Computation time of *sphere* with candidate sets (i) skyline (ii) Top-K frequent and (iii) Top-K priority by varying r

Regarding the output regret ratio, we can see that regret ratios of all methods are close. We also observe in Figure 5.6 that for small r (under 60) when considering Top 1% frequent tuples as candidate sets, the regret ratio computed by *sphere* is better than that with skyline set. This is explained by the fact that *sphere* is a heuristic approach. Indeed, Top-1% frequent tuples discards some noisy points that are then not select by *sphere*.

5.2.3 Top-KF and Top-KP as alternatives to RMS algorithms

Above, we showed that Top-KF and Top-KP queries provide good candidate sets for *sphere*. In this section, we want to answer the question: Can Top-KF or Top-KP (without sphere) compute sets that achieve regret ratio close to that achieved by *sphere*? Concretely, we evaluate the regret ratios of sets of size K computed with (i)sphere (ii) Top-KF and (iii) Top-KP. For *sphere* we consider the skyline set as input. Figures 5.7 and 5.8 depict the results. Globally, we can see that *sphere* provides better regret ratio, which is expected as it is dedicated for RMS computation. However, Top-KF achieves a good regret ratio when dimensionality grows (Figure 5.7). Also, Top-KF achieves good regret ratio when K is small (Figure 5.8). We can explain this by the fact that tuple

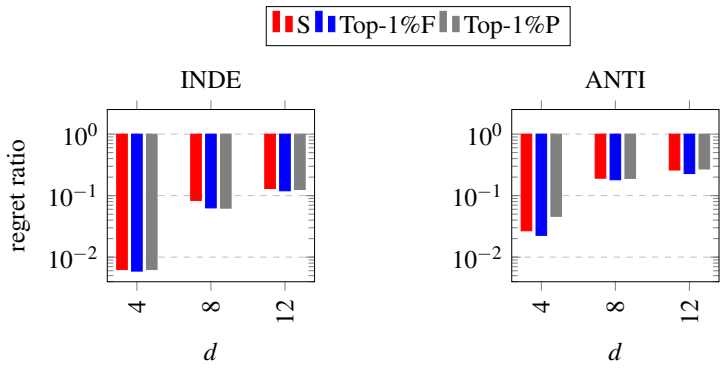


Figure 5.5: Regret of *sphere* by candidate sets (i) skyline (ii) Top-KF (iii) Top-KP by varying d

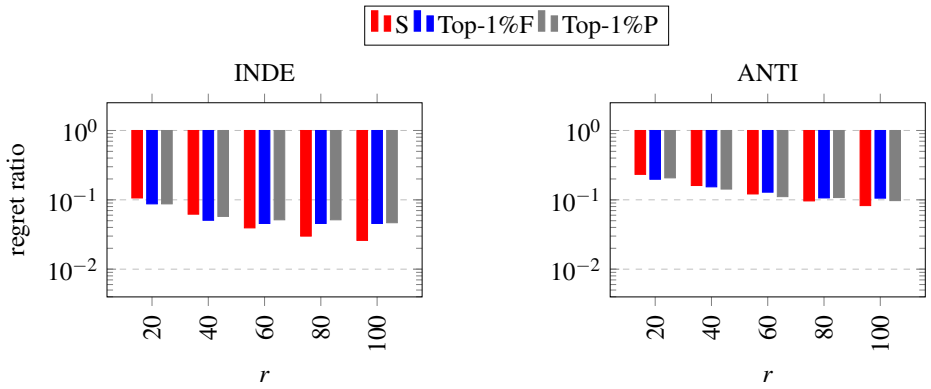


Figure 5.6: Regret of *sphere* by candidate sets (i) skyline (ii) Top-KF (iii) Top-KP by varying r

are better ranked in high dimensions. Indeed, suppose $d = 4$, the frequency domain is $[0,2^4 - 1]$=[0,15] which is small. Many tuples may share the same frequency and hence it is hard to rank them. The higher d, the larger the frequency domain, and so the better the ranking.

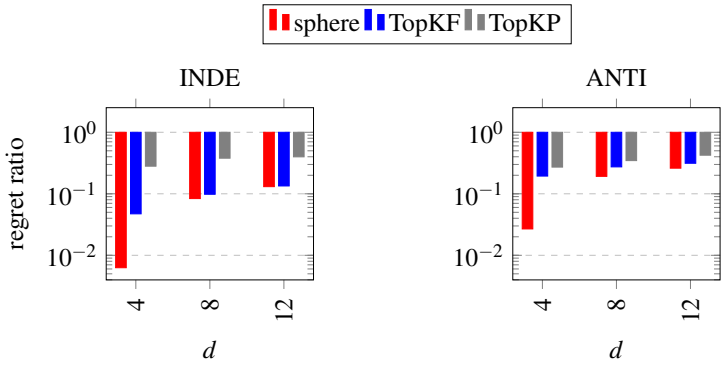


Figure 5.7: Regret of (i) *sphere* (ii) Top-KF (iii) Top-KP by varying d

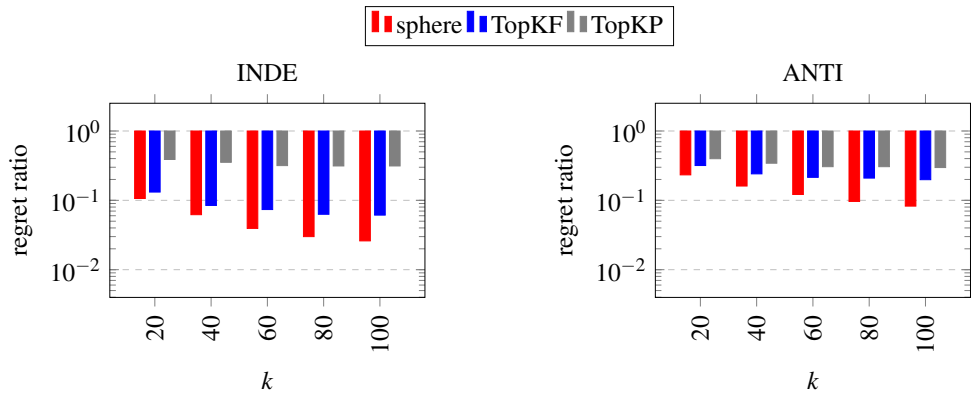


Figure 5.8: Regret of (i) *sphere* (ii) Top-KF (iii) Top-KP by varying k

5.2.4 Discussion

To summarize, we first have shown that sphere is improved by considering the skyline set as a candidate set, even with high dimensions as the skyline is computed through NSC. Second, we investigated Top-KF and Top-KP as candidate sets for *sphere*. Our experiments show that by selecting a small portion of the input data (1%) representing most frequent skyline points, not only the RMS computation is faster but its quality is sometimes better than that returned by the approximate algorithm *sphere* when it considers the whole skyline. From the experiments in Figures 5.7 and 5.8, we observe that the regret ratio computed by TopKF gets better with large dimensionalities d and small output size k. Of course, all these preliminary promising empirical results need to be confirmed theoretically. We are currently working on this aspect.

Part II

Skyline queries in presence of dynamic and partial orders

In Part I, we have considered datasets having totally and statically ordered attributes, i.e., attribute's domain is a totally ordered set. However, it is usual that datasets have their attributes' domain partially and dynamically ordered. Skyline is harder to compute in that setting because (i) traditional algorithms are not suitable with such datasets and (ii) materializing techniques are costly due to the high number of possible queries. For example, NSC is unsuitable in this situation. Hence, in this part, we address the problem of answering skyline queries with datasets having partially and dynamically ordered attributes. We provide efficient algorithms and materialization techniques that speed up the computation.

Chapter 6

On-the-fly algorithms and materialization technique

6.1 Introduction

In the previous chapters, we considered data having only numerical attributes. However in many real world use cases, datasets have nominal attributes for which no order is specified. Users express their preferences on the nominal attribute's domain. In such cases, NSC structure is not suitable, as it is built given a specific order. In this chapter, we address the optimization of skyline queries answering in presence of dynamic and partial orders.

First, we present the context of this study. Consider Table 6.1 where information about movies proposed by a *media-services provider* is registered. Movies are described by their genre and critic scores. Metacritic and Rotten Tomatoes are online platforms specialized in rating movies. Audience represents the score given by subscribers. A movie is in the skyline of Table 6.1 iff there does not exists any other movie better or equal to it wrt all four attributes, and at least strictly better on one attribute. While comparing movies regarding their respective ratings is natural, considering their genre is not immediate.

In fact, the order relationship among the values of each attribute's domain is expressed by a set of orders (preference) $\mathcal{R}$. Two aspects of $\mathcal{R}$ are relevant to skyline queries:

- $\mathcal{R}$ is either total or partial. $\mathcal{R}$ is total when every two values are ordered. Per contra, $\mathcal{R}$ is partial when there exists at least two values which are not comparable. To illustrate, consider Table 6.1. Metacritic, Tomatoes and Audience attributes have their respective domain in $\mathbb{N}$. Since larger ratings are preferred, the preference on each of these three attributes is the relation $>$ on $\mathbb{N}$ which is total. By contrast, for Genre attribute, its domain values have not to be totally ordered. E.g., one may

	Metacritic	Tomatoes	Audience	Genre
t_1	56	65	85	$c : comedy$
t_2	63	70	75	$s : sci-fi$
t_3	89	80	90	$h : horror$
t_4	70	72	88	$h : horror$
t_5	63	70	50	$r : romance$
t_6	45	42	80	$a : action$
t_7	52	69	75	$t : thriller$
t_8	64	74	52	$c : comedy$
t_9	73	80	90	$s : sci-fi$
t_{10}	81	71	84	$a : acion$

Table 6.1: Movie rating

prefer comedy over thriller but has no preference between comedy and sci-fi. These two last values are incomparable regarding the user's preference.

- $\mathcal{R}$ can be either static or dynamic. Again, consider Table 6.1. The orders on Metacritic, Tomatoes and Audience respective domains are unique and set a priori. While for Genre, the order depends on users preferences. More precisely, during their quest of the *best* movies, users are asked to express their own preference on Genre's domain in terms of an order relation.

Example 28. *Consider Table 6.1. While the preference on* Metacritic, Tomatoes *and* Audience *is: the higher the score the better the movie. For* Genre*, there is no prior preference over the attribute's domain. Users are asked to describe their preferences through a set of value to value comparability. One user preference could be* $\mathcal{R} = \{(horror, comedy),$
$(sci\text{-}fi, thriller)\}$ *which expresses that* horror *is preferred to* comedy*, and* sci-fi *is preferred to* thriller*. This preference makes comparable the movies having* comedy *or* horror *genre, i.e.,* $\{t_1, t_3, t_4, t_8\}$*. Likewise,* $\{t_2, t_7, t_9\}$ *are comparable because of* sci-fi *and* thriller *genres. The skyline set over the movie dataset by considering the user preference* $\mathcal{R}$ *expressed above is composed of* $\{t_3, t_5, t_9, t_{10}\}$*. The remaining tuples are dominated. For example,*

- t_1 *is not in the skyline because it is dominated by* t_3 *which has better scores and better genre (*horror *is preferred over* comedy*).*
- t_6 *is not in the skyline because* t_{10} *has better scores and both have the same genre* action*. Observe that this genre is not mentioned in* $\mathcal{R}$ *making* t_6 *comparable to only those tuples sharing the same genre.*

The skyline set changes dramatically with the user preference. Consider
$\mathcal{R}' = \{(romance, horror), (sci-fi, horror), (comedy, horror), (action, horror), (thriller, horror)\}$,

i.e. every genre is better than horror. The skyline set is then $\{t_1, t_2, t_3, t_5, t_6, t_7, t_8, t_9, t_{10}\}$. *Observe that* t_3 *belongs to the skyline set despite being a horror movie. This is because* t_3 *has the higher ratings in the dataset.*

As presented in Chapter 1 Section 1.1.4, previous work which investigated skyline computation with partially ordered attributes either proposed on-the-fly algorithms, i.e., computing the query from scratch, or proposed materialization techniques, i.e., precomputing some indexing structures. One of the techniques proposed so far to implement on-the-fly algorithms is: given a dataset with partially ordered attribute B, transform B into a set of totally ordered virtual attributes $\phi(B)$ and then run state of the art skyline algorithm on the transformed dataset [19]. Regarding materialization techniques, [56] proposed to compute and store the skylines wrt every total order over the attribute B. Then answer a query q which considers a preference $\mathcal{R}$ through the stored skylines. [22, 55] proposed indexes to cache skyline sets and their respective preferences $\mathcal{R}$'s then answer issued queries through *refinement*. We say that $\mathcal{R}'$ is a refinement of $\mathcal{R}$ iff $\mathcal{R} \subset \mathcal{R}'$. Accordingly the skyline wrt $\mathcal{R}'$ is included in the skyline wrt $\mathcal{R}$.

In this chapter, we exhibit a couple of properties letting the decomposition of every skyline query q, using a preference $\mathcal{R}$, into a set Q of *independent* sub-queries. The result of q is obtained by just combining the results of the sub-queries $q' \in Q$. Because these queries are independent from each others, we execute them in parallel. On another side, if all or some of these sub-queries are materialized, the computation time can be optimized.

More specifically, the main contributions of the present work are:

- A novel approach to compute skyline queries with partially and dynamically ordered attributes.
- A materialization technique to optimize skyline query answering.
- A workload driven selection of sub-queries to materialize.
- Extensive experiments showing the effectiveness of our proposals.

Chapter organization The next section presents the main definitions used throughout the chapter. Then we present our approach, first, in case of datasets with only one partially ordered attribute. Afterwards, we generalize to the case of multiple attributes. In section 6.4, we address the sub-queries materialization. Finally, we empirically evaluate our proposals wrt direct competitors.

6.2 Preliminaries

In this section, we define the additional concepts we use throughout the chapter. Some concepts such as dominance and skyline query are redefined.

The context of the problem we study is as follows: we have a set of dimensions (attributes) $\mathcal{D}$ composed of both totally and statically ordered dimensions $\mathcal{A} = \{A_1, \ldots, A_s\}$, and partially and dynamically ordered dimensions $\mathcal{B} = \{B_1, \ldots, B_l\}$. A dataset T over the set of dimensions $\mathcal{D}$. Users are interested in the skyline set of T by considering their preferences over $\{A_1, \ldots, A_s, B_1, \ldots, B_l\}$ domains.

We first define the *order* relation which expresses the user preference between **two values**.

Definition 13. *(Order) Let $D \in \mathcal{D}$, $dom(D)$ denotes its domain, and $d_i, d_j \in dom(D)$. $o = (d_i, d_j)$ is an order which expresses that d_i is preferred over d_j. We use as well the notation $d_i \prec_D d_j$*[1].

Definition 14. *(Preference) Let $D \in \mathcal{D}$. A preference $\mathcal{R}$ over D is a set of orders over $dom(D)$. $\mathcal{R}$ respects the following properties:*

- *transitivity: $(d_i, d_j) \in \mathcal{R}$ and $(d_j, d_k) \in \mathcal{R}$ then $(d_i, d_k) \in \mathcal{R}$.*
- *irreflexivity: $(d_i, d_j) \in \mathcal{R}$ then $(d_j, d_i) \notin \mathcal{R}$*

Observe that a preference over an attribute is nothing but a classical partial order relation defined on its domain.

Remark 8. *Recall that $\mathcal{D} = \mathcal{A} \cup \mathcal{B}$. Every $A_i \in \mathcal{A}$ is totally and statically ordered. For example, the preference over* Tomatoes *attribute in Table 6.1 is the relation $>$ over $\mathbb{N}$. So from now on, we consider only preferences defined on those attributes admitting dynamic partial orders over their respective domains, i.e., $B_i \in \mathcal{B}$.*

Example 29. *Consider the movie rating in Table 6.1. A user preference over the attribute **Genre** can be expressed by $\mathcal{R}$ ={(c,s),(s,h),(c,h),(c,r),(r,h),(h,t),(s,t),(r,t),(t,a),(s,a), (r,a),(h,a)}. Obviously, this preference can be represented by the DAG in Figure 6.1.*

Definition 15 (Dominance)**.** *Let T be a table over $\mathcal{D} = \{A_1, \ldots, A_s, B_1, \ldots, B_l\}$ and let $\mathcal{R} = \{R_1, \ldots, R_l\}$ be a preference over the attributes $B_1, \ldots, B_l$. Let t, t' be two tuples, then t dominates t' iff $t[D] \preceq_D t'[D]$ $\forall D \in \mathcal{D}$ and $\exists D \in \mathcal{D}$ such that $t[D] \prec_D t'[D]$. We denote the dominance relation by $t \sqsubset_{\mathcal{D}} t'$.*

Given a skyline query q, $q.\mathcal{R}$ denotes its related preference $\mathcal{R}$.

[1] We use the term *order* for ordering just a single pair of values.

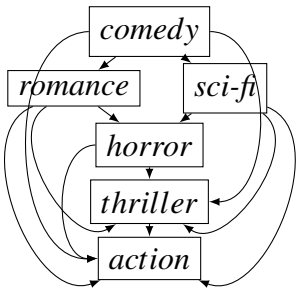


Figure 6.1: DAG representation of $\mathcal{R}$

Notation	Definition
$A_1, \ldots, A_s$	totally ordered attributes
$B_1, \ldots, B_l$	partially ordered attributes
$\mathcal{D}$	set of all attributes
m	size of $dom(B_i) \forall B_i \in \mathcal{B}$
$\mathcal{R} = \{R_1, \ldots, R_l\}$	preference over $B_1, \ldots, B_l$
o	an order
q	a skyline query
$q.\mathcal{R}$	preference of the query q
$Sky_{q.\mathcal{R}}(T, D)$	skyline set wrt q
Q	a workload

Table 6.2: Notations

Definition 16 (Skyline query)**.** *Given $\mathcal{D}$, T, and a query q. The skyline set $Sky_{q.\mathcal{R}}(T, \mathcal{D}) = \{t \in T \mid \nexists t' \in T \text{ s.t. } t' \sqsubset_{\mathcal{D}} t\}$ is the set of not dominated tuples. We denote also the skyline set by $Sky_{q.\mathcal{R}}(T)$ or simply $Sky_{q.\mathcal{R}}$ when T and/or $\mathcal{D}$ are clear from the context.*

Table 6.2 summarizes the additional notations used throughout the chapter.

In the next section we present the properties of skyline queries that we exploit to devise our solutions.

6.3 dySky algorithm

The objective of our work is to efficiently answer skyline queries q wrt user preference $q.\mathcal{R}$ over a dataset T. For the ease of the presentation, first, we consider datasets with only one dynamic dimension.

Our approach is based on the following property: given a query q and its related preference $q.\mathcal{R}$. A tuple which does not belong to the skyline set wrt a preference composed of some order in $q.\mathcal{R}$, does not belong to the skyline set wrt $q.\mathcal{R}$. More precisely,

Theorem 9. *Given $\mathcal{D} = \{A_1, \ldots, A_s, B\}$, a dataset T, and a query q. Let $t \in T$, then $t \notin Sky_{q.\mathcal{R}}$ iff $\exists o \in q.\mathcal{R}$ s.t. $t \notin Sky_{\{o\}}$.*

Proof. (i) $t \notin Sky_{q.\mathcal{R}} \Rightarrow \exists o \in q.\mathcal{R}$ s.t. $t \notin Sky_{\{o\}}$: let $t \notin Sky_{q.\mathcal{R}}$ then there exists a tuple t' dominating t such that either (i) $t'[B] = t[B]$ and $t' \prec_{\mathcal{A}} t$ or (ii) $t'[B] \prec_B [B]$ and $t' \preceq_{\mathcal{A}} t$. In the first case t' dominates t whatever the preference $q.\mathcal{R}$ hence $t \notin Sky_{\{o\}} \forall o \in q.\mathcal{R}$. For the second case, $t \notin Sky_{\{(t'[B], t[B])\}}$.

(ii) $\exists o \in q.\mathcal{R}$ s.t. $t \notin Sky_{\{o\}} \Rightarrow t \notin Sky_{q.\mathcal{R}}$: $t \notin Sky_{\{o\}}$ means there exists a tuple t' such that $t' \sqsubset_{\mathcal{D}} t$. Whatever the remaining orders in $q.\mathcal{R}$, $t' \sqsubset_{\mathcal{D}} t$. □

The above theorem states that a tuple t does not belong to the skyline wrt to a given preference $q.\mathcal{R}$ if and only if t does not belong to the skyline wrt a *singleton* preference composed of some order in $q.\mathcal{R}$.

We introduce here the notation of **complementary** skyline or shortly c-skyline. Given a query q, its c-skyline is $NSky_{q.\mathcal{R}}$, the set of dominated tuples wrt $q.\mathcal{R}$.

To summarize, by computing those tuples not belonging to the skyline wrt every preference composed of some order in $q.\mathcal{R}$, i.e., $NSky_{\{o\}}\, \forall o \in q.\mathcal{R}$, we deduce $NSky_{q.\mathcal{R}}$ as stipulated in the following corollary.

Corollary 10. *Given a query q.*

$$NSky_{q.\mathcal{R}} = \bigcup_{\forall o \in q.\mathcal{R}} NSky_{\{o\}}$$

Proof. From theorem 9. Let $t \in T$.

$t \in NSky_{q.\mathcal{R}} \Leftrightarrow t \in NSky_{\{o_1\}} \vee \cdots \vee t \in NSky_{\{o_n\}}$ s.t. $o_1, \ldots, o_n \in q.\mathcal{R}$ then $t \in \bigcup_{\forall o \in q.\mathcal{R}} NSky_{\{o\}}$ □

Example 30. *Consider the movie dataset in Table 6.1. Given a query q with $q.\mathcal{R} = \{(c,s),(s,h),(c,r),(c,h)\}$ then $NSky_{q.\mathcal{R}} = NSky_{\{(c,s)\}} \cup NSky_{\{(s,h)\}} \cup NSky_{\{(c,r)\}} \cup NSky_{\{(c,h)\}} = \{t_2, t_5\}$.*

The skyline is then $T \setminus \{t_2, t_5\} = \{t_1, t_3, t_4, t_6, t_7, t_8, t_9, t_{10}\}$.

Even though computing $NSky_{q.\mathcal{R}}$ requires to compute as many queries as the number of orders in $q.\mathcal{R}$, in the next section we show that these queries are actually easy to evaluate making the whole computation efficient.

6.3.1 Single dynamic dimension

In this section, we present an algorithm for computing $Sky_{q.\mathcal{R}}$. We consider a table T with $\mathcal{D} = \{A_1, \ldots, A_s, B\}$ where B is the unique partially and dynamically ordered dimension.

Theorem 9 and its corollary 10 suggest an algorithm for evaluating queries $Sky_{q.\mathcal{R}}$ on T: it evaluates sub-queries, i.e., the c-skyline by considering every $o \in q.\mathcal{R}$. The union of the sub-queries results is $NSky_{q.\mathcal{R}}$ and thus its complement to T is the response to $Sky_{q.\mathcal{R}}$.

The bottleneck of this direct implementation belongs to the multiple computations of $NSky_{\{o\}}$. Before presenting our solution, let us first make the following observation: Let o be an order, and let t be a tuple whose value in the dynamic dimension B is not mentioned in the preference $\{o\}$. Let $t' \in T$. Then

- $t \sqsubset_{\mathcal{D}} t' \Rightarrow t[B] = t'[B]$
- $t' \sqsubset_{\mathcal{D}} t \Rightarrow t[B] = t'[B]$

Said differently, and from the domination relationship, these tuples whose B value does not belong to o can be comparable to only those tuples sharing the same value on B. For example, consider the query q related to a *singleton* preference $q.\mathcal{R} = \{(horror, thriller)\}$ stating that *thrillers* are preferred to *romances* but there is no preference among the remaining genres. The tuples whose genre does not belong to the above two are comparable to only those with the same genre. Hence, we can partition them and restrict the comparisons to the so obtained subsets. To continue the example, we get the partition $\{\{t_1,t_8\}_c, \{t_2,t_9\}_s, \{t_5\}_r, \{t_6,t_{10}\}_a\}$. The first part $\{t_1,t_8\}_c$ corresponds to the tuples whose genre is $c(omedy)$. To this partition we can add a special part containing the remaining tuples, i.e., $\{t_3,t_4,t_7\}$. Now, each part can be processed independently to check whether a tuple is dominated or not. For example, comparing t_3 to t_5 is needless because they belong to different parts.

To summarize, computing the c-skyline wrt a *singleton* preference consists in partitioning the data into subsets of comparable tuples and identify those dominated within each subset. We formalize the above statement in Proposition 6, but first we define a dataset *part*.

Definition 17 (Part). *Given $\mathcal{D}$ and T. Let $D \in \mathcal{D}$. A part of T wrt a value d of D, denoted $\Pi_{[D|d]}(T)$, is the set $\{t \in T | t[D] = d\}$.*

Proposition 6. *Given $\mathcal{D} = \{A_1, \ldots, A_s, B\}$ and T. Let $b_i, b_j \in dom(B)$ and q such that $q.\mathcal{R} = \{(b_i, b_j)\}$.*

$NSky_{\{(b_i,b_j)\}}(T) = NSky_{\{(b_i,b_j)\}}(\Pi_{[B|b_i]}(T) \cup \Pi_{[B|b_j]}(T))$
$\cup \bigcup_{\forall b_k \in dom(B)} NSky_{\{(b_i,b_j)\}}(\Pi_{[B|b_k]}(T))$ *where* $b_k \neq b_i, b_j$.

Example 31. *Again, consider the movie dataset in Table 6.1. Let q be a skyline query s.t. $q.\mathcal{R} = \{(horror, thriller)\}$ then*

$$NSky_{q.\mathcal{R}}(T) = NSky_{\{(h,t)\}}(\Pi_{[genre|h]}(T) \cup \Pi_{[genre|t]}(T))$$
$$\cup NSky_{\{(h,t)\}}(\Pi_{[genre|c]}(T)) \cup NSky_{\{(h,t)\}}(\Pi_{[genre|s]}(T))$$
$$\cup NSky_{\{(h,t)\}}(\Pi_{[genre|r]}(T)) \cup NSky_{\{(h,t)\}}(\Pi_{[genre|a]}(T))$$
$$= \{t_1, t_3, t_8, t_9, t_{10}\}$$

Algorithm dySky_1d Algorithm 15 is the implementation of Proposition 6. First the variable *NSky* which will store the dominated tuples is initialized (line 2). Then for each order $(b_i, b_j) \in q.\mathcal{R}$, the algorithm computes the c-skyline in two steps: (i) it computes P, the subset of tuples having the values b_i, b_j on dimension B, then computes dominated tuples in P (line 4-5). (ii) It iterates on all values $b_k \neq b_i, b_j$ of the domain of B, partitions T wrt to these values, and compute the dominated tuples within each partition (line 6-7). $Sky_{q.\mathcal{R}}$ is only $T \setminus NSky$.

Algorithm 15: dySky_1d

Input: a set of dimensions $\mathcal{D} = \{A_1, \ldots, A_s, B\}$, a dataset T, a query q
Output: $Sky_{q.\mathcal{R}}$

```
1 begin
2     NSky ← ∅
3     foreach (b_i, b_j) ∈ q.R in parallel do
4         P ← Π_{[B|b_i]}(T) ∪ Π_{[B|b_j]}(T)
5         NSky ← NSky ∪ NSky_{{(b_i,b_j)}}(P)
6         foreach (b_k) ∈ dom(B) \ {b_i, b_j} do
7             NSky ← NSky ∪ NSky_{{(b_i,b_j)}}(Π_{[B|b_k]}(T))
8 return T \ NSky
```

Complexity analysis First, we consider the time complexity for evaluating a skyline query wrt some dataset of size v and over c dimensions as $O(v^2 \cdot c)$. Likewise the time complexity for evaluating the complementary skyline. Now, regarding our algorithm, let $n = |T|$ be the size of the dataset T, $s+1$ be the number of its dimensions and m be the number of values in $dom(B)$. Consider that the values in $dom(B)$ are uniformly distributed over T. Then the size of each part wrt B is $v = \frac{n}{m}$. Let q be a query. The algorithm iterates on all orders in $q.\mathcal{R}$, hence $|q.\mathcal{R}|$ iterations. For every (b_i, b_j) in $q.\mathcal{R}$, (i) it partitions and computes the c-skyline wrt every value in $dom(B)$, hence $O(m \cdot (\frac{n}{m})^2 \cdot s)$, and (ii) it partitions and computes the c-skyline wrt values b_i and b_j, hence $O((2\frac{n}{m})^2 \cdot s)$. The overall time complexity is then $O(|q.\mathcal{R}| \cdot (\frac{n}{m})^2 \cdot s)$.

In the next sections, we highlight two properties that we implement in Algorithm 16, an optimized version of Algorithm 15.

The extended preference

Observe in Algorithm 15 (lines 6-7) that for every order (b_i, b_j), we compute a non skyline set for every $b_k \notin \{b_i, b_j\}$. According to this observation, we modify our algorithm so that every b_k not appearing in any order of a query is processed only once. We achieve this by extending the input preferences as follows:

Definition 18 (Extended Preference)**.** *Let $\mathcal{R}$ be a preference on dimension B. Let $U(\mathcal{R})$ denotes the values in $dom(B)$ not mentioned in $\mathcal{R}$. The extended preference $\hat{\mathcal{R}}$ is $\mathcal{R} \cup \{(b_i, b_i) | \forall b_i \in U(\mathcal{R})\}$.*

Intuitively, adding these “artificial" orders forces Algorithm 15 to compare the tuples sharing a same value not mentioned in a preference $\mathcal{R}$. Therefore, the nested loop in Lines 6-7 can now be completely removed from that Algorithm since the outerloop (line 3) already handles those values b_k, provided that as input we have an extended preference. In the following, we consider that all preferences are extended.

Incrementally discarding dominated tuples

Observe that given two orders o_1 o_2,

$$NSky_{\{o_1\}}(T) \cup NSky_{\{o_2\}}(T) = NSky_{\{o_1\}}(T \setminus NSky_{\{o_2\}}(T))$$

The tuples which do not belong to the skyline wrt order o_2, i.e. $NSky_{\{o_2\}}(T)$, should not be reconsidered for computing $NSky_{\{o_1\}}$.

We implement Algorithm 16 according to the above properties.

Algorithm 16: dySky_1d_optimized

Input: a set of dimensions $\mathcal{D} = \{A_1, \ldots, A_s, B\}$, a dataset T, a query q
Output: $Sky_{q.\mathcal{R}}$

```
1 begin
2     T' ← T
3     foreach (b_i, b_j) ∈ q.R in parallel do
4         P ← Π_[B|b_i](T') ∪ Π_[B|b_j](T')
5         T' ← T' \ NSky_{(b_i,b_j)}(P)
6 return T'
```

The complexity of Algorithm 16 remains the same as that of Algorithm 15, however in practice, these modifications show enhancement in performance.

	A_1	A_2	A_3	B_1	B_2
t_1	1	0	1	b_{11}	b_{21}
t_2	0	0	1	b_{11}	b_{22}
t_3	1	1	1	b_{11}	b_{21}
t_4	1	2	1	b_{13}	b_{21}
t_5	1	0	2	b_{12}	b_{23}

Table 6.3: Dataset with two dynamic dimensions

R_1	R_2
$o_{11} = (b_{11}, b_{12})$	$o_{21} = (b_{21}, b_{22})$
$o_{12} = (b_{11}, b_{13})$	$o_{22} = (b_{22}, b_{23})$
	$o_{23} = (b_{21}, b_{23})$

Table 6.4: The preference $q.\mathcal{R}$

6.3.2 Multiple dynamic dimensions

In this section, we present our approach for datasets with multiple partially and dynamically ordered dimensions, i.e. $\mathcal{D} = \{A_1, \ldots, A_s, B_1, \ldots, B_l\}$. We recall that in this case, the preference $\mathcal{R}$ is composed of preferences over every dimension, i.e. $\mathcal{R} = \{R_1, \ldots, R_l\}$. Corollary 11 is a consequence of Theorem 9 when considering multiple partially ordered dimensions.

Corollary 11. *Given $\mathcal{D} = \{A_1, \ldots, A_s, B_1, \ldots, B_l\}$, a dataset T, and a query q such that $q.\mathcal{R} = \{R_1, \ldots, R_l\}$. Let $t \in T$, then $t \notin Sky_{q.\mathcal{R}}$ iff $\exists (o_1, \ldots, o_l) \in R_1 \times \cdots \times R_l$ s.t. $t \notin Sky_{\{(o_1, \ldots, o_l)\}}$.*

As said in section 6.3.1, an algorithm which naively computes $Sky_{\{(o_1,\ldots,o_l)\}} \forall (o_1, \ldots, o_l) \in R_1 \times \cdots \times R_l$ does not take advantage of skyline properties. Firstly and for the ease of the presentation, we detail our approach in case of two partially ordered dimensions, then we generalize to the case of l partially ordered dimensions. Consider the dataset and the preference $\mathcal{R}$ depicted respectively in Tables 6.3 and 6.4. Note that smaller values are preferred. Likewise the case of one partially ordered dimension, our approach consists in computing the sets of comparable tuples $\mathcal{T}$ wrt the preference $q.\mathcal{R}$ and then deduce the dominated tuples. To that purpose, we proceed as follows: (i) we compute the subsets of tuples T_1 and T_2 having respectively values in o_{11} and o_{12}, i.e., the orders belonging to the preference related to the first dimension B_1, (ii) from T_1 and T_2, we compute the subsets of tuples having respectively values in o_{21}, o_{22} and o_{23}, i.e., the orders in the preference over dimension B_2. We illustrate this process in Figure 6.2. Let $\mathcal{T}$ be the set of the so computed subsets. Then a tuple t belongs to the skyline wrt T iff it does not belong to any complementary skyline of T' $\forall T' \in \mathcal{T}$. For

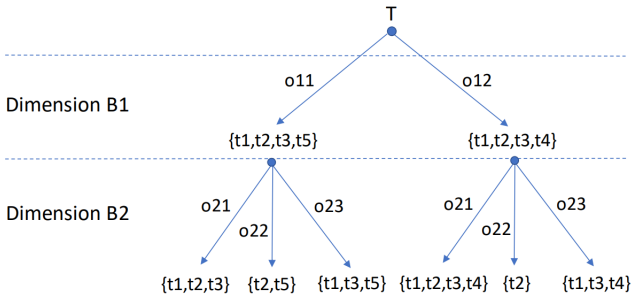


Figure 6.2: Processing q

example, in Figure 6.2, the c-skyline of the subset in the right most leaf is $\{t_3, t_4\}$, hence, $t_3, t_4 \notin Sky_{q.\mathcal{R}}$. One may verify that the union of the c-skylines is $\{t_3, t_4, t_5\}$ and therefore, $Sky_{q.\mathcal{R}} = T \setminus \{t_3, t_4, t_5\}$.

We formalize and generalize the above explanation in the following result.

Proposition 7. *Given $\mathcal{D}$, T, and a query q such that $q.\mathcal{R} = \{R_1, \ldots, R_l\}$. Let $\mathcal{O} = R_1 \times \cdots \times R_l$. Then*

$$NSky_{q.\mathcal{R}}(T) = \bigcup_{o \in \mathcal{O}} NSky_{\{o\}}(\bigcap_{i=1}^{l} \Pi_{[B_i | b_e \vee b_f \text{ s.t. } b_e, b_f \in o_i]})$$

Intuitively, the above proposition simply states that by computing the dominated tuples in each obtained subset, we get the set of all dominated tuples. Hence, the skyline set.

Remark 12. *One may notice that the obtained subsets do not form a partition. For example, the right most sub-tree in Figure 6.2, we have two sets containing t_1, t_3 and t_4. This means that these tuples are compared twice and each time, t_3 and t_4 are found dominated by t_1. To avoid this redundant computation, it suffices to remove the dominated tuples from the underlying data as soon as possible. So, the right most subset will actually contains only t_1.*

Now we present how we translate Proposition 7 to a concrete algorithm.

Algorithm dySky_md The algorithm takes as input T and a query q, and returns $Sky_{q.\mathcal{R}}$. It is composed of a main routine and a recursive procedure called *recursiveNSky*. The variable *NSky* stores the complementary skyline throughout the process. It is initialized by an empty set. The variable i indicates the dimension the algorithm is currently processing. In the beginning, i is set to 1, hence the process starts with dimension B_1. The algorithm calls the procedure *recursiveNSky* with the arguments: (i) i, i.e. which

Algorithm 17: dySky_md

Input: a set of dimensions $D = \{D_1, \ldots, D_s, B_1, \ldots, B_l\}$, a dataset $T(D)$, a query q

Output: $Sky_{q.\Re}(T)$

```
1 Procedure recursiveNSky(i, T', NSky)
2     foreach o ∈ R_i in parallel do
3         T'' ← Π_{[B_i|o.left]}(T') ∪ Π_{[B_i|o.right]}(T')
4         if i < l then
5             recursiveNSky(i+1, T'', NSky)
6         else
7             NSky ← NSky ∪ NSky(T'')
1 begin
2     NSky ← ∅
3     i ← 1
4     recursiveNSky(i, T, NSky)
5 return T \ NSky
```

indicates the first dimension, (ii) the dataset T and (iii) *NSky* (line 4). Regarding the procedure *recursiveNSky*, for each order o in R_i, it filters T' wrt o (line 3), then if $i < l$, i.e. the algorithm is not processing the last dimension, it recalls *recursiveNSky* with new parameters(line 5). Otherwise ($i = l$), i.e. the algorithm is currently processing the last dimension B_l, it computes the complementary skyline wrt T'' and add it to *NSky* (line 7). Finally, the skyline wrt the query q is T minus *NSky* (line 5 in the main routine).

Complexity analysis Given the parameters m, l, n and s. Suppose the preferences on the dynamic dimensions have the same number of orders r, i.e., $|R_i| = r$, $\forall i \in [1..l]$. At each level, *dySky_md* iterates on r orders. Globally, the algorithm iterates r^l times. We consider the filtering operations take a constant time. The argument here is that one can use bitmap indexes on the B_i's dimensions. The final step consists in computing the complementary skyline. In case of uniform distribution of the values in $dom(B_i) \forall i \in [1..l]$, at the last level of filtering, the datasets T' contains $\frac{n}{m^l}$ tuples. Then the overall complexity is $O(r^l * (\frac{n}{m^l})^2 * (s+l))$. When the preferences R_i's are total, r equals $\frac{m(m-1)}{2}$. In such case, this algorithm's complexity becomes that of a naive algorithm, however, in practice *dySky_md* performs better.

6.4 Optimization using materialization

As we have seen so far, the main idea of *dySky* algorithm is to take a query q and decompose it into a set of sub-queries q_i. Each of them operates on a subset of T obtained by a sequence of filters. For example, let us consider again query q (see Table 6.4) from the previous section. In order to answer q, we compute 6 complementary skylines, i.e. 6 sub-queries, as illustrated in Figure 6.2. Consider the left most subset in that Figure which is obtained by the filter sequence $((b_{11},b_{12}),(b_{21},b_{22}))$. Consider the sub-query q_1 which computes the complementary skyline regarding this subset. Suppose now that the answer of q_1 is materialized. Then whenever q is issued, we get the answer of q_1 automatically. Likewise, the queries sharing the same sub-query q_1 are optimized thanks to this materialization. Obviously, by materializing all possible sub-queries, we optimize all possible queries. This solution is practical only for cases where the number of possible sub-queries is reasonable. When this number is too large, a pragmatic solution is to materialize a subset of these sub-queries. The choice of the best subset should be driven by a query workload. This is the problem we address in this section.

Firstly, we give some definitions needed for this section. Then we address the full materialization of the sub-queries, i.e. we consider that there is no limitation on memory space and we materialize all possible sub-queries. Later, we consider the case where memory space is restricted, and we address the partial materialization of the sub-queries, i.e, we materialize a set of sub-queries under space constraint.

6.4.1 Materialization structure

Each sub-query q_i is uniquely identified by a filtering sequence seq_i. Before defining a sequence, we firstly define the set of orders wrt a dimension B_i.

Definition 19. *Given a partially ordered dimension B_i.*
$Orders(B_i) = \{(b_{ij},b_{ik}) \in dom(B_i) \times dom(B_i)\}$ is the set of all possible orders wrt B_i.

It is easy to see that $|Orders(B_i)| = |dom(B_i)|^2$. Given a set of partially ordered dimensions $\mathcal{B} = \{B_1,\ldots,B_l\}$, a sequence is an *l-tuple* which belongs to the Cartesian product $Orders(B_1) \times \cdots \times Orders(B_l)$. Formally speaking,

Definition 20 (Sequence)**.** *Given $\mathcal{B} = \{B_1,\ldots,B_l\}$. A sequence is an element of $Orders(B_1) \times \cdots \times Orders(B_l)$. Consequently, the set of all possible sequences is $\Sigma(\mathcal{B}) = \{seq | seq \in Orders(B_1) \times \cdots \times Orders(B_l)\}$.*

Clearly, $|\Sigma(\mathcal{B})| = \Pi_{i=1}^{l} |Orders(B_i)|$. Hereafter, we note just Σ when $\mathcal{B}$ is understood.

Example 32. *Consider again Table 6.3. We have* $dom(B_1) = \{b_{11}, b_{12}, b_{13}\}$ *and* $dom(B_2) = \{b_{21}, b_{22}, b_{23}\}$. *One possible sequence is* $((b_{13}, b_{11}), (b_{22}, b_{22}))$. Σ *contains in total* 81 *sequences.*

The sub-queries materialization structure is a set of pairs (seq_i, CS_i) such that seq_i is the filtering sequence related to a query q_i and CS_i is the complementary skyline wrt the filtered data.

Definition 21 (seqStruct)**.** *Given* $\mathcal{D} = \{A_1, \ldots, A_s, B_1, \ldots, B_l\}$ *and* T.

$seqStruct = \{(seq_i, CS_i) | seq_i \in \Sigma \text{ and } CS_i \subseteq T\}$.

Finally, given a query q, $sequences(q)$ is the set of sequences related to q. Formally speaking,

Definition 22 (Sequences related to a query)**.** *Given* $\{B_1, \ldots, B_l\}$ *and a query* q *such that* $q.\mathcal{R} = \{R_1, \ldots, R_l\}$.

$$sequences(q) = \{seq \in R_1 \times \cdots \times R_l\}$$

.

Example 33. *Consider* $q.\mathcal{R}$ *depicted in Table 6.4. It has 6 related sequences. E.g.,* (o_{11}, o_{21}) *and* (o_{11}, o_{22}).

6.4.2 Full materialization

In a nutshell, the process to materialize all possible sub-queries is to iterate on all possible sequences seq_i in Σ, to filter data wrt seq_i and to compute the complementary skyline to be stored in *seqStruct* $\mathcal{F}$. Algorithm 18 (*dySkySeq_build*) designed to this aim proceeds in a smarter way. Intuitively, one may observe that each sequence is actually a conjunction of conditions and several conditions may share the same conjunct prefix. For example, the sequences (o_{11}, o_{21}) and (o_{11}, o_{22}) share the same prefix o_{11}. To filter T wrt these two sequences, we first consider o_{11}. The result is then used for both o_{21} and o_{22}.

Algorithm dySkySeq_build This procedure (see Algorithm 18) takes $\mathcal{D}$ and T as input, and returns a *seqStruct* $\mathcal{F}$. At the beginning, $\mathcal{F}$ is empty. Variable i indicates the dimension the algorithm is processing and is initialized to 1. Variable *seq* is a stack structure and is used to store the sequences. The algorithm proceeds in a Depth-First fashion. It calls the recursive function *recursiveSeq* with parameters (i) i to indicate the dimension B_i, (ii) T, (iii) *seq*, and (iv) the set $\mathcal{F}$ (line 5). Inside *recursiveSeq*, T' is filtered wrt B_i and o

is pushed onto *seq* (line 3-4). If $(i < l)$, i.e. , the algorithm is not processing the last dimension, it recalls *recursiveSeq* with new parameters (line 6). Otherwise, i.e. $(i = l)$, at this step, *seq* contains l orders. Hence it computes the complementary skyline wrt T'' and inserts the pair $(seq, NSky_{\{seq\}}(T''))$ in $\mathcal{F}$ (line 8). Finally, o is popped from the sequence (line 9).

Algorithm 18: dySkySeq_build

Input: a set of dimensions $\mathcal{D} = \{A_1, \ldots, A_s, B_1, \ldots, B_l\}$, a dataset T
Output: *seqStruct* $\mathcal{F}$

1 **Procedure** *recursiveSeq*$(i, T', seq, \mathcal{F})$
2 **foreach** $o \in Orders(B_i)$ ***in parallel*** **do**
3 $T'' \leftarrow \Pi_{[B_i|o.left]}(T') \cup \overline{\Pi_{[B_i|o.right]}(T')}$
4 $seq.push(o)$
5 **if** $i < l$ **then**
6 *recursiveSeq*$(i+1, T'', seq, \mathcal{F})$
7 **else**
8 $\mathcal{F} \leftarrow \mathcal{F} \cup (seq, NSky_{\{seq\}}(T''))$
9 $seq.pop(o)$
1 **begin**
2 $\mathcal{F} \leftarrow \emptyset$
3 $i \leftarrow 1$
4 $seq \leftarrow \emptyset$
5 *recursiveSeq*$(i, T, seq, \mathcal{F})$
6 **return** $\mathcal{V}$

Query answering We describe here how to evaluate a query using $\mathcal{F}$. Algorithm 19 (*dySkySeq_qa*) takes as input $\mathcal{F}$, T, and a query q and returns $Sky_{q.\mathcal{R}}$. The algorithm simply merges the complementary skylines associated to sequences related to the query q.

Algorithm 19: dySkySeq_qa

Input: a query q, $\mathcal{F}$, T
Output: $Sky_{q.\mathcal{R}}$

1 **begin**
2 $NSky \leftarrow \emptyset$
3 **foreach** $seq \in sequences(q)$ **do**
4 $NSky \leftarrow NSky \cup \mathcal{F}[seq]$
5 **return** $T \setminus NSky$

6.4.3 Constrained materialization

Generally, materializing all the sub-queries can be costly. In this section we address partial materialization of the sub-queries, i.e, we materialize only a subset $\mathcal{P} \subseteq \mathcal{F}$. However, we want to select the sequences in $\mathcal{P}$ such that the answering cost of a workload $\mathcal{Q}$ is optimal. Without any constraint, the solution to this problem is obvious: materialize all and only the sequences related to $\mathcal{Q}$. Even when considering just $\mathcal{Q}$ and not all possible queries, the storage space may become prohibitive. So, we constrain the query cost optimization problem with an available memory storage H that has not to be overtaken by the chosen sequences to be materialized.

We start by defining the costs of answering queries and workloads. Then we present the partial materialization problem.

Query answering cost

We set the answering cost of a query q as the number of sequences related to q, namely,

$$Cost(q) = |sequences(q)|$$

The rationale behind this choice of cost function is that, under uniform distribution, the size of the filtered data from which the complementary skyline is computed is the same whatever is the sequence.

Now consider a set of materialized sub-queries $\mathcal{P}$, then the cost of answering q through $\mathcal{P}$ is

$$Cost(q,\mathcal{P}) = Cost(q) - |\{p \in \mathcal{P} | p.seq \in sequences(q)\}|$$

In other words, partial materialization saves query execution time proportionally to the number of sub-queries that are already materialized. The cost of a workload $\mathcal{Q}$ wrt $\mathcal{P}$ is defined accordingly:

$$Cost(\mathcal{Q},\mathcal{P}) = \sum_{q \in \mathcal{Q}} Cost(q,\mathcal{P})$$

Note that with the above definitions, when using full materialization, the cost of any query is null, thus $Cost(\mathcal{Q},\mathcal{F}) = 0$. This reflects the fact that retrieving a query answer is done without any effort.

In the next section, we formalize the problem of partial materialization of sub-queries, and we provide an algorithm to select the set $\mathcal{P}$.

Sequence selection problem

As said previously, the obvious way to optimize a workload $\mathcal{Q}$ is to cache the results of the sub-queries related to $\mathcal{Q}$. Storing all these results may require a storage space larger than the available one H. So, one needs to select a subset fitting H.

Remark 13. *Given a seqStruct $\mathcal{M}$, the required space to store $\mathcal{M}$, noted $res(\mathcal{M})$, is the total required space for storing complementary skylines related to sequences in $\mathcal{M}$.*

The sequence selection problem we address is,

> **Problem SS** Given $\mathcal{D}$, T, a workload $\mathcal{Q}$, a *seqStruct* $\mathcal{S}$ related to $\mathcal{Q}$, and an integer $H \geq 0$, compute a set $\mathcal{M} \subseteq \mathcal{S}$ such that $res(\mathcal{M}) \leq H$ and $Cost(\mathcal{Q}, \mathcal{M})$ is minimum.

A dynamic programming algorithm Our problem can be solved exactly by a 0-1 Integer Linear Programming problem

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} g_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq W, \\ & x_j \in \{0,1\}, \quad j = 1, \ldots, n \end{aligned} \tag{6.1}$$

We set n as the size of $\mathcal{S}$ and W as H. The weight vector $(w_1, \ldots, w_n)$ equals $(|p_1.CS|, \ldots, |p_n.CS|) \forall p_i \in \mathcal{S}$. The gain g_i represents the number of queries in $\mathcal{Q}$ which have the sequence $p_i.seq$ in their respective set of sequences. It is defined by the following formula.

$$g_i = Gain(p_i, \mathcal{Q}) = |\{q \in \mathcal{Q} | p_i.seq \in sequences(q)\}|$$

It is well known that 0-1 linear programs can be solved by dynamic programming techniques (e.g., see [81]). Its precise complexity, regarding our setting, is $O(|\mathcal{S}| * H)$.

In the present setting, i.e., partial materialization, when a query is submitted, it is first decomposed into a set of sub-queries. Some of them can be already materialized, thus their result is already available. The others are evaluated from scratch. The Algorithm 20 *dySkySeq_hybrid* implements this procedure.

Algorithm 20: dySkySeq_hybrid

Input: a query q, a *seqStruct* $\mathcal{M}$ and a dataset T
Output: $Sky_{q.\mathcal{R}}$

Procedure $computeSeq(i, T', seq, NSky)$
  $T'' \leftarrow \Pi_{[B_i|seq[i].left]}(T') \cup \Pi_{[B_i|seq[i].right]}(T')$
  if $i < l$ **then**
    $computeSeq(i+1, T'', seq, NSky)$
  else
    $NSky \leftarrow NSky \cup NSky_{\{seq\}}(T'')$

begin
  $NSky \leftarrow \emptyset$
  foreach $seq \in sequences(q)$ **do**
    if $seq \in \mathcal{M}$ **then**
      $NSky \leftarrow NSky \cup \mathcal{M}[seq]$
    else
      $computeSeq(1, T, seq, NSky)$
return $T \setminus NSky$

6.5 Experiments

In this section we compare our proposals to relevant literature techniques. We consider both non materialization and materialization based solutions. For the first family, we consider the algorithm *CPS* proposed by [19] as a representative solution. We recall that *CPS* transforms partially ordered dimensions into totally ordered dimensions. We combine it with BSkyTree [7, 8] in order to compute the skyline over the transformed dataset. For materialization-based techniques, we consider *Ordered Skyline Tree (OST)* structure [56]. In the remainder, we denote by *OST* both the structure and its corresponding algorithm for answering queries. Moreover we consider the query answering through refinement technique as presented in [55, 22] and we denote it by *Ref*. We adapted the BSkyTree algorithm and its authors implementation so that it returns the complementary skyline which is the main procedure of our solutions.

The experiments are organized in three parts:

1. In Section 6.5.1 and regarding query answering time, we evaluate algorithms which answer queries on the fly, i.e., *dySky_md* and *CPS* as well as those using pre-computed structures, i.e., *dySkySeq_qa* and *OST*.

2. In Section 6.5.2 and for pre-computation based techniques, we compare their respective structure build time and their memory consumption.

3. We show the ability of *dySky* to compete with the refinement strategy *Ref* proposed in [55, 22]. We consider the case where a set of queries is cached, and we measure the answering time of another set of queries by both techniques. Refer to Section 6.5.3.

4. Finally, we evaluate other specific aspects of *dySky* in Section 6.5.4. Specifically, we assess the linear cost function of answering queries presented in Section 6.4.3. We evaluate the impact of partial materialization of sub-queries on the query answering performance, and we evaluate the benefit of multithreading for *dySky*.

Hardware and software Experiments are conducted on a machine equipped with 96 cores cadenced with a frequency up to 3.40 Ghz. By default and when possible, computation is parallelized over 96 threads. This machine is also equipped with 1 TB RAM and running CentOS Linux. Regarding software, we use BSkyTree authors version. All remaining techniques implementations are ours. The software is coded in c++ and the source code is available on GitHub[2].

Datasets We use both synthetic datasets, through the framework of [2] with independent (INDE) and anti-correlated (ANTI) distribution, and real datasets commonly used in the skyline literature. The real datasets are initially composed of numerical attributes, thus we extend them with nominal attributes. The values of these attributes are randomly and uniformly generated.

For synthetic data, Table 6.5 shows the different parameters. Bold values are the default.

Parameter	Values
n (dataset size)	$100K, \mathbf{1M}, 10M$
s (static dims)	**6**
l (dynamic dims)	$1, \mathbf{2}, 3$
m (dynamic dims values)	$10, \mathbf{15}, 20$
distribution	**ANTI**, INDE

Table 6.5: Synthetic datasets

Table 6.6 shows the characteristics of the real data in addition to their respective skyline size wrt the totally ordered dimensions.

Queries generation In some of the following experiments we need to generate random queries. These are completely defined by their respective preferences on the B_i's

[2] https://github.com/karimalami7/dySky

Parameter	POKER	IPUMS	HOUSE
n	1M	75836	127931
s	11	10	6
l	2	2	3
m	15	30	10
$\|Skyline\|$	14131	3852	127931

Table 6.6: Real datasets

attributes. Each preference on a dimension B_i is actually a DAG whose set of nodes is $dom(B_i)$. Thus, we generate random DAGs on $dom(B_i)$ following a *density* parameter $\rho \in [0,1]$. Let's recall its definition. Let $G = (V,E)$ be a DAG, then the density of G is $\rho(G) = \frac{2|E|}{|V| * (|V|-1)}$. That's, the denser is G the more the values in $dom(B_i)$ are comparable. By default, we set $\rho = 0.5$.

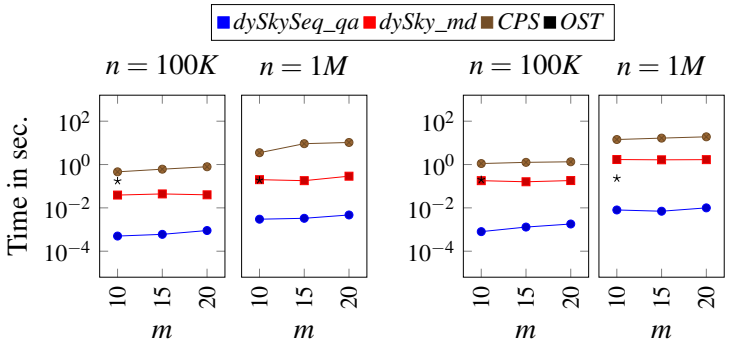
6.5.1 Query answering time

Here we compare our solutions to its competitors in terms of query answering time. In each case, we execute a same workload of 50 queries and we report the average execution time of all solutions. Sometimes *OST* values are not reported either because its related structure saturated the available memory or its execution did not terminate in a reasonable time ($>$ 24 hours).

Varying n, m, l and data distribution

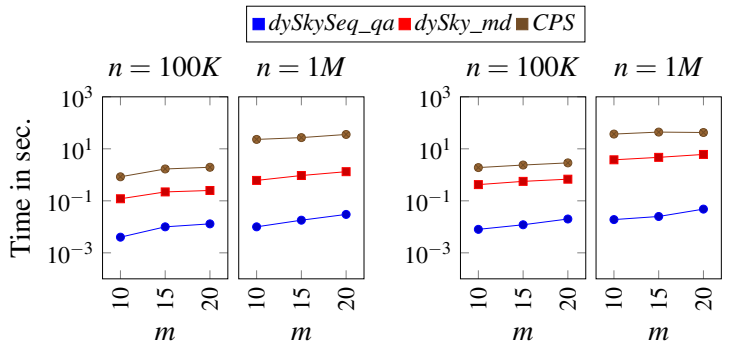
Figures 6.3, 6.4 and 6.5 depict the results with respectively 1, 2 and 3 dynamic dimensions. A first observation is that *OST* fails to build its structure in many configurations. When its structure can be built, the query answering time of *OST* is close to non materialization-based approaches *CPS* and *dySky_md* (see Figure 6.3). Regarding *CPS*, we observe that *dySkySeq_qa* and *dySky_md* perform better with (i) larger and/or (ii) anti-correlated datasets, i.e., the harder cases. For example, in Figures 6.3, 6.4 and 6.5, for $n = 1M$, *dySky_md* and *dySkySeq_qa* are respectively about one and three orders of magnitude faster than *CPS*.

Figure 6.6 depicts the query answering times for a dataset of $10M$ tuples and by varying both l and m. Globally, we can see that both *dySkySeq_qa* and *dySky_md* have better performances than *CPS*, however *dySkySeq_qa* scales less good than the two other solutions well wrt l. This trend suggests that materialization would be of no great added value with higher values of l, say $l \geq 6$.



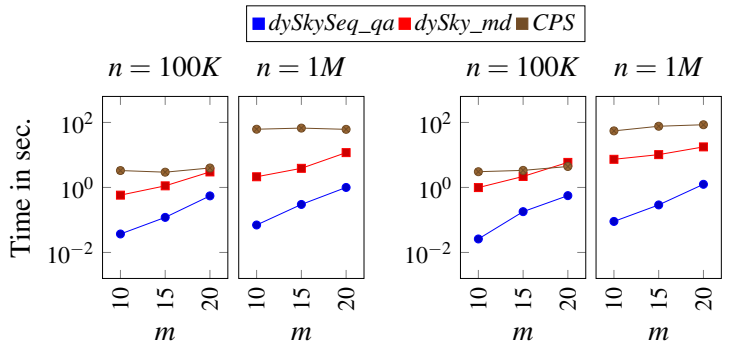
(a) INDE (b) ANTI

Figure 6.3: Query answering with $l = 1$



(a) INDE (b) ANTI

Figure 6.4: Query answering time with $l = 2$



(a) INDE (b) ANTI

Figure 6.5: Query answering time with $l = 3$

Varying the preferences density ρ

We generate queries whose $\rho \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$. Figure 6.7 depicts the results with a dataset having the default parameters. We see in the results that for a low density, *dySky* outperforms *CPS* by nearly two orders of magnitude. The gap tends to become smaller as the density grows. Recall that the lower the density, the lower the number of orders, while for *CPS*, the lower the density, the higher the number of dimensions in the transformed

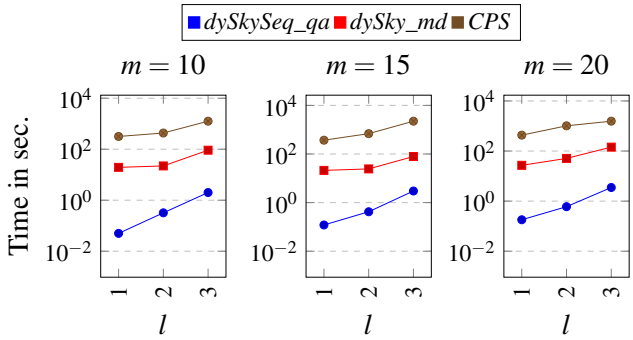


Figure 6.6: Query answering time with $10M$ tuples

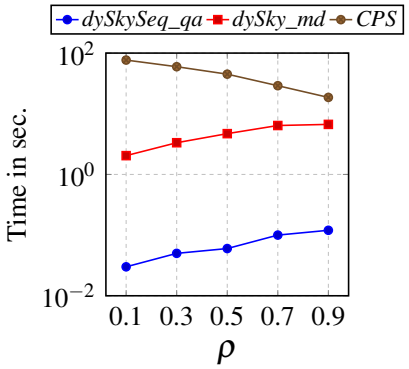


Figure 6.7: Query answering time by varying the preference's density ρ

dataset.

Querying real data

Figure 6.8 shows the obtained results. These confirm the previous findings, i.e., *dySky* with its two versions, clearly outperforms *CPS*.

6.5.2 Precomputation time and storage

In this section we compare the precomputation time and storage of both $\mathcal{F}$ and *OST* structure related respectively to *dySkySeq_qa* and *OST* algorithms. Regarding precomputation time, for $\mathcal{F}$ we measure the execution time of Algorithm 18 *dySkySeq_build* and for *OST* we measure the time of building the whole tree. W.r.t storage, we count the total number of tuples stored by each technique. Figure 6.9 depicts the obtained results with a dataset having one partially ordered dimension. We see that *OST* can not terminate when $m > 10$. When $m = 10$, the gap is large between *OST* and *dySky* wrt both time and storage. Results wrt datasets having more than one dynamic order are not reported as *OST* did not terminate for any configuration. This is due to the high size of the tree when both l and m grow as explained in Section 1.1.4.

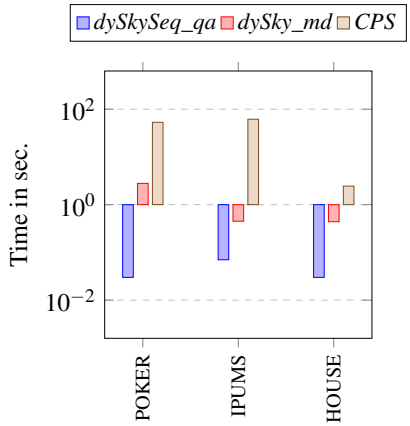
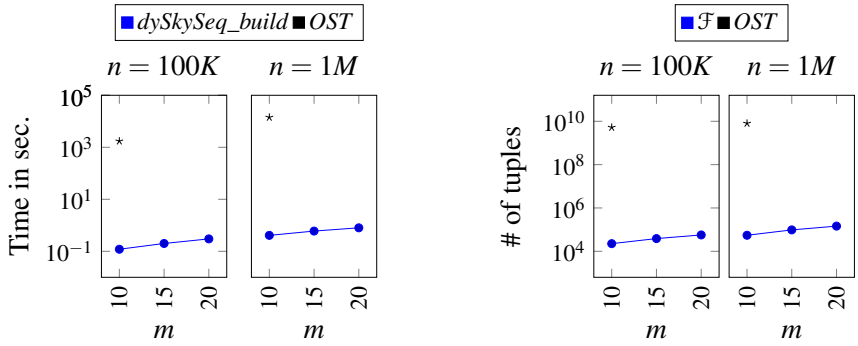


Figure 6.8: Query answering time with real data



(a) Precomputation time

(b) Storage

Figure 6.9: Precomputation with one dynamic dimension

6.5.3 Caching queries

In this experiment we show the ability of *dySky* to compete with the refinement strategy proposed in [55, 22] to optimize the queries via caching. To this aim, we consider the following scenario: Firstly, a set of queries Q_1 is selected randomly and its result is cached. Recall that for a query $q \in Q_1$ *dySky* caches the results of sub-queries related to q while *Ref* caches the result of q. Then a second set Q_2 of queries is evaluated using the previously cached results. Regarding *Ref*, a query $q \in Q$ can benefit from the cache iff there exists q' in the cache which is a refinement of q while following *dySky*, q benefits from the cache if at least one of its sub-queries is cached. We conducted experiments by varying the size of Q_1. We set $n = 100K$, $m = 10$ and $l = 2$, and a set Q_2 of 50 queries. Figure 6.10 reports the average execution time of queries in Q_2. As it may be observed, in all cases *dySky* provides better execution times than *Ref* making it a serious candidate to be used in a caching context. More precisely, Figure 6.10 shows that query answering time of *Ref* does not improve considerably when caching more queries. While *dySky_hybrid* performance improves until $|Q_1| = 100$ then it remains almost constant with larger $|Q_1|$.

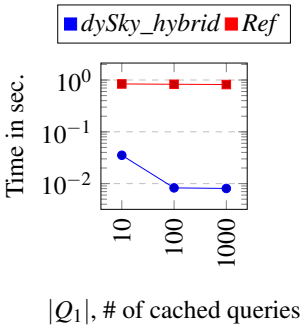


Figure 6.10: *dySky_hybrid* vs. *Ref*.

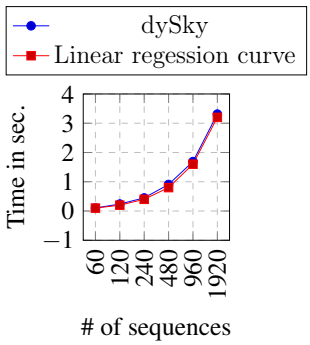


Figure 6.11: Query answering cost

This is explained by the fact that the maximum number of distinct sub-queries ($m^{2l} = 10^4$) can be reached with few queries. Figure 6.10 suggests that with $|Q_1| = 100$, the number of distinct sub-queries becomes close to 10^4, i.e., all queries in Q_2 are completely optimized. However, for *Ref* technique, even with a workload of 1000 queries, a refinement is hardly found for queries in Q_2.

6.5.4 Evaluating other aspects of dySky

In this section, we evaluate specific aspects of *dySky*.

Query answering cost estimation

In section 6.4.3, we have set the cost of answering a query q to be the number of sequences related to q. In this experiment, we want to confirm this supposition. To that purpose, we evaluate a set of queries each having a different number of sequences. For this experiment, we consider a dataset with $n = 100K$, $l = 2$ and $m = 10$. We generate 6 queries having respectively 60, 120, 240, 480, 960 and 1920 related sequences. The blue curve in Figure 6.11 depicts the obtained results, and the red curve is used to show the linear trend. We can see that the curves overlap, hence, the cost of answering a query q is clearly linear wrt its number of sequences $sequences(q)$.

Query answering time with partial materialization of the sub-queries

We consider the following scenario: we want to optimize the answering of a workload $\mathcal{Q}$. Let $\mathcal{P}$ be the *seqStruct* containing only sequences involved in $\mathcal{Q}$ and let M be the size of $\mathcal{P}$. Obviously, if we store $\mathcal{P}$, queries in $\mathcal{Q}$ are completely evaluated through materialized sub-queries. Now we consider the cases where the available memory size is equal to fractions of M, i.e. $\frac{M}{2}$, $\frac{M}{4}$, $\frac{M}{8}$ and $\frac{M}{16}$. In this experiment, we evaluate the query answering time of queries in $\mathcal{Q}$ by considering sets $\mathcal{M} \subseteq \mathcal{P}$ output of Problem *SS* presented in Section

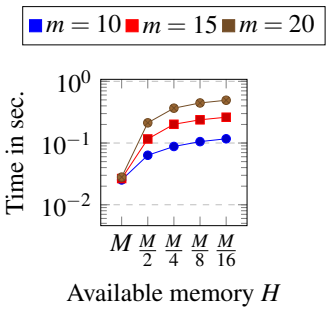


Figure 6.12: Query answering with restricted memory

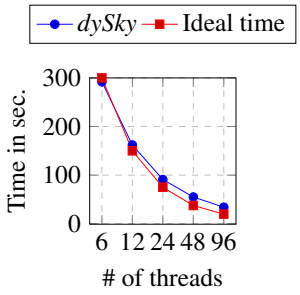


Figure 6.13: Parallel throughput

6.4.3 wrt values $M, \frac{M}{2}, \frac{M}{4}, \frac{M}{8}$, and $\frac{M}{16}$ for H. We consider datasets with $n = 10M$, $l = 1$, and m in $\{10, 15, 20\}$ as well as a workload Ω of 100 queries. Figure 6.12 depicts the results. We globally observe the same trend for all values of m. When H is equal to M, $\mathcal{P}$ is completely materialized and therefore queries in Ω are fully optimized. As long as we reduce the available memory H, the query answering time grows as now some sub-queries need to be computed from scratch.

Parallel processing throughput of dySky

In this experiment, we want to measure the multithreadring performance of *dySky*. We specifically consider Algorithm 18 *dySkySeq_build*. We run experiments with parameters $n = 10M$, $s = 6$, $l = 2$ and $m = 20$. We vary the number of parallel threads in $\{6, 12, 24, 48, 96\}$. Figure 6.13 depicts the results. We use the red curve just to show the linear trend. The results show that our algorithm is highly parallelizable because the sequential part is negligeable.

6.5.5 Concluding remarks

Globally, the performed experiments showed that our proposals outperform its competitors. Regarding query answering on the fly, we showed that in presence of challenging datasets, i.e., large and anti-correlated datasets, our algorithm *dySky_md* performs better than *CPS* which, to our knowledge, is the state of the art algorithm. Regarding precomputation based technique, we showed that our proposed structure, compared to *OST*, (i) is built faster, (ii) stores less data (iii) and provides better query answering performance. Regarding queries caching solutions, we showed that with much less cached queries, our proposal achieves better query performance than the refinement technique *Ref*. Finally, and thanks to *dySky* design, we showed that it is highly parallelizable.

6.6 Conclusion

In this chapter, we presented *dySky*, an approach for optimizing skyline queries over data with both totally and statically ordered dimensions, and partially and dynamically ordered dimensions. Given a query q and its related preference on the attributes domain $q.\mathcal{R}$, *dySky* decomposes q into sub-queries q_i, each of which operates on a small part of the dataset. In a further step for optimization, we proposed the sub-queries results as a building block for materialization. In this context, we addressed both full and partial materialization driven by a workload. The empirical experimental results we provide, show the superiority of *dySky* against its competitors. As future work, we plan to investigate the incremental maintenance of the materialized sub-queries with data updates.

Conclusion and perspectives

In this dissertation, we studied the time and memory optimization of skyline queries evaluation. We specifically considered the cases where the underlying data has dynamic properties. In a first part, we addressed the incremental maintenance of the structure NSC in presence of both dynamic data and streaming data. In a second part, we addressed the optimization of skyline queries in presence of data with dynamic orders.

In Chapter 3, we addressed the incremental maintenance of NSC in presence of dynamic data, i.e., tuples are inserted/deleted at any time. We presented the challenges of updating NSC wrt both insertions and deletions on the efficiency of structure. Regarding insertions, we showed that NSC's state changes only if inserted tuples belong to the topmost. We moreover presented an incremental compression technique based on pairs inclusion. We empirically evaluated this technique and showed it provides an interesting memory/time ratio compared to approximate compression technique. Also, we showed that the maintenance time of NSC upon insertions is proportional to the number of inserted tuples, and is ,in the worst case, better than the rebuild from scratch. Regarding deletions, we showed as well that NSC's state changes only if deleted tuples belong to the topmost. We presented the challenge of identifying the pairs computed wrt the deleted tuples and we proposed to augment the pairs with counters representing the number of tuples associated to a pair. We showed empirically that on one hand this additional information does not increase dramatically the memory usage, and on the other hand, it allows a fast maintenance procedure. Indeed, we showed that in practice very few tuples involve a large maintenance of the structure when they get deleted.

In Chapter 4, we addressed the incremental maintenance of NSC in presence of streaming data. We considered answering skyline queries over a window of size ω. To deal with that setting, we proposed a framework composed of (i) a data buffer, (ii) a main dataset, and (iii) NSCt, a variant of NSC to handle timestamped data. We proposed and explained techniques for both managing new insertions and updates. We evaluated empirically our proposals against a baseline skyline algorithm and a materialization based technique. First, we showed that our proposal outperforms the baseline skyline algorithm in terms of number of processed queries during a batch interval. Second, we exhibited the light memory consumption and fast maintenance process compared to the materialization based technique. Finally, we proved experimentally that our proposal

answers continuously and efficiently the Top-K frequent skyline which is the motivation of this work.

In Chapter 5, we investigated the optimization of regret minimization queries through NSC. These queries have been proposed to overcome the limitation of skyline queries and Top-K queries. We experimentally studied the time performance of regret minimization queries algorithms when computed on top of small candidate sets rather than the whole dataset. We showed that Top-K frequent skyline results, computed through NSC, represents a good candidate set for regret minimization queries.

In Chapter 6, we addressed the optimization of skyline queries over data with partially and dynamically ordered attributes. We proposed an approach which (i) decomposes the issued query into sub-queries, (ii) processes each sub-query independently, and (iii) integrates the results. First we considered answering queries on the fly and highlighted the interesting theoretical properties of our approach. Then, we considered the materialization of sub-queries in order to optimize further issued queries. We first described the materialization structure of the sub-queries, and the approach of answering queries through the materialized sub-queries. Then, we introduced the problem of selecting a subset of sub-queries to materialize given (i) a workload and (ii) under space constraint. We proved the hardness of that problem and proposed an efficient algorithm based on Knapsack dynamic programming algorithm. We evaluated empirically our proposal wrt several aspects. We showed its high performance wrt query answering time compared to its direct competitors. Moreover, we exhibited the improvement provided by both sub-queries materialization and multiprocessing.

The problematics addressed in this dissertation along with our proposals provide some orientations for future work.

Integrating NSC to a DBMS In Chapter 3, we addressed the problem of incrementally maintaining the indexing structure NSC upon updates. We provided techniques and procedures that makes NSC incrementally maintainable upon updates. Hence, NSC becomes an interesting and reliable tool for database management systems. As a future step, we could study NSC's integration to e.g. PostgreSQL.

MSSD in a distributed environment In Chapter 4, we proposed MSSD, a system for handling streaming data and managing NSCt a variant of NSC. We studied MSSD theoretically and proposed algorithms for (i) handling arriving data, (ii) updating the structure NSCt and (iii) answering issued queries. Nowadays, there exists frameworks for efficiently handling streams and distributed computation upon a cluster of machines such

as Spark [82]. This framework requires programs to follow the MapReduce programming model [83]. Hence, we aim to redesign MSSD in MapReduce model.

Theoretical guarantees on the regret of Top-K frequent skyline queries In Chapter 5, we investigated the relationship between regret minimization queries and multidimensional skylines. Our performed experiments provided interesting insights on this relationship. We have shown that Top-K frequent skyline query computes sets with regret close to that computed by a dedicated regret minimization algorithm. Moreover, Top-K frequent skyline computation is optimized by NSC. As future work, we want to push further the experimentation and find theoretical guarantees on regret of sets computed with Top-K frequent skyline queries.

Query preference decomposition into clusters of orders In Chapter 6, we addressed the optimization of skyline queries over data with partially and dynamically ordered attributes. We proposed a solution that decomposes the input query q into a set of sub-queries. Each sub-query considers a singleton preference composed of some order (v_i, v_j) in the query preference $q.\mathcal{R}$. A sub-query computes a set of dominated tuples wrt the singleton preference (v_i, v_j). The final result of q is then those tuples not belonging to any sub-query result. To speed up the computation and as sub-queries are independent, we process them in parallel. Still, this approach is disadvantaged in presence of dense preferences. The higher the number of orders in a query preference, the higher the number of sub-queries to compute. This involves as well redundant computation as two tuples may be compared in several sub-queries. One way to avoid this counter-performance is by decomposing the query preference $q.\mathcal{R}$ to clusters of orders rather than single orders and to map each cluster to a sub-query. This decomposition is a priory not an easy task. For example, we might decompose the preference wrt the number of available processors. Or, we might decompose the preference such that redundant computation is minimized. Hence, the efficient clustering of orders remains an open question.

Computing negative results of a query In this thesis, our approaches for optimizing skyline queries were based on the idea that computing tuples not belonging to the skyline then infer those belonging to the skyline is better than computing the tuples belonging to the skyline directly. For the problematics we addressed in this dissertation, we have proven the efficiency of such approach. We believe this technique would be extended to more general queries than skyline and sharing with it the same principle. More precisely, those queries that can be expressed by the formula $\{t \in T \mid \nexists t' \in T', E(t,t')\}$ where T and T' are either relations or relational queries, and E is some binary relation. One may even

think about other settings like for example graph data sets or ontologies. Note that skyline is an instance of such queries: T and T' are the same relation, and E is the dominance relationship.

List of Publications

Journals

- Karim Alami, Nicolas Hanusse, Patrick Kamnang-Wanko, Sofian Maabout. The negative skycube. Information Systems, Elsevier, 2020, 88, pp.101443.
- Karim Alami, Sofian Maabout. A framework for multidimensional skyline queries over streaming data. Data & Knowledge Engineering, Elsevier, 2020, pp.101792.
- Karim Alami, Sofian Maabout. A Partitioning Approach for Skyline Queries in Presence of Partial and Dynamic Orders. (Under review)

Conferences

- Karim Alami, Sofian Maabout. Multidimensional Skylines over Streaming Data. Database Systems for Advanced Applications - 24th International Conference, DASFAA 2019, Chiang Mai, Thailand, April 22-25, 2019, Proceedings, Lecture Notes in Computer Science, pp.338-342, 2019.
- Karim Alami, Sofian Maabout. Computational aspects around preference queries. Proceedings of the VLDB 2019 PhD Workshop, August 26th, 2019. Los Angeles, California.
- Karim Alami, Sofian Maabout. Experimental study of regret minimization sets and multidimensional skylines. (Under review)

References

[1] Surajit Chaudhuri and Luis Gravano. “Evaluating Top-*k* Selection Queries”. In: *VLDB’99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. Ed. by Malcolm P. Atkinson et al. Morgan Kaufmann, 1999, pp. 397–410.

[2] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. “The Skyline Operator”. In: *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*. Ed. by Dimitrios Georgakopoulos and Alexander Buchmann. IEEE Computer Society, 2001, pp. 421–430.

[3] Donald Kossmann, Frank Ramsak, and Steffen Rost. “Shooting Stars in the Sky: An Online Algorithm for Skyline Queries”. In: *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*. Morgan Kaufmann, 2002, pp. 275–286.

[4] Dimitris Papadias et al. “An Optimal and Progressive Algorithm for Skyline Queries”. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. Ed. by Alon Y. Halevy, Zachary G. Ives, and AnHai Doan. ACM, 2003, pp. 467–478.

[5] Jan Chomicki et al. “Skyline with Presorting”. In: *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*. Ed. by Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman. IEEE Computer Society, 2003, pp. 717–719.

[6] Jan Chomicki et al. “Skyline with Presorting: Theory and Optimizations”. In: *Intelligent Information Processing and Web Mining, Proceedings of the International IIS: IIPWM’05 Conference held in Gdansk, Poland, June 13-16, 2005*. Ed. by Mieczyslaw A. Klopotek, Slawomir T. Wierzchon, and Krzysztof Trojanowski. Vol. 31. Advances in Soft Computing. Springer, 2005, pp. 595–604.

[7] Jongwuk Lee and Seung-won Hwang. “BSkyTree: scalable skyline computation using a balanced pivot selection”. In: *EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010,*

Proceedings. Ed. by Ioana Manolescu et al. Vol. 426. ACM International Conference Proceeding Series. ACM, 2010, pp. 195–206.

[8] Jongwuk Lee and Seung-won Hwang. "Scalable skyline computation using a balanced pivot selection technique". In: *Inf. Syst.* 39 (2014), pp. 1–21.

[9] Sean Chester et al. "Scalable parallelization of skyline computation for multi-core processors". In: *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. Ed. by Johannes Gehrke et al. IEEE Computer Society, 2015, pp. 1083–1094.

[10] Ping Wu et al. "DeltaSky: Optimal Maintenance of Skyline Deletions without Exclusive Dominance Region Generation". In: *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*. Ed. by Rada Chirkova et al. IEEE Computer Society, 2007, pp. 486–495.

[11] Yufei Tao and Dimitris Papadias. "Maintaining Sliding Window Skylines on Data Streams". In: *IEEE Trans. Knowl. Data Eng.* 18.2 (2006), pp. 377–391.

[12] Jongwuk Lee and Seung-won Hwang. "Toward efficient multidimensional subspace skyline computation". In: *VLDB J.* 23.1 (2014), pp. 129–145.

[13] Sofian Maabout et al. "Skycube Materialization Using the Topmost Skyline or Functional Dependencies". In: *ACM Trans. Database Syst.* 41.4 (2016), 25:1–25:40.

[14] Tian Xia et al. "Online subspace skyline query processing using the compressed skycube". In: *ACM Trans. Database Syst.* 37.2 (2012), 15:1–15:36.

[15] Kenneth S. Bøgh et al. "Hashcube: A Data Structure for Space- and Query-Efficient Skycube Compression". In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*. Ed. by Jianzhong Li et al. ACM, 2014, pp. 1767–1770.

[16] Nicolas Hanusse, Patrick Kamnang Wanko, and Sofian Maabout. "Computing and Summarizing the Negative Skycube". In: *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*. Ed. by Snehasis Mukhopadhyay et al. ACM, 2016, pp. 1733–1742.

[17] Danupon Nanongkai et al. "Regret-Minimizing Representative Databases". In: *Proc. VLDB Endow.* 3.1 (2010), pp. 1114–1124.

[18] Sean Chester et al. “Computing k-Regret Minimizing Sets”. In: *Proc. VLDB Endow.* 7.5 (2014), pp. 389–400.

[19] Shiming Zhang et al. “Efficient Skyline Evaluation over Partially Ordered Domains”. In: *Proc. VLDB Endow.* 3.1 (2010), pp. 1255–1266.

[20] Peter C. Fishburn. “Combinatorics and Partially Ordered Sets: Dimension Theory (William T. Trotter)”. In: *SIAM Review* 35.3 (1993), pp. 519–520.

[21] Korte Bernhard and J Vygen. “Combinatorial optimization: Theory and algorithms”. In: *Springer, Third Edition, 2005.* (2008).

[22] Yu-Ling Hsueh, Chia-Chun Lin, and Chia-Che Chang. “An Efficient Indexing Method for Skyline Computations with Partially Ordered Domains”. In: *IEEE Trans. Knowl. Data Eng.* 29.5 (2017), pp. 963–976.

[23] Karim Alami et al. “The negative skycube”. In: *Inf. Syst.* 88 (2020).

[24] Karim Alami and Sofian Maabout. “A framework for multidimensional skyline queries over streaming data”. In: *Data Knowl. Eng.* 127 (2020), p. 101792.

[25] Karim Alami and Sofian Maabout. “Multidimensional Skylines over Streaming Data”. In: *Database Systems for Advanced Applications - 24th International Conference, DASFAA 2019, Chiang Mai, Thailand, April 22-25, 2019, Proceedings, Part III, and DASFAA 2019 International Workshops: BDMS, BDQM, and GDMA, Chiang Mai, Thailand, April 22-25, 2019, Proceedings*. Ed. by Guoliang Li et al. Vol. 11448. Lecture Notes in Computer Science. Springer, 2019, pp. 338–342.

[26] William B. T. Mock. “Pareto Optimality”. In: *Encyclopedia of Global Justice*. Ed. by Deen K. Chatterjee. Dordrecht: Springer Netherlands, 2011, pp. 808–809. ISBN: 978-1-4020-9160-5.

[27] Franco P Preparata and Michael I Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[28] H. T. Kung, Fabrizio Luccio, and Franco P. Preparata. “On Finding the Maxima of a Set of Vectors”. In: *J. ACM* 22.4 (1975), pp. 469–476.

[29] Thi Thu Trang Ngo. *Evaluating and Optimizing Skyline Algorithms on PostgreSQL*. May 2019. DOI: 10.13140/RG.2.2.20177.76646.

[30] Antonin Guttman. “R-Trees: A Dynamic Index Structure for Spatial Searching”. In: *SIGMOD Rec.* 14.2 (June 1984), 47–57. ISSN: 0163-5808.

[31] Norbert Beckmann et al. “The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles”. In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*. Ed. by Hector Garcia-Molina and H. V. Jagadish. ACM Press, 1990, pp. 322–331.

[32] Parke Godfrey, Ryan Shipley, and Jarek Gryz. “Maximal Vector Computation in Large Data Sets”. In: *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. Ed. by Klemens Böhm et al. ACM, 2005, pp. 229–240.

[33] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. “Efficient sort-based skyline evaluation”. In: *ACM Trans. Database Syst.* 33.4 (2008), 31:1–31:49.

[34] Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. “Efficient Progressive Skyline Computation”. In: *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Ed. by Peter M. G. Apers et al. Morgan Kaufmann, 2001, pp. 301–310.

[35] Chee Yong Chan and Yannis E. Ioannidis. “Bitmap Index Design and Evaluation”. In: *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. Ed. by Laura M. Haas and Ashutosh Tiwary. ACM Press, 1998, pp. 355–366.

[36] Kenneth S. Bøgh, Sean Chester, and Ira Assent. “SkyAlign: a portable, work-efficient skyline algorithm for multicore and GPU architectures”. In: *VLDB J.* 25.6 (2016), pp. 817–841.

[37] Kasper Mullesgaard et al. “Efficient Skyline Computation in MapReduce”. In: *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*. Ed. by Sihem Amer-Yahia et al. OpenProceedings.org, 2014, pp. 37–48.

[38] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN: 0-201-53771-0. URL: http://webdam.inria.fr/Alice/.

[39] Lukasz Golab and M. Tamer Özsu. “Issues in data stream management”. In: *SIGMOD Record* 32.2 (2003), pp. 5–14.

[40] Michael D. Morse, Jignesh M. Patel, and William I. Grosky. “Efficient Continuous Skyline Computation”. In: *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*. Ed. by Ling Liu et al. IEEE Computer Society, 2006, p. 108.

[41] Michael D. Morse, Jignesh M. Patel, and William I. Grosky. “Efficient continuous skyline computation”. In: *Inf. Sci.* 177.17 (2007), pp. 3411–3437.

[42] Zhenhua Huang, Sheng-Li Sun, and Wei Wang. “Efficient mining of skyline objects in subspaces over data streams”. In: *Knowl. Inf. Syst.* 22.2 (2010), pp. 159–183.

[43] Alexander Tzanakas, Eleftherios Tiakas, and Yannis Manolopoulos. “Skyline Algorithms on Streams of Multidimensional Data”. In: *New Trends in Databases and Information Systems - ADBIS 2016 Short Papers and Workshops, BigDap, DCSA, DC, Prague, Czech Republic, August 28-31, 2016, Proceedings*. Ed. by Mirjana Ivanovic et al. Vol. 637. Communications in Computer and Information Science. Springer, 2016, pp. 63–71.

[44] Christos Kalyvas, Theodoros Tzouramanis, and Yannis Manolopoulos. “Processing skyline queries in temporal databases”. In: *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*. Ed. by Ahmed Seffah et al. ACM, 2017, pp. 893–899.

[45] Yidong Yuan et al. “Efficient Computation of the Skyline Cube”. In: *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. Ed. by Klemens Böhm et al. ACM, 2005, pp. 241–252.

[46] Jian Pei et al. “Catching the Best Views of Skyline: A Semantic Approach Based on Decisive Subspaces”. In: *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. Ed. by Klemens Böhm et al. ACM, 2005, pp. 253–264.

[47] Yufei Tao, Xiaokui Xiao, and Jian Pei. “SUBSKY: Efficient Computation of Skylines in Subspaces”. In: *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*. Ed. by Ling Liu et al. IEEE Computer Society, 2006, p. 65.

[48] Akrivi Vlachou et al. “SKYPEER: Efficient Subspace Skyline Computation over Distributed Data”. In: *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*. Ed. by Rada Chirkova et al. IEEE Computer Society, 2007, pp. 416–425.

[49] Jian Pei et al. “Towards multidimensional subspace skyline analysis”. In: *ACM Trans. Database Syst.* 31.4 (2006), pp. 1335–1381.

[50] Kaiqi Zhang et al. “RSkycube: Efficient Skycube Computation by Reusing Principle”. In: *Database Systems for Advanced Applications - 22nd International Conference, DASFAA 2017, Suzhou, China, March 27-30, 2017, Proceedings, Part II*. Ed. by K. Selçuk Candan et al. Vol. 10178. Lecture Notes in Computer Science. Springer, 2017, pp. 51–64.

[51] Kenneth S. Bøgh et al. “Template Skycube Algorithms for Heterogeneous Parallelism on Multicore and GPU Architectures”. In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Semih Salihoglu et al. ACM, 2017, pp. 447–462.

[52] Chedy Raïssi, Jian Pei, and Thomas Kister. “Computing Closed Skycubes”. In: *Proc. VLDB Endow.* 3.1 (2010), pp. 838–847.

[53] Chee Yong Chan, Pin-Kwang Eng, and Kian-Lee Tan. “Stratified Computation of Skylines with Partially-Ordered Domains”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. Ed. by Fatma Özcan. ACM, 2005, pp. 203–214.

[54] Dimitris Sacharidis, Stavros Papadopoulos, and Dimitris Papadias. “Topologically Sorted Skylines for Partially Ordered Domains”. In: *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*. Ed. by Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng. IEEE Computer Society, 2009, pp. 1072–1083.

[55] Yu-Ling Hsueh and Tristan Hascoet. “Caching Support for Skyline Query Processing with Partially Ordered Domains”. In: *IEEE Trans. Knowl. Data Eng.* 26.11 (2014), pp. 2649–2661.

[56] Raymond Chi-Wing Wong et al. “Online Skyline Analysis with Dynamic Preferences on Nominal Attributes”. In: *IEEE Trans. Knowl. Data Eng.* 21.1 (2009), pp. 35–49.

[57] Afroza Sultana and Chengkai Li. “Continuous Monitoring of Pareto Frontiers on Partially Ordered Attributes for Many Users”. In: *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*. Ed. by Michael H. Böhlen et al. OpenProceedings.org, 2018, pp. 85–96.

[58] Md Farhadur Rahman et al. “Efficient Computation of Subspace Skyline over Categorical Domains”. In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*. Ed. by Ee-Peng Lim et al. ACM, 2017, pp. 407–416.

[59] Dimitris Papadias et al. “Progressive skyline computation in database systems”. In: *ACM Trans. Database Syst.* 30.1 (2005), pp. 41–82.

[60] Chee Yong Chan et al. “On High Dimensional Skylines”. In: *Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Munich, Germany, March 26-31, 2006, Proceedings*. Ed. by Yannis E. Ioannidis et al. Vol. 3896. Lecture Notes in Computer Science. Springer, 2006, pp. 478–495.

[61] Xuemin Lin et al. “Selecting Stars: The k Most Representative Skyline Operator”. In: *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*. Ed. by Rada Chirkova et al. IEEE Computer Society, 2007, pp. 86–95.

[62] Yufei Tao et al. “Distance-Based Representative Skyline”. In: *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*. Ed. by Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng. IEEE Computer Society, 2009, pp. 892–903.

[63] Tian Xia, Donghui Zhang, and Yufei Tao. “On Skylining with Flexible Dominance Relation”. In: *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*. Ed. by Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen. IEEE Computer Society, 2008, pp. 1397–1399.

[64] Akrivi Vlachou and Michalis Vazirgiannis. “Ranking the sky: Discovering the importance of skyline points through subspace dominance relationships”. In: *Data Knowl. Eng.* 69.9 (2010), pp. 943–964.

[65] Evangelos Dellis and Bernhard Seeger. “Efficient Computation of Reverse Skyline Queries”. In: *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. Ed. by Christoph Koch et al. ACM, 2007, pp. 291–302.

[66] Jinfei Liu et al. “Finding Pareto Optimal Groups: Group-based Skyline”. In: *Proc. VLDB Endow.* 8.13 (2015), pp. 2086–2097.

[67] Wei Cao et al. “k-Regret Minimizing Set: Efficient Algorithms and Hardness”. In: *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy*. Ed. by Michael Benedikt and Giorgio Orsi. Vol. 68. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 11:1–11:19.

[68] Abolfazl Asudeh et al. “Efficient Computation of Regret-ratio Minimizing Set: A Compact Maxima Representative”. In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Semih Salihoglu et al. ACM, 2017, pp. 821–834.

[69] Min Xie et al. “Efficient k-Regret Query Algorithm with Restriction-free Bound for any Dimensionality”. In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 959–974.

[70] Peng Peng and Raymond Chi-Wing Wong. “Geometry approach for k-regret query”. In: *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*. Ed. by Isabel F. Cruz et al. IEEE Computer Society, 2014, pp. 772–783.

[71] Pankaj K. Agarwal et al. “Efficient Algorithms for k-Regret Minimizing Sets”. In: *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*. Ed. by Costas S. Iliopoulos et al. Vol. 75. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 7:1–7:23.

[72] Nirman Kumar and Stavros Sintos. “Faster Approximation Algorithm for the *k*-Regret Minimizing Set and Related Problems”. In: *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX 2018, New Orleans, LA, USA, January 7-8, 2018*. Ed. by Rasmus Pagh and Suresh Venkatasubramanian. SIAM, 2018, pp. 62–74.

[73] Taylor Kessler Faulkner, Will Brackenbury, and Ashwin Lall. “k-Regret Queries with Nonlinear Utilities”. In: *Proc. VLDB Endow.* 8.13 (2015), pp. 2098–2109.

[74] Xianhong Qiu and Jiping Zheng. “An Efficient Algorithm for Computing k-Average-Regret Minimizing Sets in Databases”. In: *Web Information Systems and Applications - 15th International Conference, WISA 2018, Taiyuan, China, September 14-15, 2018, Proceedings*. Ed. by Xiaofeng Meng et al. Vol. 11242. Lecture Notes in Computer Science. Springer, 2018, pp. 404–412.

[75] Abolfazl Asudeh et al. “RRR: Rank-Regret Representative”. In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by Peter A. Boncz et al. ACM, 2019, pp. 263–280.

[76] Sudong Han, Jiping Zheng, and Qi Dong. “Efficient Processing of k-regret Queries via Skyline Priority”. In: *Web Information Systems and Applications - 15th International Conference, WISA 2018, Taiyuan, China, September 14-15, 2018, Proceedings*. Ed. by Xiaofeng Meng et al. Vol. 11242. Lecture Notes in Computer Science. Springer, 2018, pp. 413–420.

[77] Sudong Han, Jiping Zheng, and Qi Dong. “Efficient Processing of k-regret Queries via Skyline Frequency”. In: *Web Information Systems and Applications - 15th International Conference, WISA 2018, Taiyuan, China, September 14-15, 2018, Proceedings*. Ed. by Xiaofeng Meng et al. Vol. 11242. Lecture Notes in Computer Science. Springer, 2018, pp. 434–441.

[78] V. Chvatal. “A Greedy Heuristic for the Set-Covering Problem”. In: *Mathematics of Operations Research* 4.3 (1979), pp. 233–235. ISSN: 0364765X.

[79] Xuemin Lin et al. “Stabbing the Sky: Efficient Skyline Computation over Sliding Windows”. In: *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*. Ed. by Karl Aberer, Michael J. Franklin, and Shojiro Nishio. IEEE Computer Society, 2005, pp. 502–513.

[80] Lukasz Golab and M. Tamer Özsu. *Data Stream Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[81] Harold Greenberg. “A dynamic programming solution to integer linear programs”. In: *Journal of Mathematical Analysis and Applications* 26.2 (1969), pp. 454–459.

[82] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud’10, Boston, MA, USA, June 22, 2010*. Ed. by Erich M. Nahum and Dongyan Xu. USENIX Association, 2010.

[83] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. Ed. by Eric A. Brewer and Peter Chen. USENIX Association, 2004, pp. 137–150.