A Fully Dynamic Algorithm for k-Regret Minimizing Sets

Yanhao Wang*, Yuchen Li†,¶, Raymond Chi-Wing Wong‡, Kian-Lee Tan§
*University of Helsinki †Singapore Management University ‡The Hong Kong University of Science and Technology
§National University of Singapore ¶Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies
*yanhao.wang@helsinki.fi †yuchenli@smu.edu.sg ‡raywong@cse.ust.hk §tankl@comp.nus.edu.sg

***Abstract*—Selecting a small set of representatives from a large database is important in many applications such as multi-criteria decision making, web search, and recommendation. The k-regret minimizing set (k-RMS) problem was recently proposed for representative tuple discovery. Specifically, for a large database P of tuples with multiple numerical attributes, the k-RMS problem returns a size-r subset Q of P such that, for any possible ranking function, the score of the top-ranked tuple in Q is not much worse than the score of the k^{th}-ranked tuple in P. Although the k-RMS problem has been extensively studied in the literature, existing methods are designed for the static setting and cannot maintain the result efficiently when the database is updated. To address this issue, we propose the first fully-dynamic algorithm for the k-RMS problem that can efficiently provide the up-to-date result w.r.t. any tuple insertion and deletion in the database with a provable guarantee. Experimental results on several real-world and synthetic datasets demonstrate that our algorithm runs up to four orders of magnitude faster than existing k-RMS algorithms while providing results of nearly equal quality.**

***Index Terms*—regret minimizing set; dynamic algorithm; set cover; top-k query; skyline**

I. Introduction

In many real-world applications, including multi-criteria decision making [1], web search [2], recommendation [3], [4], and data description [5], a crucial task is to find a succinct representative subset from a large database to meet the requirements of various users. For example, when a user queries for a hotel on a website (e.g., booking.com and expedia.com), she/he will receive thousands of available options as results. The website would like to display the best choices in the first few pages from which almost all users could find what they are most interested in. A common method is to rank all results using a utility function that denotes a user's preference on different attributes (e.g., *price*, *rating*, and *distance to destination* for hotels) and only present the top-k tuples with the highest scores according to this function to the user. However, due to the wide diversity of user preferences, it is infeasible to represent the preferences of all users by any single utility function. Therefore, to select a set of highly representative tuples, it is necessary to take into account all (possible) user preferences.

A well-established approach to finding such representatives from databases is the *skyline* operator [6] based on the concept of *domination*: a tuple p dominates a tuple q iff p is as good as q on all attributes and strictly better than q on at least one attribute. For a given database, a skyline query returns its *Pareto-optimal* subset which consists of all tuples that are not dominated by any tuple. It is guaranteed that any user can find her/his best choice from the skyline because the top-ranked result according to any monotone function must not be dominated. Unfortunately, although skyline queries are effective for representing low-dimensional databases, their result sizes cannot be controlled and increase rapidly as the dimensionality (i.e., number of attributes in a tuple) grows, particularly so for databases with anti-correlated attributes.

Recently, the k-regret minimizing set (k-RMS) problem [1], [7]–[13] was proposed to alleviate the deficiency of skyline queries. Specifically, given a database P of tuples with d numeric attributes, the k-RMS problem aims to find a subset $Q \subseteq P$ such that, for any possible utility function, the top-1 tuple in Q can approximate the top-k tuples in P within a small error. Here, the *maximum k-regret ratio* [8] ($\mathtt{mrr}_k$) is used to measure how well Q can represent P. For a utility function f, the k-regret ratio ($\mathtt{rr}_k$) of Q over P is defined to be 0 if the top-1 tuple in Q is among the top-k tuples in P w.r.t. f, or otherwise, to be one minus the ratio between the score of the top-1 tuple in Q and the score of the k^{th}-ranked tuple in P w.r.t. f. Then, the *maximum k-regret ratio* ($\mathtt{mrr}_k$) is defined by the maximum of $\mathtt{rr}_k$ over a class of (possibly infinite) utility functions. Given a positive integer r, a k-RMS on a database P returns a subset $Q \subseteq P$ of size r to minimize $\mathtt{mrr}_k$. As an illustrative example, the website could run a k-RMS on a database of all available hotels to pick a set of r candidates from which all users can find at least one close to her/his top-k choices.

The k-RMS problem has been extensively studied recently. Theoretically, it is NP-hard [8]–[10] on any database with $d \geq 3$. In general, we categorize existing k-RMS algorithms into three types. The first type is dynamic programming algorithms [8], [10], [11] for k-RMS on two-dimensional data. Although they can provide optimal solutions when $d = 2$, they are not suitable for higher dimensions due to the NP-hardness of k-RMS. The second type is the greedy heuristic [1], [7], [8], which always adds a tuple that maximally reduces $\mathtt{mrr}_k$ at each iteration. Although these algorithms can provide high-quality results empirically, they have no theoretical guarantee and suffer from low efficiency on high-dimensional data. The third type is to transform k-RMS into another problem

*This research was mostly done when Yanhao Wang worked at National University of Singapore (NUS).

such as ε-kernel [9], [10], [12], [13], discretized matrix min-max [11], hitting set [9], [12], and k-MEDOID clustering [14], and then to utilize existing solutions of the transformed problem for k-RMS computation. Although these algorithms are more efficient than greedy heuristics while having theoretical bounds, they are designed for the static setting and cannot process database updates efficiently. Typically, most of them precompute the skyline as an input to compute the result of k-RMS. Once a tuple insertion or deletion triggers any change in the skyline, they are unable to maintain the result without re-running from scratch. Hence, existing k-RMS algorithms become very inefficient in highly dynamic environments where tuples in the databases are frequently inserted and deleted. However, dynamic databases are very common in real-world scenarios, especially for online services. For example, in a hotel booking system, the *prices* and *availabilities* of rooms are frequently changed over time. As another example, in an IoT network, a large number of sensors may often connect or disconnect with the server. Moreover, sensors also update their statistics regularly. Therefore, it is essential to address the problem of maintaining an up-to-date result for k-RMS when the database is frequently updated.

In this paper, we propose the first fully-dynamic k-RMS algorithm that can efficiently maintain the result of k-RMS w.r.t. any tuple insertion and deletion in the database with both theoretical guarantee and good empirical performance. Our main contributions are summarized as follows.

- We formally define the notion of *maximum k-regret ratio* and the *k-regret minimizing set* (k-RMS) problem in a fully-dynamic setting. (Section II)
- We propose the first fully-dynamic algorithm called FD-RMS to maintain the k-RMS result over tuple insertions and deletions in a database. Our basic idea is to transform *fully-dynamic k-RMS* into a *dynamic set cover* problem. Specifically, FD-RMS computes the (approximate) top-k tuples for a set of randomly sampled utility functions and builds a set system based on the top-k results. Then, the k-RMS result can be retrieved from an approximate solution for *set cover* on the set system. Furthermore, we devise a novel algorithm for dynamic set cover by introducing the notion of *stable solution*, which is used to efficiently update the k-RMS result whenever an insertion or deletion triggers some changes in top-k results as well as the set system. We also provide detailed theoretical analyses of FD-RMS. (Section III)
- We conduct extensive experiments on several real-world and synthetic datasets to evaluate the performance of FD-RMS. The results show that FD-RMS achieves up to four orders of magnitude speedup over existing k-RMS algorithms while providing results of nearly equal quality in a fully dynamic setting. (Section IV)

II. PRELIMINARIES

In this section, we formally define the problem we study in this paper. We first introduce the notion of maximum k-regret ratio. Then, we formulate the k-regret minimizing set (k-RMS) problem in a fully dynamic setting. Finally, we present the challenges of solving fully-dynamic k-RMS.

Tuple	x	y
p_1	0.2	1
p_2	0.6	0.8
p_3	0.7	0.5
p_4	1	0.1
p_5	0.4	0.3
p_6	0.2	0.7
p_7	0.3	0.9
p_8	0.6	0.6

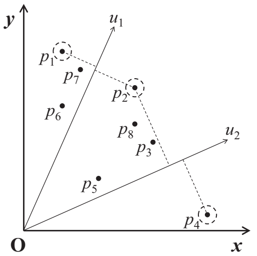


Fig. 1. A two-dimensional database of 8 tuples.

A. Maximum K-Regret Ratio

Let us consider a database P where each tuple $p \in P$ has d nonnegative numerical attributes $p[1], \ldots, p[d]$ and is represented as a point in the nonnegative orthant $\mathbb{R}_+^d$. A user's preference is denoted by a utility function $f : \mathbb{R}_+^d \rightarrow \mathbb{R}^+$ that assigns a positive score $f(p)$ to each tuple p. Following [1], [7], [8], [11], [13], we restrict the class of utility functions to *linear functions*. A function f is linear if and only if there exists a d-dimensional vector $u = (u[1], \ldots, u[d]) \in \mathbb{R}_+^d$ such that $f(p) = \langle u, p \rangle = \sum_{i=1}^{d} u[i] \cdot p[i]$ for any $p \in \mathbb{R}_+^d$. W.l.o.g., we assume the range of values on each dimension is scaled to $[0, 1]$ and any utility vector is normalized to be a unit[1], i.e., $\|u\| = 1$. Intuitively, the class of linear functions corresponds to the nonnegative orthant of d-dimensional unit sphere $\mathbb{U} = \{u \in \mathbb{R}_+^d \, : \, \|u\| = 1\}$.

We use $\varphi_j(u, P)$ to denote the tuple $p \in P$ with the j^{th}-largest score w.r.t. vector u and $\omega_j(u, P)$ to denote its score. Note that multiple tuples may have the same score w.r.t. u and any consistent rule can be adopted to break ties. For brevity, we drop the subscript j from the above notations when $j = 1$, i.e., $\varphi(u, P) = \arg\max_{p \in P} \langle u, p \rangle$ and $\omega(u, P) = \max_{p \in P} \langle u, p \rangle$. The top-$k$ tuples in P w.r.t. u is represented as $\Phi_k(u, P) = \{\varphi_j(u, P) \, : \, 1 \leq j \leq k\}$. Given a real number $\varepsilon \in (0, 1)$, the ε-approximate top-k tuples in P w.r.t. u is denoted as $\Phi_{k,\varepsilon}(u, P) = \{p \in P \, : \, \langle u, p \rangle \geq (1 - \varepsilon) \cdot \omega_k(u, P)\}$, i.e., the set of tuples whose scores are at least $(1 - \varepsilon) \cdot \omega_k(u, P)$.

For a subset $Q \subseteq P$ and an integer $k \geq 1$, we define the *k-regret ratio* of Q over P for a utility vector u by $\mathtt{rr}_k(u, Q) = \max\left(0, 1 - \frac{\omega(u,Q)}{\omega_k(u,P)}\right)$, i.e., the relative loss of replacing the k^{th}-ranked tuple in P by the top-ranked tuple in Q. Since it is required to consider the preferences of all possible users, our goal is to find a subset whose k-regret ratio is small for an arbitrary utility vector. Therefore, we define the *maximum k-regret ratio* of Q over P by $\mathtt{mrr}_k(Q) = \max_{u \in \mathbb{U}} \mathtt{rr}_k(u, Q)$. Intuitively, $\mathtt{mrr}_k(Q)$ measures how well the top-ranked tuple of Q approximates the k^{th}-ranked tuple of P in the worst case. For a real number $\varepsilon \in (0, 1)$, Q is said to be a (k, ε)-regret set of P iff $\mathtt{mrr}_k(Q) \leq \varepsilon$, or equivalently, $\varphi(u, Q) \in \Phi_{k,\varepsilon}(u, P)$ for any $u \in \mathbb{U}$. By definition, it holds that $\mathtt{mrr}_k(Q) \in [0, 1]$.

[1] The normalization does not affect our results because the maximum k-regret ratio is *scale-invariant* [1].

Example 1. Fig. 1 illustrates a database P in $\mathbb{R}^2_+$ with 8 tuples $\{p_1, \ldots, p_8\}$. For utility vectors $u_1 = (0.42, 0.91)$ and $u_2 = (0.91, 0.42)$, their top-2 results are $\Phi_2(u_1, P) = \{p_1, p_2\}$ and $\Phi_2(u_2, P) = \{p_2, p_4\}$, respectively. Given a subset $Q_1 = \{p_3, p_4\}$ of P, $\mathtt{rr}_2(u_1, Q_1) = 1 - \frac{0.749}{0.98} \approx 0.236$ as $\omega(u_1, Q_1) = \langle u_1, p_3 \rangle = 0.749$ and $\omega_2(u_1, P) = \langle u_1, p_2 \rangle = 0.98$. Furthermore, $\mathtt{mrr}_2(Q_1) \approx 0.444$ because $\mathtt{rr}_2(u, Q_1)$ is the maximum when $u = (0.0, 1.0)$ with $\mathtt{rr}_2(u, Q_1) = 1 - \frac{5}{9} \approx 0.444$. Finally, $Q_2 = \{p_1, p_2, p_4\}$ is a $(2, 0)$-regret set of P since $\mathtt{mrr}_2(Q_2) = 0$.

B. K-Regret Minimizing Set

Based on the notion of *maximum k-regret ratio*, we can formally define the *k-regret minimizing set* (k-RMS) problem in the following.

Definition 1 (k-Regret Minimizing Set)**.** Given a database $P \subset \mathbb{R}^d_+$ and a size constraint $r \in \mathbb{Z}^+$ ($r \geq d$), the k-regret minimizing set (k-RMS) problem returns a subset $Q^* \subseteq P$ of at most r tuples with the smallest *maximum k-regret ratio*, i.e., $Q^* = \arg\min_{Q \subseteq P \,:\, |Q| \leq r} \mathtt{mrr}_k(Q)$.

For any given k and r, we denote the k-RMS problem by $\mathtt{RMS}(k, r)$ and the *maximum k-regret ratio* of the optimal result Q^* for $\mathtt{RMS}(k, r)$ by $\varepsilon^*_{k,r}$. In particular, the r-regret query studied in [1], [7], [11], [13] is a special case of our k-RMS problem when $k = 1$, i.e., 1-RMS.

Example 2. Let us continue with the example in Fig. 1. For a query $\mathtt{RMS}(2, 2)$ on P, we have $Q^* = \{p_1, p_4\}$ with $\varepsilon^*_{2,2} = \mathtt{mrr}_2(Q^*) \approx 0.05$ because $\{p_1, p_4\}$ has the smallest maximum 2-regret ratio among all size-2 subsets of P.

In this paper, we focus on the fully-dynamic k-RMS problem. We consider an initial database P_0 and a (possibly countably infinite) sequence of operations $\Delta = \langle \Delta_1, \Delta_2, \ldots \rangle$. At each timestamp t ($t \in \mathbb{Z}^+$), the database is updated from P_{t-1} to P_t by performing an operation Δ_t of one of the following two types:

- **Tuple insertion** $\Delta_t = \langle p, + \rangle$: add a new tuple p to P_{t-1}, i.e., $P_t \leftarrow P_{t-1} \cup \{p\}$;
- **Tuple deletion** $\Delta_t = \langle p, - \rangle$: delete an existing tuple p from P_{t-1}, i.e., $P_t \leftarrow P_{t-1} \setminus \{p\}$.

Note that the update of a tuple can be processed by a deletion followed by an insertion, and thus is not discussed separately in this paper. Given an initial database P_0, a sequence of operations Δ, and a query $\mathtt{RMS}(k, r)$, we aim to keep track of the result Q^*_t for $\mathtt{RMS}(k, r)$ on P_t at any time t.

Fully-dynamic k-RMS faces two challenges. First, the k-RMS problem is NP-hard [8]–[10] for any $d \geq 3$. Thus, the optimal solution of k-RMS is intractable for any database with three or more attributes unless P=NP in both static and dynamic settings. Hence, we will focus on maintaining an approximate result of k-RMS in this paper. Second, existing k-RMS algorithms can only work in the static setting. They must recompute the result from scratch once an operation triggers any update in the skyline (Note that since the result of k-RMS is a subset of the skyline [1], [8], it remains unchanged for any operation on non-skyline tuples). However, frequent recomputation leads to significant overhead and causes low efficiency on highly dynamic databases. Therefore, we will propose a novel method for fully-dynamic k-RMS that can maintain a high-quality result for $\mathtt{RMS}(k, r)$ on a database w.r.t. any tuple insertion and deletion efficiently.

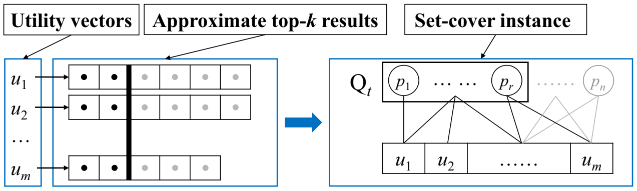


Fig. 2. An illustration of FD-RMS

III. The FD-RMS Algorithm

In this section, we present our FD-RMS algorithm for k-RMS in a fully dynamic setting. The general framework of FD-RMS is illustrated in Fig. 2. The basic idea is to transform *fully-dynamic k-RMS* to a *dynamic set cover* problem. Let us consider how to compute the result of $\mathtt{RMS}(k, r)$ on database P_t. First of all, we draw a set of m random utility vectors $\{u_1, \ldots, u_m\}$ from $\mathbb{U}$ and maintain the ε-approximate top-k result of each u_i ($i \in [1, m]$) on P_t, i.e., $\Phi_{k,\varepsilon}(u_i, P_t)$. Note that ε should be given as an input parameter of FD-RMS and we will discuss how to specify its value at the end of Section III. Then, we construct a set system $\Sigma = (\mathcal{U}, \mathcal{S})$ based on the approximate top-k results, where the universe $\mathcal{U} = \{u_1, \ldots, u_m\}$ and the collection $\mathcal{S}$ consists of n_t sets ($n_t = |P_t|$) each of which corresponds to one tuple in P_t. Specifically, for each tuple $p \in P_t$, we define $S(p)$ as a set of utility vectors for which p is an ε-approximate top-k result on P_t. Or formally, $S(p) = \{u \in \mathcal{U} : p \in \Phi_{k,\varepsilon}(u, P_t)\}$ and $\mathcal{S} = \{S(p) : p \in P_t\}$. After that, we compute a result Q_t for $\mathtt{RMS}(k, r)$ on P_t using an (approximate) solution for set cover on Σ. Let $\mathcal{C} \subseteq \mathcal{S}$ be a set-cover solution of Σ, i.e., $\bigcup_{S(p) \in \mathcal{C}} S(p) = \mathcal{U}$. We use the set Q_t of tuples corresponding to $\mathcal{C}$, i.e., $Q_t = \{p \in P_t : S(p) \in \mathcal{C}\}$, as the result of $\mathtt{RMS}(k, r)$ on P_t. Given the above framework, there are still two challenges of updating the result of k-RMS in a fully dynamic setting. Firstly, because the size of Q_t is restricted to r, it is necessary to always keep an appropriate value of m over time so that $|\mathcal{C}| \leq r$. Secondly, the updates in approximate top-k results triggered by tuple insertions and deletions in the database lead to the changes in the set collection $\mathcal{S}$. Therefore, it is essential to maintain the set-cover solution $\mathcal{C}$ over time for the changes in $\mathcal{S}$. In fact, both challenges can be treated as a *dynamic set cover* problem that keeps a set-cover solution w.r.t. changes in both $\mathcal{U}$ and $\mathcal{S}$. Therefore, we will first introduce the background on *dynamic set cover* in Section III-A. After that, we will elaborate on how FD-RMS processes k-RMS in a fully dynamic setting using the *dynamic set cover* algorithm in Section III-B.

A. Background: Dynamic Set Cover

Given a set system $\Sigma = (\mathcal{U}, \mathcal{S})$, the *set cover* problem asks for the smallest subset $\mathcal{C}^*$ of $\mathcal{S}$ whose union equals to the universe $\mathcal{U}$. It is one of Karp's 21 NP-complete problems [15], and cannot be approximated to $(1 - o(1)) \cdot \ln m$ $(m = |\mathcal{U}|)$ unless P=NP [16]. A common method to find an approximate set-cover solution is the greedy algorithm. Starting from $\mathcal{C} = \varnothing$, it always adds the set that contains the largest number of uncovered elements in $\mathcal{U}$ to $\mathcal{C}$ at each iteration until $\bigcup_{S \in \mathcal{C}} S = \mathcal{U}$. Theoretically, the solution $\mathcal{C}$ achieves an approximation ratio of $(1 + \ln m)$, i.e., $|\mathcal{C}| \leq (1 + \ln m) \cdot |\mathcal{C}^*|$. But obviously, the greedy algorithm cannot dynamically update the set-cover solution when the set system Σ is changed.

Recently, there are some theoretical advances on covering and relevant problems (e.g., vertex cover, maximum matching, set cover, and maximal independent set) in dynamic settings [17]–[20]. Although these theoretical results have opened up new ways to design dynamic set cover algorithms, they cannot be directly applied to the update procedure of FD-RMS because of two limitations. First, existing dynamic algorithms for set cover [18], [19] can only handle the update in the universe $\mathcal{U}$ but assume that the set collection $\mathcal{S}$ is not changed. But in our scenario, the changes in top-k results lead to the update of $\mathcal{S}$. Second, due to the extremely large constants introduced in their analyses, the solutions returned may be far away from the optima in practice.

Therefore, we devise a more practical approach to dynamic set cover that supports any update in both $\mathcal{U}$ and $\mathcal{S}$. Our basic idea is to introduce the notion of *stability* to a set-cover solution. Then, we prove that any stable solution is $O(\log m)$-approximate $(m = |\mathcal{U}|)$ for set cover. Based on this result, we are able to design an algorithm to maintain a set-cover solution w.r.t. any change in Σ by guaranteeing its *stability*.

We first formalize the concept of *stability* of a set-cover solution. Let $\mathcal{C} \subseteq \mathcal{S}$ be a set-cover solution on $\Sigma = (\mathcal{U}, \mathcal{S})$. We define an assignment ϕ from each element $u \in \mathcal{U}$ to a *unique* set $S \in \mathcal{C}$ that contains u (or formally, $\phi : \mathcal{U} \rightarrow \mathcal{C}$). For each set $S \in \mathcal{C}$, its cover set $\mathtt{cov}(S)$ is defined as the set of elements assigned to S, i.e., $\mathtt{cov}(S) = \{u \in \mathcal{U} : \phi(u) = S\}$. By definition, the cover sets of different sets in $\mathcal{C}$ are *mutually disjoint* from each other. Then, we can organize the sets in $\mathcal{C}$ into hierarchies according to the numbers of elements covered by them. Specifically, we put a set $S \in \mathcal{C}$ in a higher level if it covers more elements and vice versa. We associate each level $\mathcal{L}_j$ $(j \in \mathbb{N})$ with a range of cover number[2] $[2^j, 2^{j+1})$. Each set $S \in \mathcal{C}$ is assigned to a level $\mathcal{L}_j$ if $2^j \leq |\mathtt{cov}(S)| < 2^{j+1}$. We use A_j to denote the set of elements assigned to any set in $\mathcal{L}_j$, i.e., $A_j = \{u \in \mathcal{U} : \phi(u) \in \mathcal{L}_j\}$. Moreover, the notations $\mathcal{L}$ with subscripts, i.e., $\mathcal{L}_{>j}$ or $\mathcal{L}_{\geq j}$ and $\mathcal{L}_{<j}$ or $\mathcal{L}_{\leq j}$, represent the sets in all the levels above (excl. or incl.) and below $\mathcal{L}_j$ (excl. or incl.), respectively. The same subscripts are also used for A. Based on the above notions, we formally define the *stability* of a set-cover solution in Definition 2 and give its approximation ratio in Theorem 1.

Definition 2 (Stable Set-Cover Solution)**.** A solution $\mathcal{C}$ for *set cover* on $\Sigma = (\mathcal{U}, \mathcal{S})$ is stable if:

1) For each set $S \in \mathcal{L}_j$, $2^j \leq |\mathtt{cov}(S)| < 2^{j+1}$;
2) For each level $\mathcal{L}_j$, there is no $S \in \mathcal{S}$ s.t. $|S \cap A_j| \geq 2^{j+1}$.

Theorem 1. *If a set-cover solution $\mathcal{C}$ is stable, then it satisfies that $|\mathcal{C}| \leq O(\log m) \cdot |\mathcal{C}^*|$.*

Proof. Let $\mathtt{OPT} = |\mathcal{C}^*|$, $\rho^* = \frac{m}{\mathtt{OPT}}$, and j^* be the level index such that $2^{j^*} \leq \rho^* < 2^{j^*+1}$. According to Condition (1) of Definition 2, we have $|\mathtt{cov}(S)| \geq 2^{j^*}$ for any $S \in \mathcal{L}_{\geq j^*}$. Thus, it holds that $|\mathcal{L}_{\geq j^*}| \leq \frac{|A_{\geq j^*}|}{2^{j^*}} \leq \frac{m}{2^{j^*}} \leq \frac{\rho^*}{2^{j^*}} \cdot \mathtt{OPT} \leq 2 \cdot \mathtt{OPT}$. For some level $\mathcal{L}_j$ with $j < j^*$, according to Condition (2) of Definition 2, any $S \in \mathcal{S}$ covers at most 2^{j+1} elements in A_j. Hence, $\mathcal{S}^*$ needs at least $\frac{|A_j|}{2^{j+1}}$ sets to cover A_j, i.e., $\mathtt{OPT} \geq \frac{|A_j|}{2^{j+1}}$. Since $|\mathtt{cov}(S)| \geq 2^j$ for each $S \in \mathcal{L}_j$, it holds that $|\mathcal{L}_j| \leq \frac{|A_j|}{2^j} \leq 2 \cdot \mathtt{OPT}$. As $1 \leq |\mathtt{cov}(S)| \leq m$, the range of level index is $[0, \log_2 m]$. Thus, the number of levels below $\mathcal{L}_{j^*}$ is at most $\log_2 m$. To sum up, we prove that

$$|\mathcal{C}| = |\mathcal{L}_{\geq j^*}| + |\mathcal{L}_{<j^*}| \leq (2 + 2\log_2 m) \cdot \mathtt{OPT}$$

and conclude the proof. □

We then describe our method for *dynamic set cover* in Algorithm 1. First of all, we use GREEDY to initialize a set-cover solution $\mathcal{C}$ on Σ (Line 1). As shown in Lines 13–19, GREEDY follows the classic greedy algorithm for set cover, and the only difference is that all the sets in $\mathcal{C}$ are assigned to different levels according to the sizes of their cover sets. Then, the procedure of updating $\mathcal{C}$ for set operation σ is shown in Lines 2–12. Our method supports four types of set operations to update Σ as follows: $\sigma = (u, S, \pm)$, i.e., to add/remove an element u to/from a set $S \in \mathcal{S}$; $\sigma = (u, \mathcal{U}, \pm)$, i.e., to add/remove an element u to/from the universe $\mathcal{U}$. We identify three cases in which the assignment of u must be changed for σ. When $\sigma = (u, S, -)$ and $\phi(u) = S$, it will reassign u to another set containing u; For $\sigma = (u, \mathcal{U}, \pm)$, it will add or delete the assignment of u accordingly. After that, for each set with some change in its cover set, it calls RELEVEL (e.g., Lines 5, 8, and 11) to check whether the set should be moved to a new level based on the updated size of its cover set. The detailed procedure of RELEVEL is given in Lines 20–27. Finally, STABILIZE (Line 12) is always called for every σ to guarantee the stability of $\mathcal{C}$ since $\mathcal{C}$ may become unstable due to the changes in Σ and $\phi(u)$. The procedure of stabilization is presented in Lines 28–32. It finds all sets that violate Condition (2) of Definition 2 and adjust $\mathcal{C}$ for these sets until no set should be adjusted anymore.

Theoretical Analysis: Next, we will analyze Algorithm 1 theoretically. We first show that a set-cover solution returned by GREEDY is stable. Then, we prove that STABILIZE converges to a stable solution in finite steps.

Lemma 1. *The solution $\mathcal{C}$ returned by* GREEDY *is stable.*

Proof. First of all, it is obvious that each set $S \in \mathcal{C}$ is assigned to the correct level according to the size of its cover set and

[2] Here, the base 2 may be replaced by any constant greater than 1.

Algorithm 1: DYNAMIC SET COVER

```
   Input  : Set system $\Sigma$, set operation $\sigma$
   Output : Stable set-cover solution $\mathcal{C}$
   /* compute an initial solution $\mathcal{C}$ on $\Sigma$ */
 1 $\mathcal{C} \leftarrow$ GREEDY($\Sigma$);
   /* update $\mathcal{C}$ for $\sigma = (u, S, \pm)$ or $(u, \mathcal{U}, \pm)$ */
 2 if $\sigma = (u, S, -)$ and $u \in \texttt{cov}(S)$ then
 3     $\texttt{cov}(S) \leftarrow \texttt{cov}(S) \setminus \{u\}$;
 4     $\texttt{cov}(S^+) \leftarrow \texttt{cov}(S^+) \cup \{u\}$ for $S^+ \in \mathcal{S}$ s.t. $u \in S^+$;
 5     RELEVEL($S$) and RELEVEL($S^+$);
 6 else if $\sigma = (u, \mathcal{U}, +)$ then
 7     $\texttt{cov}(S^+) \leftarrow \texttt{cov}(S^+) \cup \{u\}$ for $S^+ \in \mathcal{S}$ s.t. $u \in S^+$;
 8     RELEVEL($S^+$);
 9 else if $\sigma = (u, \mathcal{U}, -)$ then
10     $\texttt{cov}(S^-) \leftarrow \texttt{cov}(S^-) \setminus \{u\}$ if $u \in \texttt{cov}(S^-)$;
11     RELEVEL($S^-$);
12 STABILIZE($\mathcal{C}$);
13 Function GREEDY($\Sigma$)
14     $I \leftarrow \mathcal{U}$, $\mathcal{L}_j \leftarrow \varnothing$ for every $j \geq 0$;
15     while $I \neq \varnothing$ do
16         $S^* \leftarrow \arg\max_{S \in \mathcal{S}} |I \cap S|$, $\texttt{cov}(S^*) \leftarrow I \cap S^*$;
17         Add $S^*$ to $\mathcal{L}_j$ s.t. $2^j \leq |\texttt{cov}(S^*)| < 2^{j+1}$;
18         $I \leftarrow I \setminus \texttt{cov}(S^*)$;
19     return $\mathcal{C} \leftarrow \bigcup_{j \geq 0} \mathcal{L}_j$;
20 Function RELEVEL($S$)
21     if $\texttt{cov}(S) = \varnothing$ then
22         $\mathcal{C} \leftarrow \mathcal{C} \setminus \{S\}$;
23     else
24         Let $\mathcal{L}_j$ be the current level of $S$;
25         if $|\texttt{cov}(S)| < 2^j$ or $|\texttt{cov}(S)| \geq 2^{j+1}$ then
26             Let $j'$ be the index s.t. $2^{j'} \leq |\texttt{cov}(S)| < 2^{j'+1}$;
27             Move $S$ from $\mathcal{L}_j$ to $\mathcal{L}_{j'}$;
28 Function STABILIZE($\mathcal{C}$)
29     while $\exists S \in \mathcal{S}$ and $\mathcal{L}_j$ s.t. $|S \cap A_j| \geq 2^{j+1}$ do
30         $\texttt{cov}(S) \leftarrow \texttt{cov}(S) \cup (S \cap A_j)$, RELEVEL($S$);
31         while $\exists S' \in \mathcal{C} : \texttt{cov}(S) \cap \texttt{cov}(S') \neq \varnothing$ do
32             $\texttt{cov}(S') \leftarrow \texttt{cov}(S') \setminus \texttt{cov}(S)$, RELEVEL($S'$);
```

Condition (1) of Definition 2 is satisfied. Then, we sort the sets in $\mathcal{C}$ as $S_1^*, \ldots, S_{|\mathcal{C}|}^*$ by the order in which they are added. Let S_i^* be the set s.t. $|\texttt{cov}(S_i^*)| < 2^{j+1}$ and $|\texttt{cov}(S_{i'}^*)| \geq 2^{j+1}$ for any $i' < i$, i.e., S_i^* is the first set added to level $\mathcal{L}_j$. We have $|I \cap S_i^*| = |\texttt{cov}(S_i^*)| < 2^{j+1}$ where I is the set of uncovered elements before S_i^* is added to $\mathcal{C}$. If there were a set $S \in \mathcal{S}$ such that $|S \cap A_j| > 2^{j+1}$, we would acquire $|I \cap S| \geq |S \cap A_j| > 2^{j+1}$ and $|I \cap S| > |I \cap S_i^*|$, which contradicts with Line 16 of Algorithm 2. Thus, $\mathcal{C}$ must satisfy Condition (2) of Definition 2. To sum up, $\mathcal{C}$ is a stable solution. □

Lemma 2. *The procedure* STABILIZE *converges to a stable solution in $O(m \log m)$ steps.*

Proof. For an iteration of the `while` loop (i.e., Lines 28–32) that picks a set S and a level $\mathcal{L}_j$, the new level $\mathcal{L}_{j'}$ of S always satisfies $j' > j$. Accordingly, all the elements in $\texttt{cov}(S)$ are moved from $A_{\leq j}$ to $A_{j'}$. At the same time, no element in $A_{\geq j'}$ is moved to lower levels. Since each level contains at most m elements ($|A_j| \leq m$), STABILIZE moves at most m elements across $O(\log m)$ levels. Therefore, it must terminate in $O(m \log m)$ steps. Furthermore, after termination, the set-cover solution $\mathcal{C}$ must satisfy both conditions in Definition 2. Thus, we conclude the proof. □

Algorithm 2: INITIALIZATION

```
   Input  : Query $\texttt{RMS}(k, r)$, initial database $P_0$, parameters
            $\varepsilon \in (0, 1)$ and $M \in \mathbb{Z}^+$ $(M > r)$
   Output : Result $Q_0$ of $\texttt{RMS}(k, r)$ on $P_0$
 1 Draw $M$ vectors $\{u_i \in \mathbb{U} : i \in [1, M]\}$ where the first $d$ are
   the standard basis of $\mathbb{R}^d_+$ and the remaining are uniformly
   sampled from $\mathbb{U}$;
 2 Compute $\Phi_{k,\varepsilon}(u_i, P_0)$ of every $u_i$ where $i \in [1, M]$;
 3 $L \leftarrow r$, $H \leftarrow M$, $m \leftarrow (L + H)/2$;
 4 while true do
 5     foreach $p \in P_0$ do
 6         $S(p) \leftarrow \{u_i : i \in [1, m] \wedge p \in \Phi_{k,\varepsilon}(u_i, P_0)\}$;
 7     $\Sigma = (\mathcal{U}, \mathcal{S})$ where $\mathcal{U} = \{u_i : i \in [1, m]\}$ and
       $\mathcal{S} = \{S(p) : p \in P_0\}$);
 8     $\mathcal{C} \leftarrow$ GREEDY($\Sigma$);
 9     if $|\mathcal{C}| < r$ then
10         $L \leftarrow m + 1$, $m \leftarrow (L + H)/2$;
11     else if $|\mathcal{C}| > r$ then
12         $H \leftarrow m - 1$, $m \leftarrow (L + H)/2$;
13     else if $|\mathcal{C}| = r$ or $m = M$ then
14         break;
15 return $Q_0 \leftarrow \{p \in P_0 : S(p) \in \mathcal{C}\}$;
```

The above two lemmas can guarantee that the set-cover solution provided by Algorithm 1 is always stable after any change in the set system. In the next subsection, we will present how to use it for fully-dynamic k-RMS.

B. Algorithmic Description

Next, we will present how FD-RMS maintains the k-RMS result by always keeping a *stable* set-cover solution on a dynamic set system built from the approximate top-k results over tuple insertions and deletions.

Initialization: We first present how FD-RMS computes an initial result Q_0 for $\texttt{RMS}(k, r)$ on P_0 from scratch in Algorithm 2. There are two parameters in FD-RMS: the approximation factor of top-k results ε and the upper bound of sample size M. The lower bound of sample size is set to r because we can always find a set-cover solution of size equal to the size of the universe (i.e., m in FD-RMS). First of all, it draws M utility vectors $\{u_1, \ldots, u_M\}$, where the first d vectors are the standard basis of $\mathbb{R}^d_+$ and the remaining are uniformly sampled from $\mathbb{U}$, and computes the ε-approximate top-k result of each vector. Subsequently, it finds an appropriate $m \in [r, M]$ so that the size of the set-cover solution on the set system Σ built on $\mathcal{U} = \{u_1, \ldots, u_m\}$ is exactly r. The detailed procedure is as presented in Lines 3–14. Specifically, it performs a binary search on range $[r, M]$ to determine the value of m. For a given m, it first constructs a set system Σ according to Lines 5–7. Next, it runs GREEDY in Algorithm 1 to compute a set-cover solution $\mathcal{C}$ on Σ. After

Algorithm 3: UPDATE

```
Input  : Query RMS($k, r$), database $P_{t-1}$, operation $\Delta_t$,
         set-cover solution $\mathcal{C}$
Output : Result $Q_t$ for RMS($k, r$) on $P_t$
1  Update $P_{t-1}$ to $P_t$ w.r.t. $\Delta_t$;
2  for $i \leftarrow 1, \ldots, M$ do
3      Update $\Phi_{k,\varepsilon}(u_i, P_{t-1})$ to $\Phi_{k,\varepsilon}(u_i, P_t)$ w.r.t. $\Delta_t$;
4  Maintain $\Sigma$ based on $\Phi_{k,\varepsilon}(u_i, P_t)$;
5  if $\Delta_t = \langle p, + \rangle$ then
6      foreach $u \in S(p)$ do
7          if $u \in \texttt{cov}(S(p'))$ and $u \notin S(p')$ then
8              Update $\mathcal{C}$ for $\sigma = (u, S(p'), -)$;
9  else if $\Delta_t = \langle p, - \rangle$ then
10     Delete $S(p)$ from $\mathcal{C}$ if $S(p) \in \mathcal{C}$;
11     foreach $u \in \texttt{cov}(S(p))$ do
12         Update $\mathcal{C}$ for $\sigma = (u, S(p), -)$;
13 if $|\mathcal{C}| \neq r$ then
14     $m, \mathcal{C} \leftarrow$ UPDATEM($\Sigma$);
15 return $Q_t \leftarrow \{p \in P_t \, : \, S(p) \in \mathcal{C}\}$;
```

Algorithm 4: UPDATEM(Σ)

```
Output : Updated sample size $m$ and solution $\mathcal{C}$ on $\Sigma$
if $|\mathcal{C}| < r$ then
    while $m < M$ and $|\mathcal{C}| < r$ do
        $m \leftarrow m + 1, \mathcal{U} \leftarrow \mathcal{U} \cup \{u_m\}$;
        foreach $p \in \Phi_{k,\varepsilon}(u_m, P_t)$ do
            $S(p) \leftarrow S(p) \cup \{u_m\}$;
        Update $\mathcal{C}$ for $\sigma = (u_m, \mathcal{U}, +)$;
else if $|\mathcal{C}| > r$ then
    while $|\mathcal{C}| > r$ do
        $\mathcal{U} \leftarrow \mathcal{U} \setminus \{u_m\}$;
        foreach $p \in \Phi_{k,\varepsilon}(u_m, P_t)$ do
            $S(p) \leftarrow S(p) \setminus \{u_m\}$;
        Update $\mathcal{C}$ for $\sigma = (u_m, \mathcal{U}, -)$;
        $m \leftarrow m - 1$;
return $m, \mathcal{C}$;
```

that, if $|\mathcal{C}| \neq r$ and $m < M$, it will refresh the value of m and rerun the above procedures; Otherwise, m is determined and the current set-cover solution $\mathcal{C}$ will be used to compute Q_0 for $\texttt{RMS}(k, r)$. Finally, it returns all the tuples whose corresponding sets are included in $\mathcal{C}$ as the result Q_0 for $\texttt{RMS}(k, r)$ on P_0 (Line 15).

Update: The procedure of updating the result of $\texttt{RMS}(k, r)$ w.r.t. Δ_t is shown in Algorithm 3. First, it updates the database from P_{t-1} to P_t and the approximate top-k result from $\Phi_{k,\varepsilon}(u_i, P_{t-1})$ to $\Phi_{k,\varepsilon}(u_i, P_t)$ for each u_i w.r.t. Δ_t (Lines 1–3). Then, it also maintains the set system Σ according to the changes in approximate top-k results (Line 4). Next, it updates the set-cover solution $\mathcal{C}$ for the changes in Σ as follows.

- **Insertion:** The procedure of updating $\mathcal{C}$ w.r.t. an insertion $\Delta_t = \langle p, + \rangle$ is presented in Lines 5–8. The changes in top-k results lead to two updates in Σ: (1) the insertion of $S(p)$ to $\mathcal{S}$ and (2) a series of deletions each of which represents a tuple p' is deleted from $\Phi_{k,\varepsilon}(u, P_t)$ due to the insertion of p. For each deletion, it needs to check whether u is previously assigned to $S(p')$. If so, it will update $\mathcal{C}$ by reassigning u to a new set according to Algorithm 1 because u has been deleted from $S(p')$.
- **Deletion:** The procedure of updating $\mathcal{C}$ w.r.t. a deletion $\Delta_t = \langle p, - \rangle$ is shown in Lines 9–12. In contrast to an insertion, the deletion of p leads to the removal of $S(p)$ from $\mathcal{S}$ and a series of insertions. Thus, it must delete $S(p)$ from $\mathcal{C}$. Next, it will reassign each $u \in \texttt{cov}(S(p))$ to a new set according to Algorithm 1.

Then, it checks whether the size of $\mathcal{C}$ is still r. If not, it will update the sample size m and the universe $\mathcal{U}$ so that the set-cover solution $\mathcal{C}$ consists of r sets. The procedure of updating m and $\mathcal{U}$ as well as maintaining $\mathcal{C}$ on the updated $\mathcal{U}$ is shown in Algorithm 4. When $|\mathcal{C}| < r$, it will add new utility vectors from u_{m+1}, and so on, to the universe and maintain $\mathcal{C}$ until $|\mathcal{C}| = r$ or $m = M$. On the contrary, if $|\mathcal{C}| > r$, it will drop existing utility vectors from u_m, and so on, from the universe and maintain $\mathcal{C}$ until $|\mathcal{C}| = r$. Finally, the updated m and $\mathcal{C}$ are returned. After all above procedures, it also returns Q_t corresponding to the set-cover solution $\mathcal{C}$ on the updated Σ as the result of $\texttt{RMS}(k, r)$ on P_t.

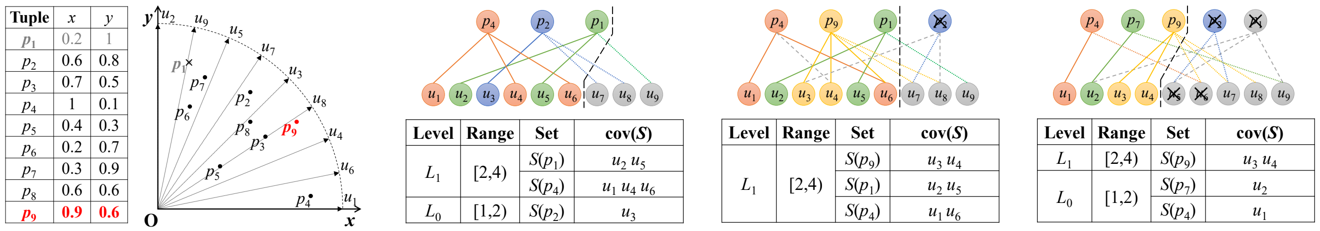
Example 3. Fig. 3 illustrates an example of using FD-RMS to process a k-RMS with $k = 1$ and $r = 3$. Here, we set $\varepsilon = 0.002$ and $M = 9$. In Fig. 3(b), we show how to compute Q_0 for $\texttt{RMS}(1, 3)$ on $P_0 = \{p_1, \ldots, p_8\}$. It first uses $m = (3 + 9)/2 = 6$ and runs GREEDY to get a set-cover solution $\mathcal{C} = \{S(p_1), S(p_2), S(p_4)\}$. Since $|\mathcal{C}| = 3$, it does not change m anymore and returns $Q_0 = \{p_1, p_2, p_4\}$ for $\texttt{RMS}(1, 3)$ on P_0. Then, the result of FD-RMS after the update procedures for $\Delta_1 = \langle p_9, + \rangle$ as Algorithm 3 is shown in Fig. 3(c). For $\texttt{RMS}(1, 3)$ on $P_1 = \{p_1, \ldots, p_9\}$, the result Q_1 is updated to $\{p_1, p_4, p_9\}$. Finally, after the update procedures for $\Delta_2 = \langle p_1, - \rangle$, as shown in Fig. 3(d), m is updated to 4 and the result Q_2 for $\texttt{RMS}(1, 3)$ on P_2 is $\{p_4, p_7, p_9\}$.

Theoretical Bound: The theoretical bound of FD-RMS is analyzed as follows. First of all, we need to verify the set-cover solution $\mathcal{C}$ maintained by Algorithms 2–4 is always stable. According to Lemma 1, it is guaranteed that the set-cover solution $\mathcal{C}$ returned by Algorithm 2 is stable. Then, we need to show it remains stable after the update procedures of Algorithms 3 and 4. In fact, both algorithms use Algorithm 1 to maintain the set-cover solution $\mathcal{C}$. Hence, the stability of $\mathcal{C}$ can be guaranteed by Lemma 2 since STABILIZE is always called after every update in Algorithm 1.

Next, we indicate the relationship between the result of k-RMS and the *set-cover* solution and provide the bound on the maximum-k regret ratio of Q_t returned by FD-RMS on P_t.

Theorem 2. *The result Q_t of FD-RMS is a $\big(k, O(\varepsilon^*_{k,r'} + \delta)\big)$-regret set of P_t with high probability where $r' = O(\frac{r}{\log m})$ and $\delta = O(m^{-\frac{1}{d}})$.*

Proof. Given a parameter $\delta > 0$, a δ-net [9] of $\mathbb{U}$ is a finite set $U \subset \mathbb{U}$ where there exists a vector $\overline{u}$ with $\|u - \overline{u}\| \leq \delta$ for



Tuple	x	y
p_1	0.2	1
p_2	0.6	0.8
p_3	0.7	0.5
p_4	1	0.1
p_5	0.4	0.3
p_6	0.2	0.7
p_7	0.3	0.9
p_8	0.6	0.6
p_9	0.9	0.6

(a) Dataset

Level	Range	Set	cov(S)
L_1	[2,4)	$S(p_1)$	u_2 u_5
		$S(p_4)$	u_1 u_4 u_6
L_0	[1,2)	$S(p_2)$	u_3

(b) Initial construction

Level	Range	Set	cov(S)
L_1	[2,4)	$S(p_9)$	u_3 u_4
		$S(p_1)$	u_2 u_5
		$S(p_4)$	u_1 u_6

(c) Add tuple p_9

Level	Range	Set	cov(S)
L_1	[2,4)	$S(p_9)$	u_3 u_4
L_0	[1,2)	$S(p_7)$	u_2
		$S(p_4)$	u_1

(d) Delete tuple p_1

Fig. 3. An example of using FD-RMS to process a k-RMS with $k = 1$ and $r = 3$

any $u \in \mathbb{U}$. Since a random set of $O(\frac{1}{\delta^{d-1}} \log \frac{1}{\delta})$ vectors in $\mathbb{U}$ is a δ-net with probability at least $\frac{1}{2}$ [9], one can generate a δ-net of size $O(\frac{1}{\delta^{d-1}} \log \frac{1}{\delta})$ with high probability by random sampling from $\mathbb{U}$ in $O(1)$ *repeated* trials.

Let B be the standard basis of $\mathbb{R}^d_+$ and U be a δ-net of $\mathbb{U}$ where $B = \{u_1, \ldots, u_d\} \subset U$. Since $p \in P_t$ is scaled to $p[i] \leq 1$ for $i \in [1, d]$, we have $\|p\| \leq \sqrt{d}$. According to the definition of δ-net, there exists a vector $\overline{u} \in U$ such that $\|\overline{u} - u\| \leq \delta$ for every $u \in \mathbb{U}$. Hence, for any tuple $p \in P_t$,

$$|\langle \overline{u}, p\rangle - \langle u, p\rangle| = |\langle \overline{u} - u, p\rangle| \leq \|\overline{u} - u\| \cdot \|p\| \leq \delta \cdot \sqrt{d} \quad (1)$$

Moreover, as Q_t corresponds to a set-cover solution $\mathcal{C}$ on Σ, there exists a tuple $q \in Q_t$ such that $\langle \overline{u}, q\rangle \geq (1-\varepsilon) \cdot \omega_k(\overline{u}, P_t)$ for any $\overline{u} \in U$. We first consider a basis vector $u_i \in U$ for some $i \in [1, d]$. We have $\omega(u_i, Q_t) \geq (1 - \varepsilon) \cdot \omega_k(u_i, P_t)$ and thus $\omega(u_i, Q_t) \geq (1 - \varepsilon) \cdot c$ where $c = \min_{i \in [1,d]} \omega_k(u_i, P_t)$. Since $\|u\| = 1$, there must exist some i with $u[i] \geq \frac{1}{\sqrt{d}}$ for any $u \in \mathbb{U}$. Therefore, it holds that $\omega(u, Q_t) \geq \omega(u_i, Q_t) \cdot \frac{1}{\sqrt{d}} \geq (1 - \varepsilon) \cdot \frac{c}{\sqrt{d}}$ for any $u \in \mathbb{U}$.

Next, we discuss two cases for $u \in \mathbb{U}$ separately.

- **Case 1** ($\omega_k(u, P_t) \leq \frac{c}{\sqrt{d}}$)**:** In this case, there always exists $q \in Q_t$ such that $\langle u, q\rangle \geq (1 - \varepsilon) \cdot \omega_k(u, P_t)$.
- **Case 2** ($\omega_k(u, P_t) > \frac{c}{\sqrt{d}}$)**:** Let $\overline{u} \in U$ be the utility vector such that $\|\overline{u} - u\| \leq \delta$. Let $\Phi_k(u, P_t) = \{p_1, \ldots, p_k\}$ be the top-k results of u on P_t. According to Equation 1, we have $\langle \overline{u}, p_i\rangle \geq \langle u, p_i\rangle - \delta \cdot \sqrt{d}$ for all $i \in [1, k]$ and thus $\langle \overline{u}, p_i\rangle \geq \omega_k(u, P_t) - \delta \cdot \sqrt{d}$. Thus, there exists k tuples in P_t with scores at least $\omega_k(u, P_t) - \delta \cdot \sqrt{d}$ for $\overline{u}$. We can acquire $\omega_k(\overline{u}, P_t) \geq \omega_k(u, P_t) - \delta \cdot \sqrt{d}$. Therefore, there exists $q \in Q_t$ such that

$$\begin{aligned}
\langle u, q\rangle &\geq \langle \overline{u}, q\rangle - \delta \cdot \sqrt{d} \geq (1 - \varepsilon) \cdot \omega_k(\overline{u}, P_t) - \delta \cdot \sqrt{d} \\
&\geq (1 - \varepsilon) \cdot \big(\omega_k(u, P_t) - \delta \cdot \sqrt{d}\big) - \delta \cdot \sqrt{d} \\
&\geq \big(1 - \varepsilon - \frac{(1 - \varepsilon)d\delta}{c} - \frac{d\delta}{c}\big) \cdot \omega_k(u, P_t) \\
&\geq \big(1 - \varepsilon - \frac{2d\delta}{c}\big) \cdot \omega_k(u, P_t)
\end{aligned}$$

Considering both cases, we have $\omega(u, Q_t) \geq (1 - \varepsilon - \frac{2d\delta}{c}) \cdot \omega_k(u, P_t)$ for any $u \in \mathbb{U}$ and thus $\mathtt{mrr}_k(Q_t)$ over P_t is at most $\varepsilon + \frac{2d\delta}{c}$. In all of our experiments, the value of c is always between 0.5 and 1, and thus we regard c as a constant in this proof. Therefore, Q_t is a $\big(k, O(\varepsilon + \delta)\big)$-regret set of P_t with high probability for any $c, d = O(1)$. Moreover, since FD-RMS uses m utility vectors including B to compute Q_t and $m = O(\frac{1}{\delta^{d-1}} \log \frac{1}{\delta})$, we can acquire $\delta = O(m^{-\frac{1}{d}})$.

Finally, because any (k, ε)-regret set of P_t corresponds to a set-cover solution on Σ (otherwise, the regret ratio is larger than ε for some utility vector) and the size of the optimal set-cover solution on Σ is $O(\frac{r}{\log m})$ according to Theorem 1, the maximum k-regret ratio of any size-r' subset of P_t is at least ε where $r' = O(\frac{r}{\log m})$, i.e., $\varepsilon^*_{k,r'} \geq \varepsilon$. Therefore, we conclude that Q_t is a $\big(k, O(\varepsilon^*_{k,r'} + \delta)\big)$-regret set of P_t with high probability. □

Finally, the upper bound of the maximum k-regret ratio of Q_t returned by FD-RMS on P_t is analyzed in the following corollary derived from the result of Theorem 2.

Corollary 1. *It satisfies that* $\mathtt{mrr}_k(Q_t) = O(r^{-\frac{1}{d}})$ *with high probability if we assume* $\varepsilon = O(m^{-\frac{1}{d}})$.

Proof. As indicated in the proof of Theorem 2, $\mathcal{U} = \{u_1, u_2, \ldots, u_m\}$ is a δ-net of $\mathbb{U}$ where $\delta = O(m^{-\frac{1}{d}})$ with high probability. Moreover, we have $\mathtt{mrr}_k(Q_t) = O(\varepsilon + \delta)$ and thus $\mathtt{mrr}_k(Q_t) = O(m^{-\frac{1}{d}})$ if $\varepsilon = O(m^{-\frac{1}{d}})$. In addition, at any time, $\mathcal{U}$ must have at least r utility vectors, i.e., $m \geq r$. Thus, we have $\mathtt{mrr}_k(Q_t) = O(r^{-\frac{1}{d}})$ since $m^{-\frac{1}{d}} \leq r^{-\frac{1}{d}}$ for any $d > 1$ and conclude the proof. □

Since ε is tunable in FD-RMS, by trying different values of ε, we can always find an appropriate one such that $\varepsilon = O(m^{-\frac{1}{d}})$. Hence, from Corollary 1, we show that the upper bound of FD-RMS is slightly higher than that of CUBE [1] and SPHERE [13] (i.e., $O(r^{-\frac{1}{d-1}})$) under a mild assumption.

Complexity Analysis: First, we use tree-based methods to maintain the approximate top-k results for FD-RMS (see Section III-C for details). Here, the time complexity of each top-k query is $O(n_0)$ where $n_0 = |P_0|$ because the size of ε-approximate top-k tuples can be $O(n_0)$. Hence, it takes $O(M \cdot n_0)$ time to compute the top-k results. Then, GREEDY runs $O(r)$ iterations to get a set-cover solution. At each iteration, it evaluates $O(n_0)$ sets to find S^* in Line 16 of Algorithm 1. Thus, the time complexity of GREEDY is $O(r \cdot n_0)$. FD-RMS calls GREEDY $O(\log M)$ times to determine the value of m. Therefore, the time complexity of computing Q_0 on P_0 is $O\big((M + r \log M) \cdot n_0\big)$. In Algorithm 3, the time complexity of updating the top-k results and set system Σ is $O\big(\mathtt{u}(\Delta_t) \cdot n_t\big)$ where $\mathtt{u}(\Delta_t)$ is the number of utility vectors whose top-k results are changed by Δ_t. Then, the maximum number of reassignments in cover sets is $|S(p)|$ for Δ_t, which is bounded by $O(\mathtt{u}(\Delta_t))$. In addition, the time complexity of STABILIZE is $O(m \log m)$ according to Lemma 2. Moreover, the maximum difference between the old and new values of m is bounded by $O(m)$. Hence, the total time complexity of updating Q_t w.r.t. Δ_t is $O\big(\mathtt{u}(\Delta_t) \cdot n_t + m^2 \log m\big)$.

C. Implementation Issues

Index Structures: As indicated in Line 2 of Algorithm 2 and Line 3 of Algorithm 3, FD-RMS should compute the ε-approximate top-k result of each u_i ($i \in [1, M]$) on P_0 and update it w.r.t. Δ_t. Here, we elaborate on our implementation for top-k maintenance. In order to process a large number of (approximate) top-k queries with frequent updates in the database, we implement a *dual-tree* [21]–[23] that comprises a tuple index TI and a utility index UI.

The goal of TI is to efficiently retrieve the ε-approximate top-k result $\Phi_{k,\varepsilon}(u, P_t)$ of any utility vector u on the up-to-date P_t. Hence, any space-partitioning index, e.g., *k-d tree* [24] and *Quadtree* [25], can serve as TI for top-k query processing. In practice, we use *k-d tree* as TI. We adopt the scheme of [26] to transform a top-k query in $\mathbb{R}^d$ into a kNN query in $\mathbb{R}^{d+1}$. Then, we implement the standard top-down methods to construct TI on P_0 and update it w.r.t. Δ_t. The branch-and-bound algorithm is used for top-k queries on TI.

The goal of UI is to cluster the sampled utility vectors so as to efficiently find each vector whose ε-approximate top-k result is updated by Δ_t. Since the top-k results of linear functions are merely determined by directions, the basic idea of UI is to cluster the utilities with high *cosine similarities* together. Therefore, we adopt an angular-based binary space partitioning tree called *cone tree* [21] as UI. We generally follow Algorithms 8–9 in [21] to build UI for $\{u_1, \ldots, u_M\}$. We implement a top-down approach based on Section 3.2 of [22] to update the top-k results affected by Δ_t.

Parameter Tuning: Now, we discuss how to specify the values of ε, i.e., the approximation factor of top-k queries, and M, i.e., the upper bound of m, in FD-RMS. In general, the value of ε has direct effect on m as well as the efficiency and quality of results of FD-RMS. In particular, if ε is larger, the ε-approximate top-k result of each utility vector will include more tuples and the set system built on top-k results will be more dense. As a result, to guarantee the result size to be exactly r, FD-RMS will use more utility vectors (i.e., a larger m) for a larger ε. Therefore, a smaller ε leads to higher efficiency and lower solution quality due to smaller m and larger δ, and vice versa. In our implementation, we use a trial-and-error method to find appropriate values of ε and M: For each query $\mathtt{RMS}(k, r)$ on a dataset, we test different values of ε chosen from $[0.0001, \ldots, 0.1024]$ and, for each value of ε, M is set to the smallest one chosen from $[2^{10}, \ldots, 2^{20}]$ that always guarantees $m < M$. If the result size is still smaller than r when $m = M = 2^{20}$, we will not use larger M anymore due to efficiency issue. The values of ε and M that strike the best balance between efficiency and quality of results will be used. In Fig. 5, we present how the value of ε affects the performance of FD-RMS empirically.

IV. EXPERIMENTS

In this section, we evaluate the performance of FD-RMS on real-world and synthetic datasets. We first introduce the experimental setup in Section IV-A. Then, we present the experimental results in Section IV-B.

A. Experimental Setup

Algorithms: The algorithms compared are listed as follows.

- GREEDY: the greedy algorithm for 1-RMS in [1].
- GREEDY*: the randomized greedy algorithm for k-RMS when $k > 1$ proposed in [8].
- GEOGREEDY: a variation of GREEDY for 1-RMS in [7].
- DMM-RRMS: a discretized matrix min-max based algorithm for 1-RMS in [11].
- ε-KERNEL: computing an ε-kernel coreset as the k-RMS result [9], [10] directly.
- HS: a hitting-set based algorithm for k-RMS in [9].
- SPHERE: an algorithm that combines ε-KERNEL with GREEDY for 1-RMS in [13].
- URM: a k-MEDOID clustering based algorithm for 1-RMS in [14]. Following [14], we use DMM-RRMS to compute an initial solution for URM.
- FD-RMS: our fully-dynamic k-RMS algorithm proposed in Section III.

The algorithms that only work in two dimensions are not compared. All the above algorithms except FD-RMS and URM[3] cannot directly work in a fully dynamic setting. In our experiments, they rerun from scratch to compute the up-to-date k-RMS result once the skyline is updated by any insertion or deletion. In addition, the algorithms that are not applicable when $k > 1$ are not compared for the experiments with varying k. Since ε-KERNEL and HS are proposed for min-size k-RMS that returns the smallest subset whose maximum k-regret ratio is at most ε, we adapt them to our problem by performing a binary search on ε in the range $(0, 1)$ to find the smallest value of ε that guarantees the result size is at most r.

Our implementation of FD-RMS and URM is in Java 8 and published on GitHub[4]. We used the C++ implementations of baseline algorithms published by authors and followed the default parameter settings as described in the original papers. All the experiments were conducted on a server running Ubuntu 18.04.1 with a 2.3GHz processor and 256GB memory.

Datasets: The datasets we use are listed as follows.

- **BB**[5] is a basketball dataset that contains $21,961$ tuples, each of which represents one player/season combination with 5 attributes such as *points* and *rebounds*.
- **AQ**[6] includes hourly air-pollution and weather data from 12 monitoring sites in Beijing. It has $382,168$ tuples and

[3] The original URM in [14] is also a static algorithm. We extend URM to support dynamic updates as described in the technical report [27].

[4] https://github.com/yhwang1990/dynamic-rms

[5] www.basketball-reference.com

[6] archive.ics.uci.edu/ml/datasets/Beijing+Multi-Site+Air-Quality+Data

TABLE I
STATISTICS OF DATASETS

Dataset	n	d	*#skylines*	*updates* (%)
BB	$21,961$	5	200	1.07
AQ	$382,168$	9	$21,065$	5.60
CT	$581,012$	8	$77,217$	13.4
Movie	$13,176$	12	$3,293$	26.5
Indep	100K–1M	2–10	see Fig. 4	
AntiCor	100K–1M	2–10	see Fig. 4	

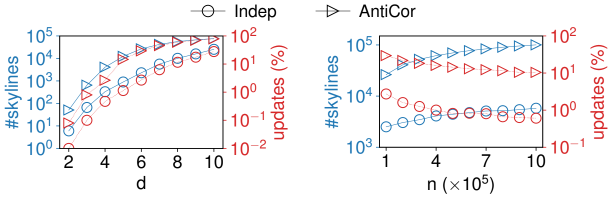


Fig. 4. Sizes and update rates of the skylines of synthetic datasets

each tuple has 9 attributes including the concentrations of 6 air pollutants like $PM_{2.5}$, as well as 3 meteorological parameters like *temperature*.

- **CT**[7] contains the cartographic data of forest covers in the Roosevelt National Forest of northern Colorado. It has $581,012$ tuples and we choose 8 numerical attributes, e.g., *elevation* and *slope*, for evaluation.
- **Movie**[8] is the tag genome dataset published by MovieLens. We extract the relevance scores of $13,176$ movies and 12 tags for evaluation. Each tuple represents the relevance scores of 12 tags to a movie.
- **Indep** is generated as described in [6]. It is a set of uniform points on the unit hypercube where different attributes are independent of each other.
- **AntiCor** is also generated as described in [6]. It is a set of random points with anti-correlated attributes.

The statistics of datasets are reported in Table I. Here, n is the number of tuples; d is the dimensionality; *#skylines* is the number of tuples on the skyline; and *updates* (%) is the percentage of tuple operations that trigger any update on the skyline. Note that we generated several **Indep** and **AntiCor** datasets by varying n from 100K to 1M and d from 2 to 10 for scalability tests. By default, we used the ones with $n = 100$K and $d = 6$. The sizes and update rates of the skylines of synthetic datasets are shown in Fig. 4.

Workloads: The workload of each experiment was generated as follows: First, we randomly picked 50% of tuples as the initial dataset P_0; Second, we inserted the remaining 50% of tuples one by one into the dataset to test the performances for insertions; Third, we randomly deleted 50% of tuples one by one from the dataset to test the performances for deletions. It is guaranteed that the orders of operations kept the same for all algorithms. The k-RMS results were recorded 10 times when $10\%, 20\%, \ldots, 100\%$ of the operations were performed.

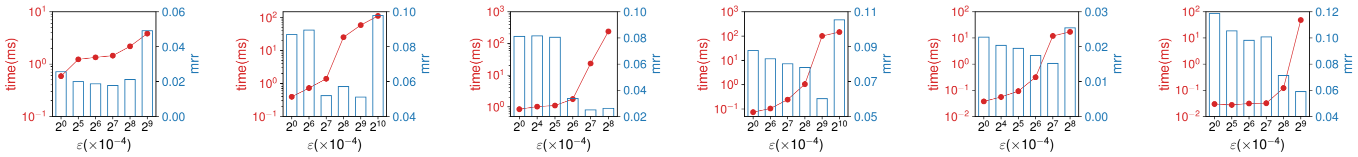
Performance Measures: The efficiency of each algorithm was measured by *average update time*, i.e., the average wall-clock time used per operation. For the static algorithms, we only took the time for k-RMS computation into account and ignored the time for skyline maintenance for fair comparison. The quality of results was measured by the *maximum k-regret ratio* ($\mathtt{mrr}_k$) for a given size constraint r, and, of course, the smaller $\mathtt{mrr}_k$ the better. To compute $\mathtt{mrr}_k(Q)$ of a result Q, we generated a test set of 500K random utility vectors and used the maximum regret value found as our estimate. Since the k-RMS results were recorded 10 times for each query, we reported the average of the maximum k-regret ratios of 10 results for evaluation.

B. *Experimental Results*

Effect of parameter ε on FD-RMS: In Fig. 5, we present the effect of the parameter ε on the performance of FD-RMS. We report the update time and maximum regret ratios of FD-RMS for $k = 1$ and $r = 50$ on each dataset (except $r = 20$ on **BB**) with varying ε. We use the method described in Section III-C to set the value of M for each value of ε. First of all, the update time of FD-RMS increases significantly with ε. This is because both the time to process an ε-approximate top-k query and the number of top-k queries (i.e., M) grow with ε, which requires a larger overhead to maintain both top-k results and set-cover solutions. Meanwhile, the quality of results first becomes better when ε is larger but then could degrade if ε is too large. The improvement in quality with increasing ε is attributed to larger m and thus smaller δ. However, once ε is greater than the maximum regret ratio $\varepsilon^*_{k,r}$ of the optimal result (whose upper bound can be inferred from practical results), e.g., $\varepsilon = 0.0512$ on **BB**, the result of FD-RMS will contain less than r tuples and its maximum regret ratio will be close to ε no matter how large m is. To sum up, by setting ε to the one that is slightly lower than $\varepsilon^*_{k,r}$ among $[0.0001, \ldots, 0.1024]$, FD-RMS performs better in terms of both efficiency and solution quality, and the values of ε in FD-RMS are decided in this way for the remaining experiments.

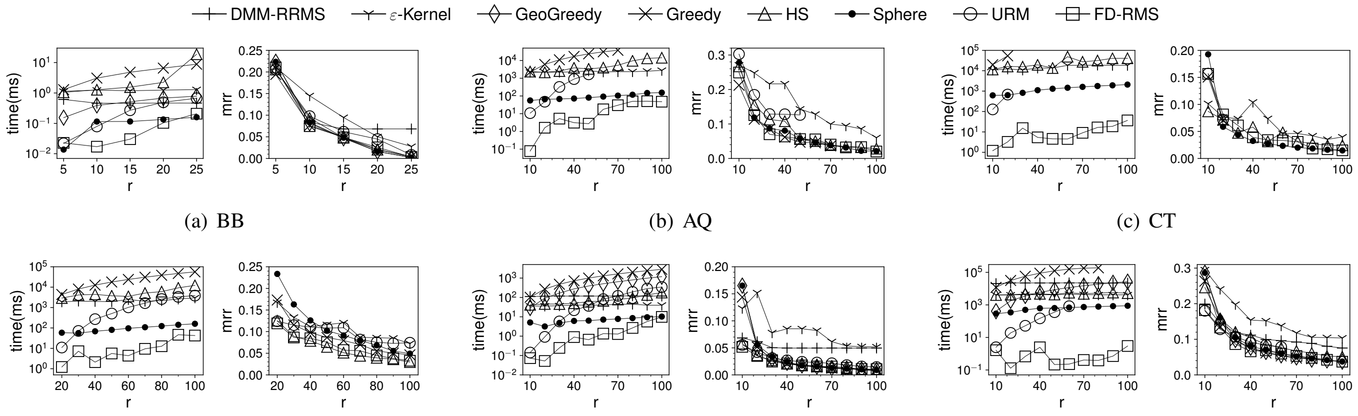
Effect of result size r: In Fig. 6, we present the performance of different algorithms for 1-RMS (a.k.a. r-regret query) with varying r. In particular, r is ranged from 10 to 100 on each dataset (except **BB** where r is ranged from 5 to 25). In general, the update time of each algorithm grows while the maximum regret ratios drop with increasing r. But, for FD-RMS, it could take less update time when r is larger in some cases. The efficiency of FD-RMS is positively correlated with m but negatively correlated with ε. On a specific dataset, FD-RMS typically chooses a smaller ε when r is large, and vice versa. When ε is smaller, m may decrease even though r is larger. Therefore, the update time of FD-RMS decreases with r in some cases because a smaller ε is used. Among all algorithms tested, GREEDY is the slowest and fails to provide results within one day on **AQ**, **CT**, and **AntiCor** when $r > 80$. GEOGREEDY runs much faster than GREEDY while having equivalent quality on low-dimensional data. However, it cannot scale up to high dimensions (i.e., $d > 7$) because the cost

[7] archive.ics.uci.edu/ml/datasets/covertype
[8] grouplens.org/datasets/movielens



(a) BB (b) AQ (c) CT (d) Movie (e) Indep (f) AntiCor

Fig. 5. Performance of FD-RMS with varying ε ($k = 1$; $r = 20$ for BB and $r = 50$ for other datasets). Note that the red line represents the update time and the blue bars denote the maximum regret ratios.



(a) BB (b) AQ (c) CT

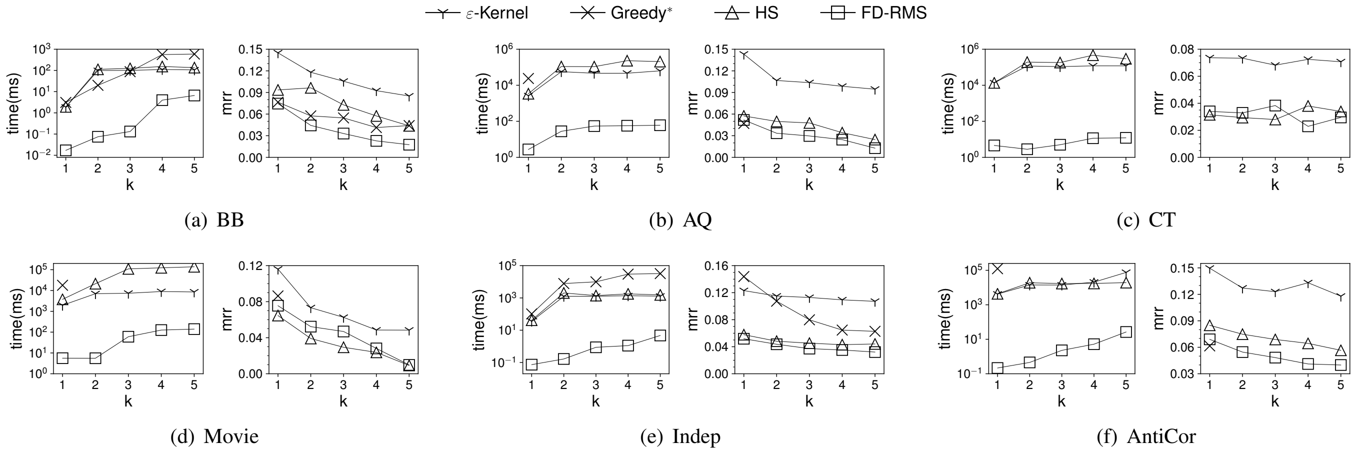
(d) Movie (e) Indep (f) AntiCor

Fig. 6. Update time and maximum regret ratios with varying the result size r ($k = 1$)

of finding *happy points* grows significantly with d. DMM-RRMS suffers from two drawbacks: (1) it also cannot scale up to $d > 7$ due to huge memory consumption; (2) its solution quality is not competitive when $r \geq 50$ because of the sparsity of space discretization. The solution quality of ε-KERNEL is generally inferior to any other algorithm because the size of an ε-kernel coreset is much larger than the size of the minimum $(1, \varepsilon)$-regret set. Although HS provides results of good quality in most cases, it runs several orders of magnitude slower than FD-RMS. SPHERE demonstrates better performance than other static algorithms. Nevertheless, its efficiency is still much lower than FD-RMS, especially on datasets with large skyline sizes, e.g., **CT** and **AntiCor**, where FD-RMS runs up to three orders of magnitude faster. URM shows good performance in both efficiency and solution quality for small r (e.g., $r \leq 20$) and skyline sizes (e.g., on **BB** and **Indep**). However, it scales poorly to large r and skyline sizes because the convergence of k-MEDOID becomes very slow and the number of linear programs for regret computation grows rapidly when r and and skyline sizes increase. In addition, URM does not provide high-quality results in many cases since k-MEDOID cannot escape from local optima. To sum up, FD-RMS outperforms all other algorithms for fully-dynamic 1-RMS in terms of efficiency. Meanwhile, the maximum regret ratios of the results of FD-RMS are very close (the differences are less than 0.01 in almost all cases) to the best of static algorithms.

Effect of k: The results for k-RMS with varying k from 1 to 5 are illustrated in Fig. 7. We only compare FD-RMS with GREEDY*, ε-KERNEL, and HS because other algorithms are not applicable to the case when $k > 1$. We set $r = 10$ for **BB** and **Indep** and $r = 50$ for the other datasets. The results of GREEDY* for $k > 1$ are only available on **BB** and **Indep**. For the other datasets, GREEDY* fails to return any result within one day when $k > 1$. We can see all algorithms run much slower when k increases. For FD-RMS, lower efficiencies are caused by higher cost of maintaining top-k results. HS and ε-KERNEL must consider all tuples in the datasets instead of only skylines to validate that the maximum k-regret ratio is at most ε when $k > 1$. For GREEDY*, the number of linear programs to compute k-regret ratios increases drastically with k. Meanwhile, the maximum k-regret ratios drop with k, which is obvious according to its definition. FD-RMS achieves speedups of up to four orders of magnitude than the baselines on all datasets. At the same time, the solution quality of FD-RMS is also better on all datasets except **Movie** and **CT**, where the results of HS are of slightly higher quality in some cases.

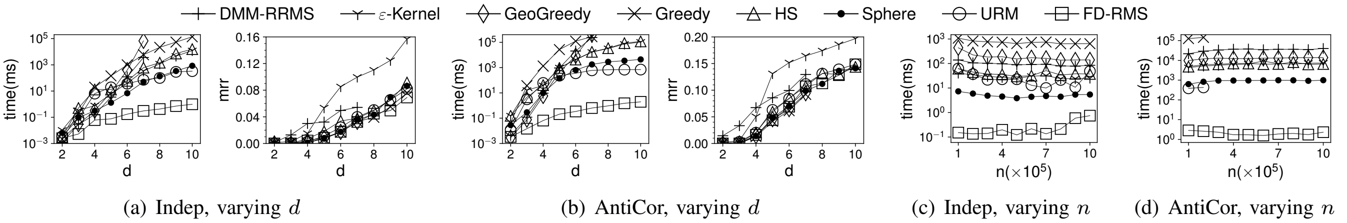
Scalability: Finally, we evaluate the scalability of different algorithms w.r.t. the dimensionality d and dataset size n. To test the impact of d, we fix $n = 100$K, $k = 1$, $r = 50$, and vary d from 2 to 10. The performance with varying d is shown in Fig. 8(a)–8(b). Both the update time and maximum regret ratios of all algorithms increase dramatically with d. Although almost all algorithms show good performance when $d = 2, 3$, most of them quickly become very inefficient in high dimensions. Nevertheless, FD-RMS has a significantly better scalability w.r.t. d: It achieves speedups of at least 100 times over any other algorithm while providing results of equivalent quality when $d \geq 7$.



(a) BB (b) AQ (c) CT

(d) Movie (e) Indep (f) AntiCor

Fig. 7. Update time and maximum regret ratios with varying k ($r = 10$ for BB and Indep; $r = 50$ for other datasets)



(a) Indep, varying d (b) AntiCor, varying d (c) Indep, varying n (d) AntiCor, varying n

Fig. 8. Scalability with varying the dimensionality d and dataset size n ($k = 1$, $r = 50$)

To test the impact of n, we fix $d = 6$, $k = 1$, $r = 50$, and vary n from 100K to 1M. The update time with varying n is shown in Fig. 8(c)–8(d). For static algorithms, we observe different trends in efficiency on two datasets: The update time slightly drops on **Indep** but keeps steady on **AntiCor**. The efficiencies are determined by two factors, i.e., *the number of tuples on the skyline* and *the frequency of skyline updates*. As shown in Fig. 4, when n is larger, the number of tuples on the skyline increases but the frequency of skyline updates decreases. On **Indep**, the benefits of lower update frequencies outweigh the cost of more skyline tuples; on **AntiCor**, two factors cancel each other. FD-RMS runs slower when n increases due to higher cost of maintaining top-k results on **Indep**. But, on **AntiCor**, the update time keeps steady with n because of smaller values of ε and m, which cancel the higher cost of maintaining top-k results. In addition, since the maximum regret ratios are not obviously affected by n, we omit the results here and leave them to the technical report [27]. Generally, FD-RMS always outperforms all baselines for different values of n.

V. Related Work

There have been extensive studies on the k-regret minimizing set (k-RMS) problem (see [28] for a survey). Nanongkai et al. [1] first introduced the notions of *maximum regret ratio* and *r-regret query* (i.e., maximum 1-regret ratio and 1-RMS in this paper). They proposed the CUBE algorithm to provide an upper-bound guarantee for the maximum regret ratio of the optimal solution of 1-RMS. They also proposed the GREEDY heuristic for 1-RMS, which always picked a tuple that maximally reduced the maximum regret ratio at each iteration. Peng and Wong [7] proposed the GEOGREEDY algorithm to improve the efficiency of GREEDY by utilizing the geometric properties of 1-RMS. Asudeh et al. [11] proposed two discretized matrix min-max (DMM) based algorithms for 1-RMS. Xie et al. [13] designed the SPHERE algorithm for 1-RMS based on the notion of ε-kernel [29]. Shetiya et al. [14] proposed a unified algorithm called URM based on k-MEDOID clustering for l^p-norm RMS problems, of which 1-RMS was a special case when $p = \infty$. The aforementioned algorithms cannot be used for k-RMS when $k > 1$. Chester et al. [8] first extended the notion of 1-RMS to k-RMS. They also proposed a randomized GREEDY* algorithm that extended the GREEDY heuristic to support k-RMS when $k > 1$. The min-size version of k-RMS that returned the minimum subset whose maximum k-regret ratio was at most ε for a given $\varepsilon \in (0, 1)$ was studied in [9], [12]. They proposed two algorithms for min-size k-RMS based on the notion of ε-kernel [29] and hitting-set, respectively. However, all above algorithms are designed for the static setting and very inefficient to process database updates. To the best of our knowledge, FD-RMS is the first k-RMS algorithm that is optimized for the fully dynamic setting and efficiently maintains the result for dynamic updates.

Different variations of regret minimizing set problems were also studied recently. The 1-RMS problem with nonlinear utility functions were studied in [30]–[32]. Specifically, they generalized the class of utility functions to *convex functions* [30], *multiplicative functions* [31], and *submodular functions* [32], respectively. Asudeh et al. [33] proposed the rank-regret representative (RRR) problem. The difference between RRR and RMS is that the regret in RRR is defined by ranking while the regret in RMS is defined by score. Several studies [14], [34], [35] investigated the *average regret minimization* (ARM) problem. Instead of minimizing the maximum regret ratio, ARM returns a subset of r tuples such that the average regret of all possible users is minimized. The problem of *interactive regret minimization* that aimed to enhance the regret minimization problem with user interactions was studied in [36], [37]. Xie et al. [38] proposed a variation of min-size RMS called α-happiness query. Since these variations have different formulations from the original k-RMS problem, the algorithms proposed for them cannot be directly applied to the k-RMS problem. Moreover, these algorithms are still proposed for the static setting without considering database updates.

VI. Conclusion

In this paper, we studied the problem of maintaining k-regret minimizing sets (k-RMS) on dynamic datasets with arbitrary insertions and deletions of tuples. We proposed the first fully-dynamic k-RMS algorithm called FD-RMS. FD-RMS was based on transforming fully-dynamic k-RMS to a dynamic set cover problem, and it could dynamically maintain the result of k-RMS with a theoretical guarantee. Extensive experiments on real-world and synthetic datasets confirmed the efficiency, effectiveness, and scalability of FD-RMS compared with existing static approaches to k-RMS. For future work, it would be interesting to investigate whether our techniques can be extended to k-RMS and related problems on higher dimensions (i.e., $d > 10$) or with nonlinear utility functions (e.g., [30]–[32]) in dynamic settings.

Acknowledgment

We thank anonymous reviewers for their helpful comments. Yanhao Wang has been supported by the MLDB project of Academy of Finland (decision number: 322046). The research of Raymond Chi-Wing Wong is supported by HKRGC GRF 16219816.

References

[1] D. Nanongkai, A. D. Sarma, A. Lall, R. J. Lipton, and J. Xu, "Regret-minimizing representative databases," *PVLDB*, vol. 3, no. 1, pp. 1114–1124, 2010.

[2] J. Stoyanovich, K. Yang, and H. V. Jagadish, "Online set selection with fairness and diversity constraints," in *EDBT*, 2018, pp. 241–252.

[3] Y. Wang, Y. Li, and K. Tan, "Efficient representative subset selection over sliding windows," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 7, pp. 1327–1340, 2019.

[4] N. N. Liu, X. Meng, C. Liu, and Q. Yang, "Wisdom of the better few: cold start recommendation via representative based rating elicitation," in *RecSys*, 2011, pp. 37–44.

[5] Y. Wang, Y. Li, and K. Tan, "Coresets for minimum enclosing balls over sliding windows," in *KDD*, 2019, pp. 314–323.

[6] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001, pp. 421–430.

[7] P. Peng and R. C. Wong, "Geometry approach for k-regret query," in *ICDE*, 2014, pp. 772–783.

[8] S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides, "Computing k-regret minimizing sets," *PVLDB*, vol. 7, no. 5, pp. 389–400, 2014.

[9] P. K. Agarwal, N. Kumar, S. Sintos, and S. Suri, "Efficient algorithms for k-regret minimizing sets," in *SEA*, 2017, pp. 7:1–7:23.

[10] W. Cao, J. Li, H. Wang, K. Wang, R. Wang, R. C. Wong, and W. Zhan, "K-regret minimizing set: Efficient algorithms and hardness," in *ICDT*, 2017, pp. 11:1–11:19.

[11] A. Asudeh, A. Nazi, N. Zhang, and G. Das, "Efficient computation of regret-ratio minimizing set: A compact maxima representative," in *SIGMOD*, 2017, pp. 821–834.

[12] N. Kumar and S. Sintos, "Faster approximation algorithm for the k-regret minimizing set and related problems," in *ALENEX*, 2018, pp. 62–74.

[13] M. Xie, R. C. Wong, J. Li, C. Long, and A. Lall, "Efficient k-regret query algorithm with restriction-free bound for any dimensionality," in *SIGMOD*, 2018, pp. 959–974.

[14] S. Shetiya, A. Asudeh, S. Ahmed, and G. Das, "A unified optimization algorithm for solving "regret-minimizing representative" problems," *PVLDB*, vol. 13, no. 3, pp. 239–251, 2019.

[15] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, 1972, pp. 85–103.

[16] U. Feige, "A threshold of ln *n* for approximating set cover," *J. ACM*, vol. 45, no. 4, pp. 634–652, 1998.

[17] S. Bhattacharya, M. Henzinger, and G. F. Italiano, "Deterministic fully dynamic data structures for vertex cover and matching," *SIAM J. Comput.*, vol. 47, no. 3, pp. 859–887, 2018.

[18] A. Gupta, R. Krishnaswamy, A. Kumar, and D. Panigrahi, "Online and dynamic algorithms for set cover," in *STOC*, 2017, pp. 537–550.

[19] A. Abboud, R. Addanki, F. Grandoni, D. Panigrahi, and B. Saha, "Dynamic set cover: improved algorithms and lower bounds," in *STOC*, 2019, pp. 114–125.

[20] N. Hjuler, G. F. Italiano, N. Parotsidis, and D. Saulpic, "Dominating sets and connected dominating sets in dynamic graphs," in *STACS*, 2019, pp. 35:1–35:17.

[21] P. Ram and A. G. Gray, "Maximum inner-product search using cone trees," in *KDD*, 2012, pp. 931–939.

[22] A. Yu, P. K. Agarwal, and J. Yang, "Processing a large number of continuous preference top-k queries," in *SIGMOD*, 2012, pp. 397–408.

[23] R. R. Curtin, A. G. Gray, and P. Ram, "Fast exact max-kernel search," in *SDM*, 2013, pp. 1–9.

[24] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[25] R. A. Finkel and J. L. Bentley, "Quad trees: A data structure for retrieval on composite keys," *Acta Inf.*, vol. 4, pp. 1–9, 1974.

[26] Y. Bachrach, Y. Finkelstein, R. Gilad-Bachrach, L. Katzir, N. Koenigstein, N. Nice, and U. Paquet, "Speeding up the xbox recommender system using a euclidean transformation for inner-product spaces," in *RecSys*, 2014, pp. 257–264.

[27] Y. Wang, Y. Li, R. C.-W. Wong, and K.-L. Tan, "A fully dynamic algorithm for k-regret minimizing sets," https://arxiv.org/abs/2005.14493, 2020.

[28] M. Xie, R. C. Wong, and A. Lall, "An experimental survey of regret minimization query and variants: bridging the best worlds between top-k query and skyline query," *VLDB J.*, vol. 29, no. 1, pp. 147–175, 2020.

[29] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan, "Approximating extent measures of points," *J. ACM*, vol. 51, no. 4, pp. 606–635, 2004.

[30] T. K. Faulkner, W. Brackenbury, and A. Lall, "K-regret queries with nonlinear utilities," *PVLDB*, vol. 8, no. 13, pp. 2098–2109, 2015.

[31] J. Qi, F. Zuo, H. Samet, and J. C. Yao, "K-regret queries using multiplicative utility functions," *ACM Trans. Database Syst.*, vol. 43, no. 2, pp. 10:1–10:41, 2018.

[32] T. Soma and Y. Yoshida, "Regret ratio minimization in multi-objective submodular function maximization," in *AAAI*, 2017, pp. 905–911.

[33] A. Asudeh, A. Nazi, N. Zhang, G. Das, and H. V. Jagadish, "RRR: Rank-regret representative," in *SIGMOD*, 2019, pp. 263–280.

[34] S. Zeighami and R. C. Wong, "Finding average regret ratio minimizing set in database," in *ICDE*, 2019, pp. 1722–1725.

[35] S. Storandt and S. Funke, "Algorithms for average regret minimization," in *AAAI*, 2019, pp. 1600–1607.

[36] D. Nanongkai, A. Lall, A. D. Sarma, and K. Makino, "Interactive regret minimization," in *SIGMOD*, 2012, pp. 109–120.

[37] M. Xie, R. C. Wong, and A. Lall, "Strongly truthful interactive regret minimization," in *SIGMOD*, 2019, pp. 281–298.

[38] M. Xie, R. C. Wong, P. Peng, and V. J. Tsotras, "Being happy with the least: Achieving α-happiness with minimum number of tuples," in *ICDE*, 2020, pp. 1009–1020.