Secure Dynamic Skyline Queries Using Result Materialization

Sepanta Zeighami
University of Southern California
zeighami@usc.edu

Gabriel Ghinita
University of Massachusetts Boston
gabriel.ghinita@umb.edu

Cyrus Shahabi
University of Southern California
shahabi@usc.edu

ABSTRACT

Skyline computation is an increasingly popular query, with broad applicability in domains such as healthcare, travel and finance. Given the recent trend to outsource databases and query evaluation, and due to the proprietary and sometimes highly sensitivity nature of the data (e.g., in healthcare), it is essential to evaluate skylines on encrypted datasets. Several research efforts acknowledged the importance of secure skyline computation, but existing solutions suffer from at least one of the following shortcomings: (i) they only provide ad-hoc security; (ii) they are prohibitively expensive; or (iii) they rely on unrealistic assumptions, such as the presence of multiple non-colluding parties in the protocol.

Inspired from solutions for secure nearest-neighbors (NN) computation, we conjecture that the most secure and efficient way to compute skylines is through result materialization. However, this approach is significantly more challenging for skylines than for NN queries. We exhaustively study and provide algorithms for pre-computation of skyline results, and we perform an in-depth theoretical analysis of this process. We show that pre-computing results while minimizing storage overhead is NP-hard, and we provide dynamic programming and greedy heuristics that solve the problem more efficiently, while maintaining storage at reasonable levels. Our algorithms are novel and applicable to plain-text skyline computation, but we focus on the encrypted setting where materialization reduces the cost of skyline computation from hours to seconds. Extensive experiments show that we clearly outperform existing work in terms of performance, and our security analysis proves that we obtain a smaller (and quantifiable) data leakage than competitors.

1. INTRODUCTION

The skyline query finds points in a dataset which are not dominated by any other data point in at least one attribute value. These points have the property of "standing out" among other data points. For instance, in airfare booking, the skyline may contain routes that are either cheapest, shortest, or have the fewest stopovers. In a hospital database, the skyline may contain patients with lowest age, or patients with minimum value for a certain test result (e.g., hemoglobin level). Many research efforts in the past decade focused on efficient computation of skylines over plaintext data [1, 16, 4]. However, few solutions exist for the problem of *secure* skyline, where the data and query execution are outsourced to a service provider (SP). Since the data may be proprietary or protected by law (e.g., healthcare records), the computation must be executed over *encrypted* datasets.

The work in [2] was the first to formulate the secure skyline problem, but the solution proposed only provided ad-hoc security. Later in [24, 22, 14], several solutions were proposed that used either homomorphic encryption, or secure multi-party computation. However, they either leak excessive information to the SP, or they incur prohibitive computation and communication costs. The state-of-the-art approach in [22] assumes a system architecture with two non-colluding parties that engage in a secure multi-party protocol that needs to scan the entire dataset for each query, and perform expensive operations for a large subset of the Cartesian product of all records. This results in a response time of around 3 hours for a single query, which is clearly impractical. Each query starts anew, and cannot use any information computed form the previous query.

We propose a different approach, which has been shown in the seminal work of [30] to be the only secure and efficient approach for computing nearest-neighbor (NN) queries on encrypted data. The main idea of [30] is to pre-compute query results using a Voronoi diagram, and then partition and materialize the results in a data structure whose properties are not dependent on the data characteristics (to minimize leakage). At query time, the user provides an encrypted representation of the query point, and then the SP and the user engage in an interactive protocol that allows the user to retrieve the partition that contains the results to the query (together with a number of possible additional results, i.e., false positives)[1]. It is shown in [30] that any method that uses some sort of encrypted processing directly on the data points leaks a significant amount of information, in the form of either inter-point distances, or distance order.

Inspired by [30], we extend the idea of pre-computation and partitioning to the skyline query. This leads to a more secure solution, and a much faster response time, as we do *not* process the entire dataset for each query. However, pre-

Proceedings of the VLDB Endowment, Vol. XX, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

[1]The Voronoi diagram materialization used in SNN was first introduced by the authors of this submission in [10] for private NN queries on public datasets.

computing skyline results in the dynamic case (i.e., for every possible query point) is a very challenging problem. In fact, the equivalent of this problem on *plaintext* has attracted limited attention, because doing so would be too expensive. Contrast this to the case of NN queries, where Voronoi diagrams have been extensively used even for plaintext queries. Our work is pioneering in that it provides a thorough analysis of skyline result pre-computation, which may find applications beyond encrypted data. Although expensive in comparison to other plaintext skyline computation counterparts, we believe that the pre-computation approach becomes valuable, and in fact the only viable approach, in the context of encrypted data. This is because techniques that do not perform materialization require hours of processing for a single query. Our approach can answer a query in less than a second, even though there is a one-time setup cost.

In a nutshell, our materialization approach reduces the skyline query to a simple index look-up. First, we perform a partitioning of the space into non-overlapping regions called *skyline tiles*, such that the answer to all skyline queries that fall within the same tile are identical. This is done by finding, for each data point, the space of queries for which the data point is in the skyline (see Figs. 1 and 2). Then, skyline tiles are created by intersecting these regions (see Fig. 3). We store the query answer in each tile, and we index the tiles in a data structure. This way, we can answer a dynamic skyline query by simply performing a lookup in the index. The index is then encrypted to ensure the security of our approach. Finally, note that reducing skyline query computation to an index look-up allows us to utilize significantly less expensive cryptographic primitives, which manifests itself in orders of magnitude improvement in query time compared with existing approaches.

Our specific contributions are:

1. We comprehensively study the problem of pre-computing skyline results, and we perform an in-depth theoretical analysis of this process.

2. We show that the problem of pre-computing results while minimizing storage overhead is NP-hard, and we provide dynamic programming and greedy heuristics that solve the problem more efficiently, while maintaining storage costs at reasonable levels.

3. We perform an extensive experimental evaluation showing that our techniques clearly outperform existing work in terms of performance.

4. We provide an in-depth security analysis to measure the leakage of our proposed approach, and conclude that the amount of leakage is quantifiable, and smaller compared to existing techniques.

The rest of the paper is organized as follows: we provide background information in Section 2. Section 3 introduces a construction to pre-compute skyline query results for data units called *tiles*. We show how to aggregate tiles and perform data partitioning to reduce storage overhead in Section 4. We generalize the tile concept in Section 5 to obtain further performance gains, and outline the complete, end-to-end solution to the secure skyline query in Section 6. Section 7 presents our security analysis, followed by experiments in Section 8. We review related work in Section 9 and conclude in Section 10.

2. BACKGROUND AND DEFINITIONS

2.1 Preliminaries

Consider database D with n points in the d-dimensional space R^d. For point $p \in D$, $p[i]$ denotes its value in i^{th} dimension. A query is denoted by $q \in R^d$. For ease of discussion, let ∞ be a large constant and assume that the query space is bounded by ∞ in every dimension (i.e., $q[i] < \infty$ for all i). The *domination* relationship between two points in D with respect to a query q is defined as follows:

Definition 1 (Domination)**.** *A point $p \in D$ dominates another point $p' \in D$ with respect to q, denoted by $p >_q p'$, if and only if $\forall i, 1 \leq i \leq d, |p[i] - q[i]| \leq |p'[i] - q[i]|$ and $\exists i, 1 \leq i \leq d, |p[i] - q[i]| < |p'[i] - q[i]|$. We use $p \not>_q p'$ if p does not dominate p' with respect to q.*

Intuitively, $p >_q p'$ implies that p is at least as close to q as p' in all dimensions, and it is closer to q in at least one.

Definition 2 (Dynamic Skyline Query)**.** *Given a database D and a query point q, return a set $S \subseteq D$, such that no point in S is dominated by a point in D with respect to q, that is, $S = \{p \in D | \forall p' \in D, p' \not>_q p\}$.*

Note that, the conventional skyline query (where q is the domain origin) can be defined as a special case of dynamic skyline. Our focus is on the more challenging dynamic skyline setting, so whenever we mention skyline query, we refer to the dynamic case (given query q). We use Fig. 1 (a) as a running example: the skyline query answers for queries q_1 and q_2 are $\{p_3, p_4, p_5\}$, and $\{p_2, p_5\}$, respectively.

2.2 System Model

We assume three types of participants: the data owner (DO), the service provider (SP), and users. Users are trusted by the DO, and wish to obtain the result to dynamic skyline queries on the dataset owned by DO. The DO does not have the infrastructure to run such a service, so it outsources the functionality to SP (e.g., a commercial entity), which is *honest but curious*. The SP runs the protocol correctly, but may try to infer private details about the data. In addition, the SP may be compromised by an attacker, in which case the data may be exposed, with serious consequences (e.g., leakage of healthcare records). To address such threats, the DO first encrypts the dataset, and only shares the encrypted version with the SP. At runtime, the DO may be offline, and only the SP and the user engage in a protocol to determine the encrypted result to the user's skyline query q.

Users are trusted with some secret tokens (e.g., encryption keys), and are assumed not to collude with the SP (in practice, the users may be highly vetted individuals, e.g., medical doctors). The user retrieves a superset of the actual query result in encrypted form (i.e., including false positives), and performs a local *lightweight* filtering step to narrow down the exact result. Our solution *guarantees* that the user always obtains the exact result, and we also provide an upper bound on the total amount of false positives, in order to minimize the communication and computational cost on the user.

To support this protocol, the DO must prepare and encrypt the dataset, which may incur a significant overhead. However, this is a *one-time setup cost*. It helps reduce significantly the query processing overhead at runtime, which is the most important component of the cost, since that is the response time perceived by the user. We also assume that

Symbol	Definition
D, n, d	d-dimensional database D of cardinality n
$p >_q p'$	p dominates p' with respect to query q
$D^p_{p'}$	Domination region of p' with respect to p
S_p	Skyline region of p
$m_{p,p'}$	Mid-point between p and p'
$cnt(T)$, $spc(T)$	For $T = (S, P)$, $cnt(T) = P$, $spc(T) = S$
$\mathcal{T}_D$	Set of skyline tiles for database D
L_i	Boundaries of skyline tiles in dim. i
N_i, N	$N_i = \|L_i\|$, $N = \max_i\{N_i\}$
$\mathcal{I}$	Set of skyline indices
l	Number of skyline regions to intersect
k	Max. number of false positives allowed
m	Pre-partitioning parameter

Table 1: Summary of Notations

when the user registers for the service, there is a one-time setup cost on the user device. This may include transferring of a relatively small amount of metadata needed to run the protocol with the SP (our evaluation shows that the user download size is in the order of 10s of MB, which is a reasonable amount even on a mobile connection).

3. SKYLINE RESULT MATERIALIZATION

In this section, we introduce some preliminary concepts that are built upon later in Sections 4 and 5 to obtain efficient algorithms for building and storing an index on the plaintext data. Section 6 presents a complete, end-to-end processing algorithm on plaintext data. Finally, in Section 7 we show how the index is encrypted using a special transformation before being sent to the SP, and how traversal is performed on the encrypted structure.

3.1 Skyline Region of a Point

Consider a point $p \in D$. Recall that for a query q, p is in the skyline if p is not dominated by any other point in D with respect to q. Denoted by S_p is the *skyline region* of p, i.e., the set of all query points for which p is a skyline point: $S_p = \{x \in R^d | \forall p' \in D, p' \not>_x p\}$. Due to the properties of the domination relation, S_p is a polytope of a specific shape and can be constructed easily. We introduce several auxiliary concepts needed to define skyline regions.

Domination Region of a point. First, consider two points p and p'. Recall that $p' >_q p$, if and only if

$$\forall_{i,1\le i\le d} \; |p'[i] - q[i]| \le |p[i] - q[i]| \tag{1}$$

and

$$\exists_{i,1\le i\le d} \; |p[i] - q[i]| < |p[i] - q[i]| \tag{2}$$

Rephrasing the definition of domination, observe that p' dominates p for all the query points q in $D^p_{p'} = \{q \in R^d | q$ satisfies (1) and (2)$\}$. We refer to $D^p_{p'}$ as the domination region of p' with respect to p. For convenience, define $D^p_p = \emptyset$. The dominance region of all the points with respect to p_4 is shown in Fig. 1 (b)-(e).

Note that $D^p_{p'}$ is the solution to inequalities (1) and (2). We first focus on the solutions to Eq. (1). Observe that

$$|p'[i] - q[i]| \le |p[i] - q[i]| \iff \begin{cases} q[i] \ge \frac{p'[i]+p[i]}{2} & p'[i] > p[i] \\ q[i] \le \frac{p'[i]+p[i]}{2} & p'[i] < p[i] \\ \text{true} & \text{otherwise} \end{cases} \tag{3}$$

Given p and p' and for each i, (3) is an inequality of the form $q[i] \le c$ or $q[i] \ge c$, for some constant c depending on $p[i]$ and $p'[i]$. That is, the solution to the inequality for each i is of the form $(-\infty, c]$ or $[c, \infty)$. Let $m_{p',p}$ be the mid-point between p and p', i.e., $m_{p',p}[i] = \frac{p'[i]+p[i]}{2}$. Based on Eq. (3), a $q \in R^d$ satisfies Eq. (1) if

$$\forall_{i,1\le i\le d}, q[i] \in \begin{cases} [m_{p,p'}[i], \infty) & p'[i] > p[i] \\ (-\infty, m_{p,p'}[i]] & p'[i] < p[i] \\ (-\infty, \infty) & p'[i] = p[i] \end{cases} \tag{4}$$

Let $Z^p_{p'}$ be the set of q that satisfies Eq. (4). Observe that $Z^p_{p'}$ is a hyper-rectangle with its axes parallel to the coordinate axes, starting at $m_{p',p}$ and going to infinity or negative infinity in the direction of p'.

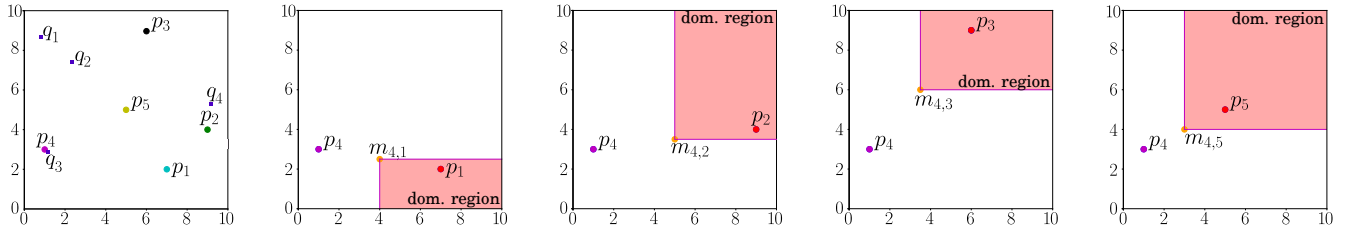
Finally, note that $D^p_{p'}$ is a subset of $Z^p_{p'}$ that also satisfies Eq. (2). The interior of $Z^p_{p'}$ always satisfies (2), but some points on the boundary of $Z^p_{p'}$ may not. E.g., $m_{p,p'}$ does not satisfy (2) by definition. Hence, $D^p_{p'}$ is the set Z except some of its boundaries. Since $Z^p_{p'}$ is defined by a set of hyper-planes, to define $D^p_{p'}$, we also use the set of hyper-planes, but in addition to its coordinates we store the possible exceptions in the boundaries. We formalize our notation of hyper-planes later in this section. Fig. 1(b)-(e) shows the domination regions of all points with respect to the p_4.

Skyline region of a point. Recall that S_p is the space where p is not dominated by any other point. Using the terminology above, S_p is the entire space except $\cup_{p'\in D} D^p_{p'}$, that is, $S_p = R^d \setminus (\cup_{p'\in D} D^p_{p'})$. This is because $\cup_{p'\in D} D^p_{p'}$ is the region where p is dominated by some point in D. S_p can be defined as a subset of R^d except the union of hyper-rectangles with their axes parallel to the coordinate axes.

Fig. 2(d) shows how we can find the skyline region of point p_4. We first find the domination regions of all points with respect to p_4. Then we take their union and the skyline region of p_4 is the entire space except the union of the domination regions. As Fig. 2(d) shows, not all points in D contribute to the skyline region of p_4.

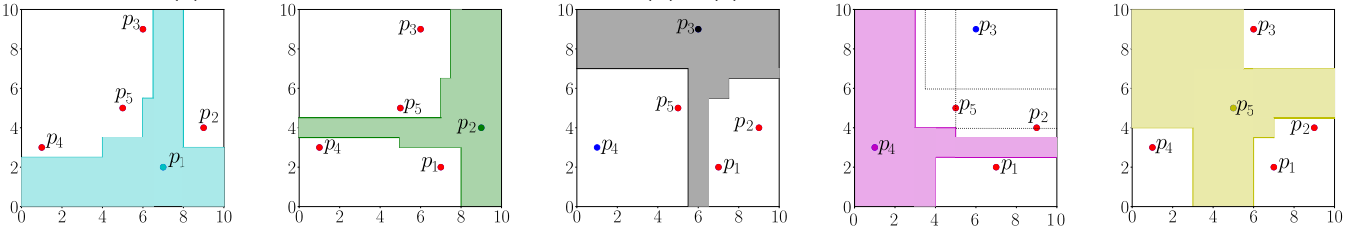
Skyline hyper-planes. The concepts defined so far consider subspaces of the query space defined by axis-parallel hyper-planes. In the rest of the paper, we refer by skyline hyper-plane (or simply hyper-plane) to the following data structure: for a hyper-plane H, the structure contains for each dimension i two points $min[i]$ and $max[i]$, representing the smallest (largest) value on the hyper-plane in the i-th dimension. For any hyper-plane, there exists a dimension i such that $min[i] = max[i]$. We say that a hyper-plane *is in* the i-th dimension if all the points on the hyper-plane have exactly the same value in that dimension. Furthermore, if the hyper-plane corresponds to the skyline region S_p, it stores the point p, and p is referred to as the hyper-plane's *generator*. In general, p is not a skyline point on the hyper-plane. However, as discussed before, a hyper-plane stores a set of exceptions, corresponding to coordinates (if any) on the hyper-plane for which p is actually a skyline point. Furthermore, we say that a hyper-plane is bounded by a set of hyper-planes H if every point on its boundary (defined by min and max) also belongs to another hyper-plane in H.

Border points. Not all the points in D contribute to $\cup_{p'\in D} D^p_{p'}$. That is, for two points, $p'_1, p'_2 \in D$, $D^p_{p'_1}$ may be a subset of $D^p_{p'_2}$, in which scenario $D^p_{p'_1}$ does not impact S_p. We refer to all the points such that $\forall p'_2 \in D, D^p_{p'_1} \not\subseteq D^p_{p'_2}$ as



(a) A database and 4 queries (b) Dom. region of p_1 w.r.t p_4 (c) Dom. region of p_2 w.r.t p_4 (d) Dom. region of p_3 w.r.t p_4 (e) Dom. region of p_5 w.r.t p_4

Figure 1: (a) A database and 4 sample queries. (b) - (e) Domination regions of all the points w.r.t p_4



(a) Skyline region of p_1 (b) Skyline region of p_2 (c) Skyline region of p_3 (d) Skyline region of p_4 (e) Skyline region of p_5

Figure 2: Skyline regions of all points

border points of p. Fig. 2 (a)-(e) shows the skyline regions of all points in the database. In each figure, the points in red are the border points and the points in blue are non-border points. The importance of border points is that they define how complex the skyline region of a point is. That is, the more number of border points, the more edges the skyline region will have. The following property helps us understand how many border points any point has.

Property 1. *p divides the space into 2^d quadrants. Let X_i contain the points in $D \setminus \{p\}$ that are in the i-th quadrant. p' is a border point of p if, for some i, p' is a skyline point with respect to a query at p for the database X_i.*

3.2 Tiling of the Space

3.2.1 Skyline Tiles

Observe that for a query q we can tell that p is in the skyline iff $q \in S_p$. The answer S to the skyline query q is $S = \{p \in D | q \in S_p\}$. We need to find an efficient way to check whether q is in S_p for all points p in D. To that end, we *intersect* S_p for all p with each other.

First, define a *partitioning* of a space Q as a set Π, such that for each $\pi \in \Pi$, $\pi \subseteq Q$, $\cup_{\pi \in \Pi} \pi = Q$ and that $\pi_i \cap \pi_j = \emptyset$ for any $\pi_i, \pi_j \in \Pi$, $\pi_i \neq \pi_j$. Let H_p be the set of hyper-planes defining the skyline region, S_p, for a given p. Now consider the set $H = \cup_{p \in D} H_p$. Observe that H_p is a set of intersecting hyper-planes where each hyper-plane is bounded by other hyper-planes. Consider the set of all polytopes created by the intersection of hyper-planes in H_p. Each polytope creates a partition, and their union Π_D is a partitioning of the space. For the *exceptions* stored in each hyper-plane, we also consider them to be a partition in the partitioning Π_D (For ease of discussion, in the remainder of this paper, we do not explicitly mention the partitions created by the exceptions as their shape is different from the polytope partitions, but the discussion either directly holds for both or the extension to the exceptions is straight forward). We use this partitioning to define *skyline tiles*. Formally, a *tile* is defined as follows.

Definition 3 (Tiles). *A tile, T, is defined as a tuple $T = (S, P)$, where S is a subspace of R^d defined by a set of hyper-planes and P is a subset of D. S satisfies the following conditions. Firstly, each hyper-plane of S is parallel to one of the axes. Secondly, each hyper-plane is bounded by another hyper-plane on all sides.*

For a tile $T = (S, P)$, S is called the *location* of T and P is called the *content* of T, also written as $cnt(T) = P$. Moreover, define *space* of T, written as $spc(T)$, as the subset of R^d that is inside S. We say a query *falls* inside T if $q \in spc(T)$. Finally, a *tiling of the space* is a set of tiles, X, such that $\cup_{T \in X} spc(T) = R^d$ and $spc(T) \cap spc(T') = \emptyset$ for all $T, T' \in X$.

For each partition $\pi_i \in \Pi_D$, a tile is τ_i defined as follows. Let $P_{\tau_i} = \{p \in D | \pi_i \cap S_p \neq \emptyset\}$ (i.e., the set of points whose skyline region intersects π). Let tile $\tau_i = (\pi_i, P_{\tau_i})$. We call τ a skyline tile and the set $\mathcal{T}_D = \{\tau_i, \forall i\}$ the set of skyline tiles of D. Skyline tiles have the following properties.

Property 2. *For a query q that falls inside a skyline tile T, the answer to the skyline query at q is $cnt(T)$.*

Property 3. *Consider two skyline tiles T_1 and T_2 such that both have a hyper-plane, H, as one of their edges. Assume hyper-plane H correspond to some point p. Then, either $p \in cnt(T_1)$ or $p \in cnt(T_2)$ but not both. Furthermore, $cnt(T_2) \setminus \{p\} = cnt(T_1) \setminus \{p\}$. In other words, content of T_1 and T_2 differ in exactly one point, p.*

Fig. 3 shows the skyline tiles created from intersecting all the skyline regions in Fig. 2. In Fig. 3, observe that t_1 is a tile defined by four hyper-planes (or lines, since $d = 2$), and its content is the set of points $\{p_3, p_4, p_5\}$. Now consider queries q_1 and q_2 in Fig. 3. Both queries fall in the tile t_1 and therefore their answer is p_3, p_4 and p_5. The query q_3 is in tile t_2 and its answer is p_4; whereas the answer to q_4 is $\{p_2, p_4\}$. This shows how an answer to a query can easily be retrieved by finding which tile the query falls into.

Border Locations. Recall that skyline tiles are created based on a set of hyper-planes, H. Consider the set H_i of all the hyper-planes in H that are in the i-th dimension. Let the set $L_i = \{x | h \in H_i,\ x = h.min[i] = h.max[i]\}$ (note that $h.min[i] = h.max[i]$ holds because h is a hyper-plane

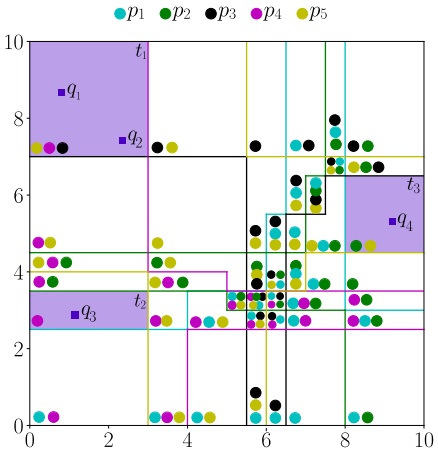


Figure 3: Skyline tiles

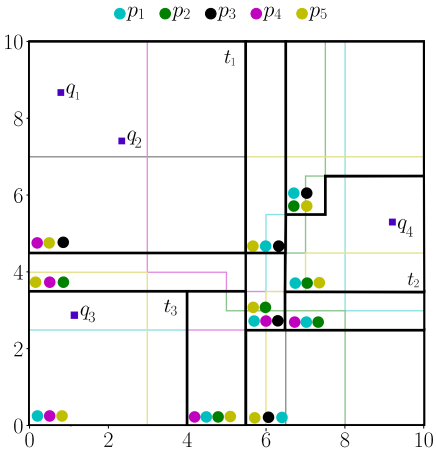


Figure 4: A solution to TAP

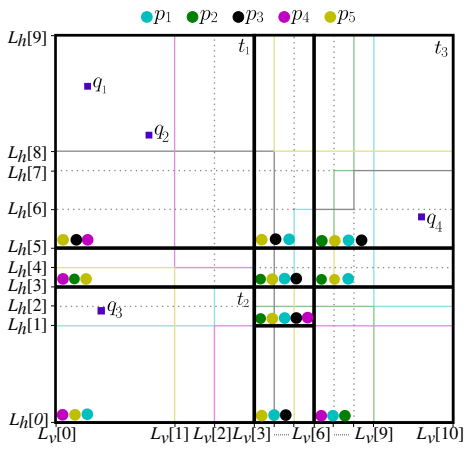


Figure 5: A solution to APP

in the i-th dimension). Observe that the set L_i contains the i-th dimension boundary of all the skyline tiles. Define $N_i = |L_i|$ for all i and let $N = \max_i L_i$. Observe that there is a relationship between number of border points and number of border locations. More specifically, every border point creates at most one hyper-plane in every dimension. Therefore, for every border point, we have a hyper-plane in H_i that corresponds to a location in L_i. Thus, we can study the value of N by analyzing the number of border points. This analysis is used in studying the performance of our algorithms.

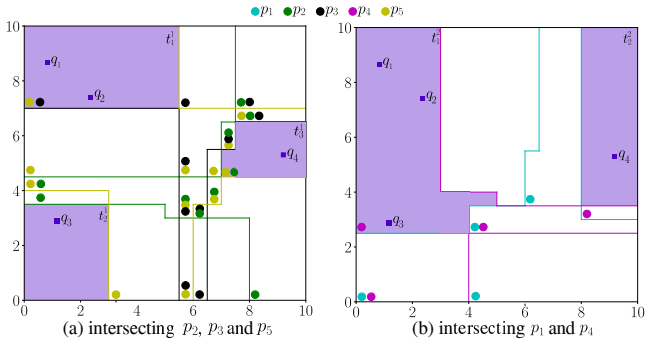
3.2.2 *Analysis*

Two important properties of skyline tiles that are utilized in our analysis are the value of N and the total number of tiles. They determine the space and time complexity of the algorithms discussed in the rest of the paper.

Number of tiles. Tiles are the intersection of n different skyline regions, and each region contains at most n hyper-planes in each dimension. For a hyper-plane in the first dimension, consider the maximum number of tiles it can be part of. In any other dimension, there can be at most $2^d \times n$ hyper-planes intersecting it . Thus, it can be a part of at most $2^{d(d-1)}n^{d-1}$ tiles. There are at most n^2 hyper-planes in the first dimension, and every tile must have one of those as its edge. Thus, there are at most $O(2^{d^2}n^{d+1})$ tiles in total.

Value of N. Let B_p be the set of border points for p. According to Property 1, $|B_p|$ is the size of a skyline query at p. Let $B_{avg} = \sum_{p \in D} \frac{|B_p|}{n}$. Observe that N is at most $\sum_{p \in D} |B_p|$. Thus, we can write $N = nB_{avg}$. If data points are uniformly distributed, there will be $O(\frac{n(2\log n)^d}{d!})$ number of skyline points [3] ($\frac{(\log n)^d}{d!}$ is the expected number of skyline points for uniformly distribution, there are n data points, and we need to consider skyline points for each of the 2^d quadrants created by p). Therefore, in such a scenario, $B_{avg} = O(\frac{n(2\log n)^d}{d!})$ on expectation.

Challenges. The number of tiles created by this approach is exponential in data size and makes the problem intractable. This occurs because skyline regions may impact parts of the space far from their generating point. Working directly with skyline tiles may be inefficient. We present two different methods in Sections 4 and 5 to deal with this issue.



(a) intersecting p_2, p_3 and p_5 (b) intersecting p_1 and p_4

Figure 6: Generalized tiling with $l = 3$.

4. AGGREGATING TILES

One approach to address the high space complexity of skyline tiles is to aggregate some of the tiles together. This saves spaces by storing the content of multiple tiles only once, but it may require the inclusion of false positives in the answer. Recent work [30] utilizes the computational power of the user to filter the final results based on the data returned to the user. Considering that the user can decrypt the data and operate on plaintexts, the amount of work required to filter out the false positives is very small. On the other hand, allowing a small amount of false positives can significantly increase storage and processing efficiency at the SP. In essence, we no longer partition the data domain so that the answer to each skyline query *is the same* within each tile. Instead, we partition the domain such that the answer to a query *does not change by much* within each partition. We achieve this by combining some of the tiles together. The tile aggregation problem is formalized as follows.

4.1 Tile Aggregation Problem

When false positives are permitted, a solution S^q returned by skyline processing algorithm for query q consists of two sets: set S_c^q which contains only (and all) points that are in the skyline of q, and the set S_f^q which contains none of the skyline points of q. We refer to the points in S_f^q as *false hits* and $|S_f^q|$ as the number of false hits for the query q.

For efficient post-processing, we bound the number of false hits for any query. We define the *false-hit requirement* as follows: let k be an integer; then the maximum number of false hits in a solution for any query must be at most k, that is $\max_q |S_f^q| \leq k$. Recall that, by allowing false-hits, we

aim at reducing the space complexity of our skyline result materialization structure. We reduce the space complexity by *aggregating* some of the skyline tiles. An aggregation is formally defined as follows: let $T = \{t_1 = (S_1, P_1), t_2 = (S_2, P_2), ..., t_r = (S_r, P_r)\}$ for the following definitions.

Definition 4 (Aggregation)**.** *An aggregation (or aggregate tile), A, of a set of tiles T is itself a tile $A = (S, P)$ that satisfies the following conditions. (1) $P = \cup_i P_i$ and (2) S is the smallest set such that $\cup_i S_i \subseteq S$. The tiles in T are called the component tiles of A.*

Definition 5 (Location-wise validity)**.** *An aggregation A of T is location-wise valid if the aggregation creates a single connected polytope.*

Intuitively, an aggregation is location-wise valid if all of its component tiles are next to each other, i.e., there is no empty space between them.

Definition 6 (Cardinality-wise validity)**.** *We say that an aggregation A of T is cardinality-wise valid if $|P| \leq k + \min_r |P_i|$, for a parameter k.*

That is, the aggregation contains at most k additional points compared to any of its component tiles. Observe that if an aggregation A is location-wise valid, any query q that falls inside the aggregation also falls inside one of its component tiles, T. If the aggregation is also cardinality-wise valid, we can return the content of A instead of T to answer the query q, and the solution will satisfy the false-hit requirement. Thus, we aim at finding aggregations that are both location-wise and cardinality-wise valid. Furthermore, we aim to reduce the total space used by the algorithm. We can express this as an optimization problem as follows.

Definition 7 (Tile Aggregation Problem)**.** *Given a set of tiles T and an integer k, return set $S \subseteq 2^T$ such that $\cup_{t \in S} = T$, $\forall t_i \in S$ the aggregation a_i of t_i is both location-wise and cardinality-wise valid, and $\sum_i |cnt(a_i)|$ is minimized.*

Fig. 4 shows a feasible solution to the Tile Aggregation Problem (TAP) when $k = 2$. Observe that the content of each aggregated tile is the union of the points in each of the component tiles. Furthermore, for queries q_1 and q_2, the answer is the same as the answer with no aggregation (see Fig. 3). However, q_3 now returns two false hits, i.e., p_1 and p_5 while q_4 returns one false hit, i.e., p_1.

The tile aggregation problem (TAP) as defined above is difficult to solve optimally. Specifically, we show that it is NP-hard even in two dimensions.

Theorem 1. *TAP is NP-hard.*

Proof. See Appendix A.

Two issues arise when solving TAP. First, solving TAP optimally is NP-hard. Second, the aggregate tiles of the TAP solution can have complicated shapes, slowing down the process of searching them. We address these issues next.

4.2 Relaxed Aggregations

We propose a relaxation of the aggregation problem. We restrict the possible choices by enforcing that aggregations must have a certain shape. However, we allow aggregations to split existing tiles into two (that is, half of a tile may belong to one aggregations and the other half to another) to avoid over-restrictive requirements. These modifications make it more intuitive to formulate the problem as a space partitioning problem, as discussed below.

Note that, any feasible solution, S, to TAP corresponds to a set of aggregations A whose union of space, i.e., $\cup_{a \in A} spc(a)$, covers the entire data domain. On the other hand, if we allow splitting of the tiles during aggregation, the observation here is that in fact *any* partitioning of the space, can be used to create an aggregation that in turn can be used to answer the skyline query. Consider a partitioning of the query space $\Pi = \{\pi_1, \pi_2, ..., \pi_r\}$. Define a set of tile $T = \{t_1, ..., .t_r\}$ such that $spc(t_i) = \pi_i$. Moreover, for a tile t_i, let $B_i = \{\tau \in \mathcal{T}_D | spc(\tau) \cap spc(t_i) \neq \emptyset\}$ and set $cnt(t_i) = \cup_{\tau \in B_i} cnt(\tau)$. Note that the set T defines a tiling of the space that covers the entire domain and if a query falls into a tile, the content of the tile will be a super-set of the answer to the query.

Since every partitioning of the query space corresponds to a tiling by the construction above, we formulate this relaxation of TAP (this is a relaxation because we now allow splitting of tiles), in terms of a space partitioning problem that minimizes the storage cost. Finding a space partitioning in the general case is also difficult, since, many aggregations are also space partitioning. However, we reduce the complexity of the problem by restricting the shape of a partitioning allowed.

Definition 8 (Shape-wise validity)**.** *A partitioning is shape-wise valid if it can be represented by a set of hyper-planes where a hyper-plane in the i-th dimension has boundaries from $-\infty$ to ∞ in all dimensions j when $j > i$.*

Intuitively, the partitioning contains hyper-planes in dimension i that are not bounded by any hyper-plane in dimensions after i.

Definition 9 (Aggregation by Partitioning Problem)**.** *For the query space Q, find a set of hyper-planes that define a shape-wise valid partitioning of Q such that the tiling, T, corresponding to the partitioning is cardinality-wise valid and that $\sum_{t \in T} |cnt(t)|$ is minimized.*

We provide a dynamic programming solution that solves Aggregation by Partitioning Problem (APP) optimally and then discuss a number of heuristics that solve it without approximation guarantees. It is worth mentioning that the quality of our solution to APP (i.e., whether it is optimal or not), helps with reducing storage cost of our index. However, our guarantees regarding properties of tiles (e.g., the false-hit requirement) hold true irrespective of whether the problem is solved optimally or not. Thus, heuristics can be useful in practice when using minimal space is not critical.

4.2.1 Dynamic Programming Solution to APP

We can solve APP optimally by dynamic programming. We limit our discussion to two-dimensions for ease of illustration. The idea can be extended to higher dimensions, and we can obtain a polynomial running time for any fixed dimensionality. However, in practice, the run time will become large for high dimensions, and using heuristics may be a better option in those scenarios.

The dynamic programming formulation uses two observations. First, the shape-wise validity requirement of APP in two dimensions implies the that the space partitioning is defined by a set of vertical lines that cross the entire space and

a set of horizontal lines that start and end between adjacent vertical lines. Second, there exists an optimal solution where all line overlaps the boundary of some skyline tile. This is because for any line that does not overlap the boundary of any skyline tile, we can move that line until it does overlap the boundary, and the total cost will not increase. Thus, to find the optimal solution, we only need to consider lines that pass through the boundaries of the skyline tiles.

Recurrence relation. Let L_v be an array of all possible x locations for the vertical lines and let L_h be an array of all possible y locations for the horizontal lines, both sorted in ascending order. To obtain L_v and L_h, we enumerate all the skyline tiles and find the x and y values of their boundary lines and add them to L_v and L_h if they don't exist. Let $N_h = |L_h|$ and $N_v = |L_v|$. Furthermore, consider $R = ((x_1, y_1), (x_2, y_2))$ as a rectangle with lower left coordinates (x_1, y_1) and upper right coordinates (x_2, y_2). Define $C(s, i, t, j)$ as the size of the content of the aggregate tile whose space is $R = ((L_v[s], L_h[i]), (L_v[t], L_h[j]))$ or infinity if such an aggregation is not cardinality-wise valid. That is, let $B = \{\tau \in \mathcal{T}_D | spc(\tau) \cap R \neq \emptyset\}$. Define $smallest = \min_{\tau \in B} |cnt(\tau)|$ and $size = |\cup_{\tau \in B} cnt(\tau)|$. Then let

$$C(s,i,t,j) = \begin{cases} size & smallest + k \leq size \\ \infty & \text{otherwise} \end{cases}$$

Define $V(i)$ as the optimal solution to the problem of APP in the space where $x > L_v[i]$, given that there exists a vertical line at $L_v[i]$. Note that the optimal solution to our problem is $V(0)$. Furthermore, define $H(i; s, t)$ as the optimal solution to APP in the space $L_v[s] < x < L_v[t]$, $y > L_h[i]$ with only horizontal lines given that there are vertical lines at $L_v[s]$ and $L_v[t]$ and a horizontal line at $L_h[i]$. Then, We can write the following recurrence relations.

$$V(i) = \min_{i<j\leq N_v} H(0; i, j) + V(j) \tag{5}$$

$$H(i; s, t) = \min_{i<j\leq N_h} C(s, i, t, j) + H(j; s, t) \tag{6}$$

with the base cases $H(N_h; s, t) = 0$ for all s, t and $V(N_v) = 0$. Intuitively, the recurrence relation tries to find the next vertical line after a given location and breaks down the problem into two separate instances of smaller size: (1) from the current line to the next and (2) from the next line to the end of the space. Doing so is possible because the vertical lines go through the entire space, that is the instance of the problem before a vertical line is independent of the instance after the vertical line. Furthermore, instance (1) contains only horizontal lines, therefore, we can solve it recursively by only considering all possible horizontal lines.

Algorithm. Algorithm 1 implements the recurrence relation. The algorithm starts backwards and tabulates the values for H and V. Note that one optimization compared with the recurrence relation is that distance between i and j is bounded by v_{max} (and similarly s and t by h_{max}). This is because of the false-hit requirement of the solution. That is, v_{max} is the maximum possible integer such that the space from $L_v[v_{max}]$ to $L_v[v_{max} + i]$ has a feasible solution using only horizontal lines (i.e., a partitioning exists that satisfies the false-hit requirement) for all $i \leq |L_v| - v_{max}$.

Correctness. Observe that, given that a vertical line exists at $L_v[i]$, the optimal solution must contain a next vertical line at a location $L_v[j]$, for some $j > i$ (except for the base case). Each j splits the space into two: (1) the space from $L_v[i]$ to $L_v[j]$; and (2) the space after $L_v[j]$. Note that, no partition is allowed to cross $L_v[j]$ because the partitioning has to be shape-wise valid. As a result, the two instances can be solved independently. Consider instance (1): it is the problem of finding the optimal tiling of the space between $L_v[i]$ and $L_v[j]$ using only horizontal lines, given that there are vertical lines at $L_v[i]$ and $L_v[j]$. This is the same as $H(0; i, j)$. Consider instance (2): it is the problem of finding the optimal tiling of the space after $L_v[j]$ given that there is a vertical line at $L_v[j]$. This is the same as $V(j)$. Therefore, for each j, the minimum possible cost is $H(0; i, j) + V(j)$. Since the optimal solution has to contain one of the possible values of j, then its cost must be $\min_j H(0; i, j) + V(j)$. A similar argument proves the correctness of the recurrence relation for H.

Algorithm 1 DP($\mathcal{T}_D$)

```
1: L_h ← horizontal boundaries of T_D, sorted
2: L_v ← vertical boundaries of T_D, sorted
3: N_v ← |L_v| and N_h ← |L_h|
4: V(N_v) ← 0
5: for i ← N_v − 1 to 0 do
6:   V(i) ← ∞
7:   for j ← i + 1 to N_v and j − i ≤ v_max do
8:     H(N_h) ← 0
9:     for s ← N_h − 1 to 0 do
10:      H(s) ← ∞
11:      for t ← s + 1 to N_h and t − s ≤ h_max do
12:        c ← C(i, s, j, t) + H(t)
13:        if c ≤ H(s) then
14:          H(s) ← c
15:    c ← H(0) + V(j)
16:    if c ≤ V(i) then
17:      V(i) ← c
18: return V(0)
```

Time Complexity. As Algorithm 1 shows, there are four nested loops, and each of them take N_v, v_{max}, N_h and h_{max} respectively. $C(i, s, j, t)$ can be found in $\log n$ using Property 3 (we keep track of the content of the tiles, and whenever a hyper-plan is crossed, we use Property 3 to determine the content of the new tile encountered). $O(\log n)$ is needed to determine whether the differing point mentioned in Property 3 already exists in the previous tile or not. Let $N = \max\{N_v, N_h\}$. Thus, the total running time is $O(v_{max} h_{max} N^2 \log n)$ ($C(i, s, j, t)$ can also be pre-computed to avoid the $\log n$ factor at the expense of storage cost). v_{max} and h_{max} depend on k and on how the tiles are distributed. They are generally similar to k in value, since every skyline tile differs in exactly one point from any of its neighbors.

4.2.2 Heuristics to Reduce Computation Time

We present two heuristics that can be used either independently, or together with our dynamic programming approach, to improve the running time of our algorithm at the cost of losing optimality.

Prepartitioning. To reduce the time complexity of the algorithm, we first *pre-partition* the space. We can do this by placing a vertical and horizontal line at every every multiple of m coordinates, where m is a fixed parameter. This bounds v_{max} and h_{max} to m. We solve the problem for each partition created using DP. Each partition now has m possible vertical positions and m horizontal positions. The runtime of the algorithm is reduced to $O(N^2 m^2 \log n)$, where m can be used to trade-off optimality with run-time. The solution approaches the optimal as m increases.

Greedy. Another heuristic is to choose the lines greedily, that is, starting from the beginning and choosing the next vertical line as far away from the current position as possible, and repeating that for the horizontal dimension. This reduces the cost to $O(|\mathcal{T}_D|)$.

5. GENERALIZED SKYLINE TILES

Another way to reduce the storage overhead is to generalize the skyline tile concept, and to allow for a tunable parameter providing a trade-off between space complexity and query time. Recall that, to generate skyline tiles, we intersected the skyline region of *all* the points, which lead to the creation of a large number of tiles. In this section, we generalize the process by only intersection l of the skyline regions, for a parameter l which gives us $\lceil \frac{n}{l} \rceil$ different sets of tiles. This helps reducing the space complexity, because it reduces the fragmentation of tiles (fewer data points result in fewer skyline region intersections). However, it increases query time, since we need to search multiple sets of tiles to find the final answer to the query. In the extreme case when $l = n$, we obtain one set of tiles that intersects all the skyline regions. When $l = 1$ we do not intersect the skyline regions at all, but we use each skyline region individually to know whether a point is in the skyline or not.

Observe that, generalizations of Property 2 and Property 3 hold for generalized skyline tiles. For Property 2, observe that a skyline tile created from the intersection of a set of points $D_i \subseteq D$ contains the subset of the answer to the skyline query in D. In other words, the union of the tiles a skyline query falls into is the answer to the skyline query. Furthermore, note that the aggregation methods discussed in Section 4 can still be applied to each set of generalized tiles created. That is, we apply the method $\lceil \frac{n}{l} \rceil$ times. However, the total number of false-hits will now be $\lceil \frac{n}{l} \rceil \times k$, as k is the number of false hits per use of the aggregation method.

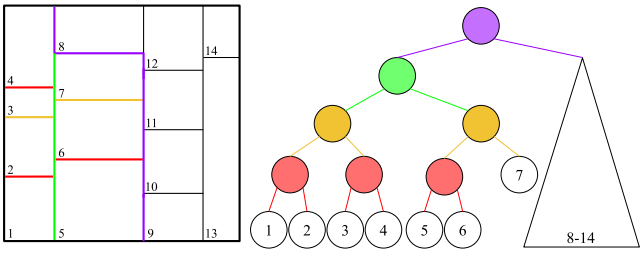
Fig. 6 illustrates the generalization concept. Setting $l = 3$, we get two different sets of tiles. Fig. 6(a) shows the intersection of skyline regions for p_2, p_4 and p_5, whereas Fig. 6(b) shows the intersection for the remaining skyline regions. Note that, a query has to search both sets of tiles. For instance, queries q_1 and q_2 fall into the tile t_1^1 from which we obtain points p_5 and p_3. They also fall into the tile t_1^2 from which we obtain p_4. Thus, the answer to q_1 and q_2 is p_3, p_4 and p_5. Observe that q_3 falls into t_2^1, but t_2^1 now does not contain any point at all (this is possible when the intersection is not done on all the skyline regions). q_3 also falls into t_1^2 which contains p_4. Thus, the answer to q_3 is p_4.

5.1 Analysis

Number of Tiles. Each index contains l skyline regions and each skyline region contains at most n hyperplanes in each of the d dimensions. For a hyperplane in the first dimension, consider the maximum number of tiles it can be a part of. There can be at most $2^d \times l$ hyper-planes intersecting it in any other dimension. Thus, it can be a part of at most $2^{d^2} l^{d-1}$ tiles. There are at most nl hyperplanes in the first dimension, and every tile has to have one of those as its edge. Thus, there are at most $O(2^{d^2} nl^d)$ tiles in total for each set of tiles.

Value of N. Let B_p be the set of border points for p. According to Property 1, $|B_p|$ is the size of a skyline query at p. Let $B_{avg} = \sum_p \frac{|B_p|}{l}$. Since N is at most $\sum_p |B_p|$, we can write $N = lB_{avg}$. If data points are uniformly distributed, there will be $O(\frac{n(2\log n)^d}{d!})$ skyline points. This is because $\frac{(\log n)^d}{d!}$ is the expected number of skyline points if the points are uniformly distributed, there are n data points, and we need to consider the number of skyline points for each of the 2^d quadrants created based on p. Therefore, in such a scenario, $B_{avg} = O(\frac{n(2\log n)^d}{d!})$ on expectation.

Remark. Observe that based on our choice of l, we can avoid both space and time complexity n^d (for instance, by setting $l = n^{\frac{1}{d}}$). Thus, generalized tiling is particularly useful as dimensionality increases.



(a) Solution to APP (b) Corresponding kd-tree

Figure 7: Building a kd-tree for a solution to APP

Algorithm 2 GetSkylineRegion(D, p)

```
1: B ← border points of p
2: S_p ← ∅
3: for p_i ∈ B do
4:    m ← (p + p_i)/2
5:    for dim ≤ d do
6:       h ← d-dimensional hyper-plane
7:       for dim' ≤ d do
8:          m ← (p[dim'] + p_i[dim'])/2
9:          end ← end point of hyper-plane
10:         h[dim] ← (m, end)
11:      S_p ← S_p ∪ {h}
12: return S_p
```

6. COMPLETE SKYLINE ALGORITHM

We described several methods to pre-compute the skyline result for any query within the data domain. Our approaches achieve various trade-offs between computation time, storage size and number of false hits. In this section, we show how the final query can be determined using plaintext data and returned to the user based on our pre-computed set of results. In the next section, we present how the data structures are encrypted, and how the query answering operations are performed using encrypted data.

The performance of our approach can be tuned using two main parameters: l and k. The former determines the number of partitions we divide our dataset into, and the latter the number of false positives allowed. Setting $k = 0$, the set of tiles will provide an exact answer, while in general there can be $\lceil \frac{n}{l} \rceil k$ number of false hits in the answer. Following the result pre-computation, we generate a set of indices, $\mathcal{I}$, containing $\lceil \frac{n}{l} \rceil$ separate index structures, used to answer the skyline query. We call $\mathcal{I}_i \in \mathcal{I}$ a skyline index.

According to our validity conditions (Sec. 4), $\mathcal{I}_i$ must partition the space recursively while disallowing overlaps between partitions. We employ a *kd-tree* index for this purpose. Alg. 3 illustrates the combined process of result pre-computation and index construction, with four stages:

Algorithm 3 GetSkylineIndices(D, l, k)

```
1: for p_i ∈ D do
2:    S_{p_i} ← GetSkylineRegion(D, p_i)
3: I ← ∅
4: for i ≤ ⌈n/l⌉ do
5:    H ← AggregateTiles(k, S_{p_{i×l}} ∪ S_{p_{i×l+1}} ... ∪ S_{p_{(i+1)×l}})
6:    I_i ← kd-tree on H
7:    for each leaf node n_l in I_i do
8:       n_l.points ← GetSkylinePoints(n_l)
9:    Add I_i to I
10: return I
```

1. Construction of Skyline Region. To create the skyline region of a point p, we first find its border points by issuing a skyline query at p on $D \setminus \{p\}$. Then, we iterate through all the border points, p' of p and find $D^p_{p'}$. $D^p_{p'}$ is defined in terms of a number of hyper-planes. Thus, we store these hyper-planes for each points for future use.

2. Aggregating Tiles. Line 5 of Alg. 3 uses one of the methods discussed in Sec. 4 to perform tile aggregation and returns the resulting hyper-planes. Note that, even when $k = 0$, we run an APP algorithm with $k = 0$ and obtain the corresponding hyper-planes. This approaches traverses the tiles once and splits some tiles into two, but constructs hyper-planes that can be easily used to create a balanced indexed (see below). Alternatively, when $k = 0$, we can skip this step but the process of creating a balanced tree become more complicated.

3. Building kd-tree. We build a balanced kd-tree from our tiles, as follows. All the hyper-planes are given in advance by our solution to APP, so we build a balanced kd-tree. Since for a solution to APP, the hyper-planes in the i-th dimension do not cross hyper-planes in the j-th dimension for $j < i$, we impose the following ordering on tiles by utilizing the partitioning. We traverse the tiles by going through the hyper-planes in the first dimension iteratively, and for each hyper-plane, recursively going through the hyper-planes in the next dimensions that fall right before it. Fig. 7 (a) shows how we can do this in two dimensions. We start from the left-most vertical line, and go through the tile in the ascending order of the horizontal lines. Then we move on to the next vertical line. This gives us the ordering shown in Fig. 7(a), where the numbers show the position of each tile in that order.

Then, to build the tree, we choose the split points so that half of the tiles are stored in one sub-tree and the other half in another. For instance, in Fig. 7, tiles 1-7 are in the left sub-tree and tiles 8-14 are in the right sub-tree. Note that the condition for the split can be define by $2 \times d - 1$ number of comparisons at each node. For instance, the condition for splitting at the root in Fig. 7(b) is shown by the purple lines in Fig. 7(a). It shows how two vertical lines and one horizontal line is enough to separate tiles 1-7 from tiles 8-14. This process is repeated recursively (each coloured line in Fig. 7 corresponds to the condition for a node with the same colour). Observe that the leaf nodes in the final tree created will have heights that differ by at most one. strictly speaking, the created index is not exactly a kd-tree, as the splitting conditions are different than a typical kd-tree.

4. Assigning Skyline Points. For each leaf node of the kd-tree we assign the corresponding skyline points. The content of each leaf node is already determined by running APP. Thus, we traverse all the leaf nodes and respectively copy the content from the corresponding aggregation.

Performing Queries. At runtime, the query result is determined by a simple traversal of the kd-tree index. The search locates the leaf node that encloses the query, and the list of points stored in that leaf represents the (super)set of the skyline query. In case of generalized tiles, the process is run separately for each index structure (i.e., $\lceil n/l \rceil$ times). All searches are completely independent, so the search can be ran in parallel at the SP, thus improving response time.

6.1 Performance Analysis

Index Construction Time. Alg. 2 first finds the border points of p, which takes $O(n^2)$. Then, for each point, it finds the hyper-planes delimiting its skyline region, which takes $O(d^3n^3)$ (line 9 takes $O(dn^2)$ because for any candidate end-point, we need to check if another end-point covers it). Overall, Alg. 2 takes $O(d^3n^3)$. Alg. 3 calls Alg. 2 routine for all the points, which costs $O(d^2n^4)$. Then, for each l points, Alg. 3 builds a kd-tree. Observe that, the height of the kd-tree is $O(d \log N)$, and in total, there are $O(dN)$ hyper-planes. Thus, building the index costs $O(d^2 N \log N)$. Then, there are a total of N^d leaf nodes, and filling the content of each takes $O(n)$ which is in total $O(nN^d)$.

Query Time. Searching each index takes $O(d^2 \log N)$ (index height is at most N^d and each level requires $O(d)$ comparisons), for a query time of $O(\lceil \frac{n}{l} \rceil d^2 \log n)$.

Space Complexity. Every tile can contain $O(l)$ points, and there are $O(\frac{n}{l})$ separate index structures. Therefore, the total space complexity is $O(2^{d^2} n^2 l^{d-1})$. In general, we can observe that increasing l reduces query time but increases space complexity. Thus, l can be set depending on the space constraints that exist at the service provider.

7. ENCRYPTED SKYLINE SEARCH

Our result materialization approach reduces the skyline query to a simple index look-up. The benefits of our method become even more clear when performing skyline queries on encrypted data. We do not require any distance calculations at query time, as existing methods do. We only require value comparisons for traversing the index. Furthermore, these comparisons are not performed on the actual data points, but on index node extents. In addition, we bulk-load the indices, which hides any data distribution details, and makes the indexes fully balanced. These features allow us to utilize simple and efficient cryptograpic primitives, while at the same time providing strong security guarantees.

7.1 Encryption Method

We need to encrypt a set of skyline indexes $\mathcal{I}$. For each index, we must encrypt (1) the data points stored in the leaf set and (2) the index structure itself.

Encrypting Data Points. The search does not perform comparisons on data points, so we can use conventional symmetric encryption, such as AES, which provides strong protection and also achieve semnatic security. After traversing the index and reaching the leaf level, we return the entire contents of the leaf to the user, who decrypts them locally.

Encrypting Index Structures. Since a kd-tree is used, we only need to perform comparisons at each index level. We employ two alternative encryption techniques: mutable order-preserving encryption (mOPE) [26] and practical

order revealing encryption *(pORE)* [7]. mOPE has been proven to be ideal, and does not leak any information about values, (e.g, no value distribution, density, etc.). However, it requires the encoding for each index value to be determined in advance, and the user needs to be aware of this mapping. As a result, the user must perform a one-time setup operation through which it downloads the mapping from the DO. On the other hand, with pORE, the user can compute the ciphertext of an arbitrary data value based on the secret key alone, without any mapping tables. However, pORE incurs a small and measurable leakage in the form of the position of the most significant bit that differs when comparing two values. Recall that, the comparison is not performed directly on data points, but on intermediate index values. Still, this amount of leakage may not be acceptable in some scenarios. Therefore, our two solutions offer a measurable trade-off between protection and setup cost: if the user is willing to download the mapping locally, then the ideal mOPE can be used. Otherwise, if the user is willing to trade a small amount of leakage, then pORE can be used, with no additional one-time setup required.

7.2 Security Analysis

We assume the clients do not collude with the server and that the server is honest-but-curious (i.e., it correctly follows the protocol, but tries to infer additional information). Our analysis quantifies the security leakage and answers the following questions: (1) what can the server learn about the data by just observing the skyline index (static data leakage), (2) what can the server learn about the data while performing encrypted queries on the index (dynamic data leakage) and (3) what can the server learn about a query when performing the encrypted query (query security).

Static Data Leakage. The SP observes the index and attempts to learn information from the structure. We need to show that, given a leakage function $\mathcal{L}_\mathcal{S}$, the server can only distinguish with negligible probability between a *real* index, R, created based on the original data and a *simulated* index, S, created based on the leakage function $\mathcal{L}_S$. The security game is to repeatedly show the server a pair (R_i, S_i) for a polynomial number of rounds and let the server guess whether the first index or the second index is the real one (the game is, in essence, similar to that of IND-OCPA [26]). The following theorem quantifies our leakage.

Theorem 2. *The server can distinguish between a real and simulated index with negligible probability given leakage function* $\mathcal{L}(\mathcal{I}) = (|\mathcal{I}|, \{\forall_{I\in\mathcal{I}|}\ (h_I, \sigma(I))\})$*, where* $|\mathcal{I}|$ *is the number of indices,* h_I *is the height of the index* I *and* $\sigma(I)$ *is the size of the content of each node in the index* I*.*

Proof sketch. Given the leakage function, and a fixed leaf set size, a simulated index can be constructed to have exactly the same structure. This results from our proposed index construction method, where we bulk-load the tree, and ensure that all leaves are at the same height. Thus, the security of our method boils down to the security of the underlying order-preserving encryption scheme. □

The above result is possible because we build a completely balanced tree, and only the order of magnitude for the leaf set is revealed. From an empirical perspective, the height of the index reveals very little about the data, as a wide range of data sizes lead to a skyline index structure of the same height. If leaking the leaf set size is considered unacceptable, one can employ padding, where fake points are added to the leaf nodes. Doing so will not have an effect on search performance, but increases communication cost.

Dynamic Data Leakage and Query Security. We study dynamic data security and query security together as they are both determined by the encryption method used for performing comparisons.

mOPE. If we use mOPE [26], we can guarantee that the only leakage from an individual query is its traversal path, and that there is no extra leakage (in addition to $\mathcal{L}_S$) from the data. This follows the security analysis of mOPE [26]. Intuitively, this holds because a query is translated into a traversal path of the index and then sent to the server. As a result, the server only learns the path accessed by the query and nothing more. At the same time, the server only accesses the index nodes based on the path provided by the query, which it could have done without the query as well.

pORE. Using pORE [7] increases the leakage of our algorithm but removes the extra storage requirement at the user side because of mOPE. Note that our encryption algorithm sends an independently encrypted value for comparison at each level of the index (that is, the encrypted query may contain the same dimension encrypted multiple times, and the size of the query is equal to the height of the tree).

Intuitively, this ensures that, at the server side, every comparison is independently done, and the server cannot learn anything by cross-examining the queries. Thus, the leakage is reduced to that of pORE [7], which is the index of the first bit that is different between the query and the index node. Since the server does not encrypt the query points, this leakage may not have any meaningful implication regarding the security of the data. However, it provides a lower level of guarantee compared with mOPE.

Protecting Traversal Patterns. We do not directly protect access patterns to the index. While access patterns themselves may not lead to an adversary learning the data values, they may reveal information about the query. To avoid such disclosure, our approach can be used in conjunction with oblivious RAM structures. There are a number of existing techniques [28, 29] that can be used in conjunction with generic index structures (including kd-trees) in order to hide access patterns through re-balancing. That will generate additional maintenance cost, although query response time will not be significantly affected. The details of combining ORAM with our approach are not specific to skyline queries, so they fall outside the scope of this submission.

8. EMPIRICAL EVALUATION

8.1 Experimental Setup

We performed experiments on an Intel i9-9980XE CPU (3GHz) with 128GB RAM running Ubuntu 18.04 LTS.

Dataset. Following the setup from [22], we use both synthetic and real datasets, but with larger sizes. We used the NBA[2] dataset with 2438 points in 5 dimensions, where each point represents an NBA player's performance metric (e.g., points scored, blocks, assists, etc.). We use three synthetic datasets: uniform, correlated (Gaussian distribution with correlation coefficient 0.9) and anti-correlated (Gaussian with coefficient -0.9). We consider up to 50,000 points, and dimensionality up to 5.

[2]Retrieved from https://stats.nba.com/ on on 04/15/2015

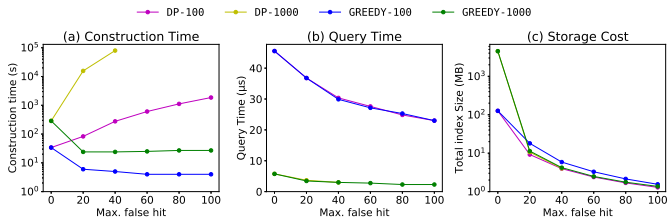


Figure 8: Varying false hits on smaller dataset

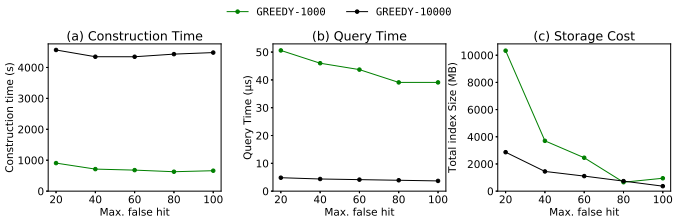


Figure 9: Varying false hit on larger dataset

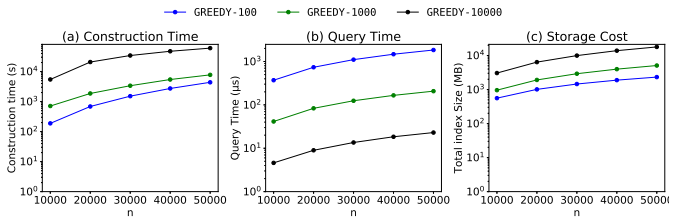


Figure 10: Varying data size

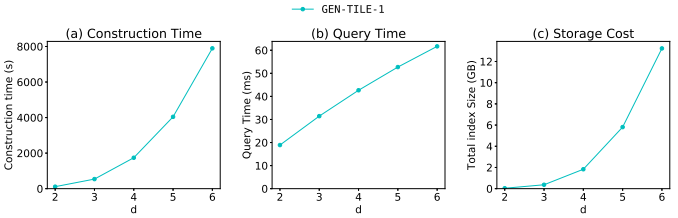


Figure 11: Varying dimensionality

Algorithms. We evaluate our dynamic programming algorithm (label `DP`) and greedy algorithm (label `GREEDY`) from Sec. 4, and for each of them we consider the generalized tiling option discussed in Sec. 5. The notation `GREEDY-l` (respectively `DP-l`) for some value of `l` refers to the greedy (respectively DP) algorithm with generalized tiling parameter set to l. We also include the standalone generalized tiling algorithm (without aggregation, i.e., skyline indexes are built directly on the skyline tiles) with label `GEN-TILE`.

To the best of our knowledge, the state-of-the-art work for skyline queries on encrypted data is that of [22] and [14]. The former requires two non-colluding servers, and it takes around three hours to complete a query, whereas the latter requires a time in the order of seconds for 100 data points. Since our method is much faster (sub-second query response time), we did not include a direct comparison with these approaches, which we clearly outperform – mainly due to our approach of materializing results.

Measurements. We report construction time, query time and storage cost. Construction time is the time required to build and encrypt the index structure(s) at the data owner. Query time is the time to execute a query on the encrypted index. Storage cost measures the amount required to store the SP. Result filtering time at the user takes less than half a second, so we omit it from the measurements.

8.2 Results on Uniform Data

Experiments on two-dimensional data. First, we compare the performance of the proposed DP and greedy algorithms for multiple settings of k and l. Since the DP approach is costly, we restrict these comparative runs to a dataset of $1,000$ records, and evaluate the greedy approach later on using larger data sizes. Fig. 8 summarizes our findings. Fig 8(c) shows that although `GREEDY` may not return optimal solutions, in practice, it returns solutions with storage cost very close to that of `DP`. However, Fig. 8(a) shows that `DP` takes much longer to run due, to its higher time complexity. Fig. 8(b) shows that the query time is almost the same regardless of whether `GREEDY` or `DP` is used. Observe that, the construction time is generally smaller for $l = 100$ compared with $l = 1000$, while query times are about a multiplicative factor apart ($l = 100$ requires 10 different indexes to be searched while $l = 1000$ only searches one index of larger size). We emphasize that the query time, which is the delay perceived by the user at runtime, is very small, always less than one millisecond.

The results show that `DP` provides a relatively small storage advantage compared with `GREEDY`, whereas it index construction time overhead is considerably larger. `DP` becomes impractical for larger values of k even when data set contains only $1,000$ points. In the remainder of the experiments, we exclude `DP` from our evaluation. However, it remains an interesting approach from a theoretical perspective, and may prove valuable in future work as a base for deriving effective heuristics that reduce storage cost.

Next, we increase data size to $n = 10,000$ and vary parameter k (which controls the maximum number of false hits allowed). Results are shown in Fig. 9. In general, k does not impact significantly construction time, and only has a visible impact on query time when $l = 1,000$. This is due to the fact that allowing more false hits reduces more significantly the number of tiles created when $l = 1,000$, compared with the case when $l = 10,000$. We suspect this occurs due to k being smaller (ranges from 2 to 10) when l is $1,000$ and increasing it allows for more flexibility of aggregations. Finally, we observed that index structure size is about 50MB for $k = 60$, which shows that the communication cost for transferring the structure when using mOPE is small.

We further increase data size up to $50,000$ data points and set $k = \frac{l}{100}$, so that the false hits count is $k \times \frac{n}{l} = \frac{n}{100}$, i.e., 1% of the data. Fig. 10 shows the results. For $l = 10,000$ the storage cost and construction time of the index become prohibitive, but the query time is at least an order of magnitude smaller than in other cases. However, smaller values of l can be used in practice to handle this workload.

Experiments on higher dimensional data. Observing the theoretical results of Sec. 5, we only use our generalized tiling approach with no aggregation in this section (i.e., the `Gen-Tile` algorithm). Our decision is due to the following factors: recall that in Sec. 5 we discussed decreasing the value of l for higher dimensional data to be able to overcome the curse of dimensionality. At the same time, the number of false hits, whenever $k > 0$, increases when decreasing l. Moreover, to be able to achieve construction time similar to that for 2D data when dimensionality increases, we need to decrease the value of l. As a consequence, we need to set l to a small value, but then setting $k > 0$ results in an unacceptably large number of false hits. Therefore, for high dimensional data, we suggest setting $k = 0$ and l to a small

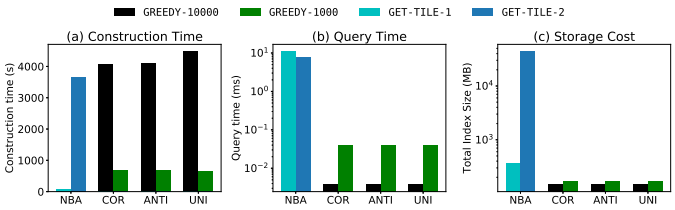


Figure 12: Other Distributions

value. That is, we perform no aggregation, but increase the number of skyline indexes. Thus, in this section, we only report results for `Gen-Tile`.

Fig. 11 shows the results when $n = 10,000$. We set $l = 1$ since for $d > 4$, larger values of l incur significantly more storage cost. Overall, the query time increases linearly with dimensionality. However, for higher dimensional data, storage cost surpasses 10GB and makes this approach less applicable for dimensionality higher than five.

8.3 Results on Non-Uniform and Real Data

We performed experiments on non-uniform data and real datasets as well. The results are shown in Fig. 8.3. The algorithms performs almost identical when comparing different synthetic distributions. This shows one significant difference between dynamic skyline queries and conventional skyline queries. Since in dynamic skyline queries the query point can be anywhere in the space, an anti-correlated distribution does not necessarily increase the size of the skyline result for a query (whereas in the traditional skyline, number of skyline results may change significantly by changing the data distribution). Finally, on the real NBA dataset, our algorithm can perform skyline queries within milliseconds, and when $l = 1$ only requires storage cost of about 300MB.

9. RELATED WORK

Plain-text Skyline Queries. The skyline query was first discussed in [17], and gained significant attention following the more recent work in [1]. Variations of the query under different scenarios have been extensively studied [8, 25, 27, 4, 19, 21, 31, 16]. The *dynamic* skyline query was formalized in [8], although general skyline algorithms [25] were able to answer dynamic skyline queries before that.

Closer to our work are algorithms focusing on continuous skyline queries for location-based services [18, 15, 20, 5]. These algorithms find ranges where the answer to the query does not change, and incrementally update the skyline when the answer does change. These algorithms exploit *spatio-temporal coherence*, which focuses on how the query and objects move over time to determine how the result of the skyline query evolves. This creates a fundamentally different problem, as the dominance relationship in our problem is defined differently, and there is no assumption on how queries change over time. As an example, observe that a data point far from the query (measured by Euclidean distance) will be *spatially dominated* [27] by closer points, but this is not necessarily the case with our dynamic skyline problem. In fact, this lack of *locality* is the main challenge in materializing dynamic skyline queries, as data points far from a query point can impact the result.

The work in [23] studied independently from us how to materialize the result of dynamic skyline queries. In [23], the space is partitioned into a grid, and grid cells with the same skyline result are merged. This leads to redundant time and space utilization, although the outcome is similar to our skyline tile concept discussed in Sec. 3. The authors of [23] proposed various methods for merging the grid cells to obtain the skyline tiles. Our method of finding skyline regions allows us to directly create *exact* skyline tiles. More importantly, [23] ignores the large storage cost of full materialization, which is the motivation for our contributions in Secs. 4 and 5. In contrast to our approach, [23] always performs full materialization, which is highly impractical due to the storage cost.

Secure Skyline Queries. The powerful trend towards outsourcing data storage and querying [11] led to a significant body of research on querying encrypted data. Most of this work focused on nearest-neighbor (NN) queries [12, 9, 13], culminating with the work in [30] which showed that the most secure and efficient way to answer NN queries on encrypted data is through materialization of results and encryption of the resulting structure. Our work follows a similar model, but we tackle the dynamic skyline query, for which result materialization is much more challenging.

The problem of securely answering skyline queries only received attention very recently. The work in [6] was one of the first to address outsourced skyline queries, but it only focused on authentication of results, *not* data confidentiality. Skyline queries on encrypted data were first considered in [24], where multiple parties engage in an interactive protocol to execute skyline queries on their joint datasets. Each party has access to its own data in plaintext (e.g., there are multiple data owners, and they all must be online at query time). This setting is considerably less challenging than ours. The work in [22] considers a model with two non-colluding servers that engage in a secure multi-party computation protocol to determine the result of the skyline query. The solution is slow, and the assumption that two non-colluding SP's agree to jointly offer a secure skyline service may not be feasible in practice. Finally, the recent work in [14] provides a single-SP solution based on fast secure permutations and comparisons. However, the solution relies on bilinear map pairings, which are notoriously expensive. The experimental results in [14] show performance numbers only up to 200 data points.

10. CONCLUSION

We proposed the first approach to use query result materialization for answering dynamic skyline queries on encrypted data. Compared to existing work on secure nearest-neighbors, the problem we tackle is much more complex, due to the fact that pre-computing skyline results lacks the locality property that allows NN solutions to be so efficient. We provided an in-depth theoretical analysis of skyline result materialization, and investigated extensively the trade-off that emerges between computational and storage costs. Our proposed heuristics are able to build the result materialization structure in reasonable time, while keeping the storage overhead at practical levels. Our ability to create balanced result materialization structures helps minimize the amount of leakage. In future work, we plan to study additional heuristics that can further reduce construction time. In addition, we will focus on the challenging problem of supporting incremental updates to skyline result materialization, so we can efficiently handle fast-changing datasets.

11. REFERENCES

[1] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings 17th international conference on data engineering*, pages 421–430. IEEE, 2001.

[2] S. Bothe, A. Cuzzocrea, P. Karras, and A. Vlachou. Skyline query processing over encrypted data: An attribute-order-preserving free approach. In *Proc. of Intl. Workshop Privacy and Security for Big Data*, pages 37–43, 2014.

[3] C. Buchta. On the average number of maxima in a set of vectors. *Information Processing Letters*, 33(2):63–65, 1989.

[4] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. K. Tung, and Z. Zhang. Finding k-dominant skylines in high dimensional space. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 503–514, 2006.

[5] M. A. Cheema, X. Lin, W. Zhang, and Y. Zhang. A safe zone based approach for monitoring moving skyline queries. In *Proceedings of the 16th international conference on extending database technology*, pages 275–286, 2013.

[6] W. Chen, M. Liu, R. Zhang, Y. Zhang, and S. Liu. Secure outsourced skyline query processing via untrusted cloud service providers. In *Proceedings of Annual IEEE Intl. Conf. on Computer Communications*, 2016.

[7] N. Chenette, K. Lewi, S. A. Weis, and D. J. Wu. Practical order-revealing encryption with limited leakage. In *International Conference on Fast Software Encryption*, pages 474–493. Springer, 2016.

[8] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *VLDB*, volume 7, pages 291–302, 2007.

[9] Y. Elmehdwi, B. K. Samanthula, and W. Jiang. Secure k-nearest neighbor query over encrypted data in outsourced environments. In *IEEE International Conference on Data Engineering (ICDE)*, pages 664–675, 2013.

[10] G. Ghinita, P. Kalnis, A. Khoshgozaran, C. Shahabi, and K. L. Tan. Private queries in location based services: Anonymizers are not necessary. In *Proceedings of International Conference on Management of Data (ACM SIGMOD)*, 2008.

[11] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 216–227, 06 2002.

[12] T. Hashem, L. Kulik, and R. Zhang. Privacy preserving group nearest neighbor queries. In *IEEE International Conference on Data Engineering (ICDE)*, pages 489–500, 01 2010.

[13] H. Hu, J. Xu, C. Ren, and B. Choi. Processing private queries over untrusted data cloud through privacy homomorphism. In *IEEE International Conference on Data Engineering (ICDE)*, pages 601–612, 2011.

[14] J. Hua, H. Zhu, F. Wang, X. Liu, R. Lu, H. Li, and Y. Zhang. Cinema: Efficient and privacy-preserving online medical primary diagnosis with skyline query. *IEEE Internet of Things*, 6(2):1450–1551, 2019.

[15] Z. Huang, H. Lu, B. C. Ooi, and A. K. Tung. Continuous skyline queries for moving objects. *IEEE transactions on knowledge and data engineering*, 18(12):1645–1658, 2006.

[16] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proc. of Very Large Data Bases*, pages 275–286, 2002.

[17] H.-T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM (JACM)*, 22(4):469–476, 1975.

[18] M.-W. Lee and S.-w. Hwang. Continuous skylining on volatile moving data. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1568–1575. IEEE, 2009.

[19] X. Lian and L. Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 213–226, 2008.

[20] X. Lin, J. Xu, and H. Hu. Range-based skyline queries in mobile environments. *IEEE Transactions on Knowledge and Data Engineering*, 25(4):835–849, 2011.

[21] J. Liu, L. Xiong, J. Pei, J. Luo, and H. Zhang. Finding pareto optimal groups: Group-based skyline. *Proceedings of the VLDB Endowment*, 8(13):2086–2097, 2015.

[22] J. Liu, J. Yang, L. Xiong, and J. Pei. Secure and efficient skyline queries on encrypted data. *IEEE Transactions on Knowledge and Data Engineering*, 31(7):1397–1411, 2018.

[23] J. Liu, J. Yang, L. Xiong, J. Pei, and J. Luo. Skyline diagram: Finding the voronoi counterpart for skyline queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 653–664. IEEE, 2018.

[24] X. Liu, R. Lu, J. Ma, L. Chen, and H. Bao. Efficient and privacy-preserving skyline computation framework across domains. *Future Generation Computer Systems*, 62:161–174, 2016.

[25] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 467–478, 2003.

[26] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *2013 IEEE Symposium on Security and Privacy*, pages 463–477. IEEE, 2013.

[27] M. Sharifzadeh and C. Shahabi. The spatial skyline queries. In *Proceedings of the 32nd international conference on Very large data bases*, pages 751–762. Citeseer, 2006.

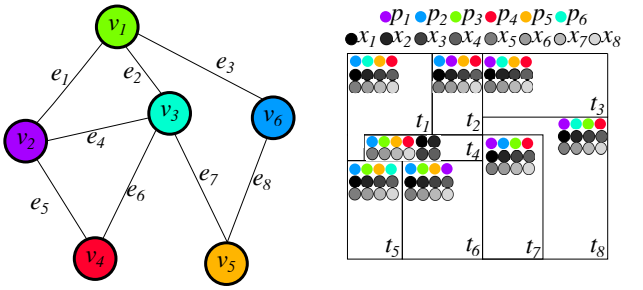
[28] E. Stefanov, M. V. Djik, E. Shi, T. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. *Journal of the ACM*, 65(4), April 2018.

[29] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious ram. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2012.

[30] B. Yao, F. Li, and X. Xiao. Secure nearest neighbor revisited. In *Proc. of Intl. Conf. on Data Engineering*,

pages 733–744, 2013.

[31] W. Yu, Z. Qin, J. Liu, L. Xiong, X. Chen, and H. Zhang. Fast algorithms for pareto optimal group-based skyline. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 417–426, 2017.



(a) A planar graph (b) Corresponding TAP instance

Figure 13: Reduction from PVC to TAP

APPENDIX

A. PROOFS

Proof of Theorem 1. We provide a polynomial time reduction from an instance of planar vertex cover to an instance of TAP.

Definition 10. *Planar Vertex Cover (PVC) Given a planar graph $G = (E, V)$, find a set of vertices of minimum cordiality, $S \subseteq V$ such that every edge, $e \in E$, there exists a vertex $v \in S$ where e is incident on v.*

Recall that and instance of PVC, $\mathcal{I}_{PVC}$ is defined by a graph, G, where an instance of TAP, $\mathcal{I}_{TAP}$ is defined by a set of tiles T, a database D and an integer k such that the set the content of each tile in T is a subset of D. Thus, we describe a polynomial time algorithm that returns T and k given a graph G, such that by solving $\mathcal{I}_{TAP}$ optimally we can solve $\mathcal{I}_{PVC}$ optimally as well.

Our reduction works as follows. For each edge in E we create a corresponding tile in T (therefore $|T| = |E|$). For each vertex in V and each edge in e we create a corresponding data point (to be assigned to the tiles, therefore, $|D| = |E| + |V|$).

The reduction has two steps. First, we construct the tiles and then assign points to the tiles. The construction of the tiles is done so that, for any set edges, B, incident on a given vertex, the aggregation of the tiles corresponding to B is location-wise valid. We that such a set of tiles can be constructed in polynomial time below.

Lemma 1. *Given a planar graph $G = (E, V)$, we can construct, in polynomial time, a set of tiles such that for any set edges, B, incident on a given vertex, the aggregation of the tiles corresponding to B is location-wise valid*

Now consider allocating points to the tiles. Consider two sets of points, $D_V = \{p_1, p_2, ..., p_{|V|}\}$, where p_i is a point corresponding to the vertex v_i, and the set $D_E = \{x_1, x_2, ..., x_{|E|}\}$, where x_i is a point corresponding to the edge e_i. Let $D = D_E \cup D_V$. First, for each tile, we insert the entire set D_E as its content. Furthermore, for a tile t_i corresponding to the edge $e_i = (v_x, v_y)$ we insert the set $D_V \setminus \{p_x, p_y\}$ to its content. Therefore, each tile contains exactly $|D| - 2$ points. The intuition behind adding some points of D_V to each tile is to control which tile aggregations are cardinality-wise valid. The intuition behind adding D_E to all the tiles is simply to increase the size of the content of each tile. Doing this enforces TAP to selects the fewest number of aggregations and helps us translate TAP's objective (which is in terms of total number of poitns stored) to VPC's objective (which is in terms of total number of aggregations performed), shown below. Fig. 13 shows the reduction for an instance of VPC to an instance of TAP.

Finally set, $k = 1$. This enforces that aggregation of two tiles that do not correspond to incident edges is cardinality-wise invalid. This is because any such aggregation will always have exactly $|D|$ points, since it will contain $D_E \cup D_V \setminus \{p_x, p_y\} \cup D_V \setminus \{p_{x'}, p_{y'}\}$. Note that $\{p_{x'}, p_{y'}\} \subseteq D_V \setminus \{p_x, p_y\}$ and therefore, $D_V \setminus \{p_x, p_y\} \cup D_V \setminus \{p_{x'}, p_{y'}\}$ contains both $p_{x'}$ and $p_{y'}$, as well as $D_V \setminus \{p_{x'}, p_{y'}$, which implies it contains the entire D_V. Moreover, aggregation of two tiles corresponding to incident edges is cardinality-wise valid. This is because the aggregation contains $D_V \setminus \{p_x, p_y\} \cup D_V \setminus \{p_x, p_{y'}\}$, which is equal to $D_V \setminus \{p_x\}$.

Now consider any feasible solution S to the TAP problem. The solution contains aggregations of two types. Firstly, aggregations that contain more than one tile and aggregations that contain exactly one tile. For aggregations corresponding to exactly one tile, t_i, consider any one of the two end points of e_i, and let $C_1 = \cup\{v_x\}$. For aggregations with more than one tile, all the tiles have to be corresponding to edges incident on a particular vertex, v_i. Let $C_2 = \cup_x\{v_x\}$. Consider the set $C_S = C_1 \cup C_2$. Note that C is a vertex cover of G. This is because all tiles are part of some aggregation and the corresponding edges for each tile is covered by the vertex corresponding to the aggregation. In general, for any feasible solution S to TAP we denote the corresponding feasible solution to VPC as C_S. Note that we have $|S| = |C_S|$. Therefore, $|S^*| = |C_{S^*}| \geq |C^*|$.

Next, we show that $|S^*| \leq |S_{C^*}| = |C^*|$. Observe that for any feasible solution C to VPC, we can construct a feasible solution S to TAP by just taking, for each vertex in C, the aggregation of all tiles corresponding to edges incident on C. We denote by S_C the corresponding solution to TAP for a solution C to VPC. Let cost of S, denoted by $c(S)$ for a solution S to TAP be the value of the objective function for the solution S. First note that since S^* is optimal, $c(S^*) \leq c(S_{C^*})$. Note that $S^* = S_1^* \cup S_2^*$, where S_1^* contains aggregations that contains exactly one tile and S_2^* contains the rest of the aggregations. Then $c(S^*) = |S_1^*|(||D| - 2) + |S_2^*|(|D| - 1) = (|S_1^*| + |S_2^*|)(||D| - 1) - |S_1^*|$ and similarly $c(S_{C^*}) = (|S_1^{C^*}| + |S_2^{C^*}|)(||D| - 1) - |S_1^{C^*}|$. Now assume that $|S_{C^*}| < |S^*|$ or $(|S^*| - |S_{C^*}|) \geq 1$. Because $c(S^*) \leq c(S_{C^*})$, we have that

$$(|S_1^*| + |S_2^*|)(|D| - 1) - |S_1^*| \leq (|S_1^{C^*}| + |S_2^{C^*}|)(|D| - 1) - |S_1^{C^*}|$$
$$(|S^*|)(|D| - 1) - |S_1^*| \leq (|S_{C^*}|)(|D| - 1) - |S_1^{C^*}|$$

Therefore,

$$|S_1^*| - |S_1^{C^*}| \geq (|S^*|)(|D| - 1) - (|S_{C^*}|)(|D| - 1)$$
$$\geq (|D| - 1)$$

Which is a contradiction. Therefore, $|C^*| = |S_{C^*}| \geq |S^*|$. Finally, we get that $|C^*| = |S^*| = |C_{S^*}|$. Therefore, the VPC solution corresponding to S^* is an optimal solution to VPC.

□

Proof of Lemma 1. The construction works as follows. Construction has two steps. First step is assigning tiles. It works as follows. (1) Draw the planar graph on a plane with all edges as straight lines and draw a minimum bounding

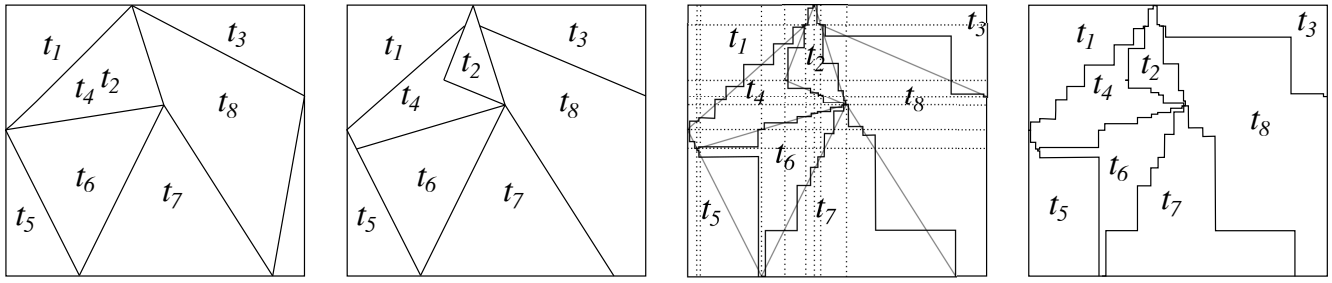


Figure 14: Construction of tiles

rectangle around the graph, (2) Arbitrarily assign each edge to one of the two faces of the graph the edge is a border of, (3) split faces with more than one corresponding edge, (4) if a face has no assigned edge, remove one of its edges and (5) move the boundaries. Step (1) is straight forward and can be done by Fry's theorem [X]. Step (2) is also straight forward, and each edge is a border of at most two faces because the graph is planar. For step (3), consider a point inside a face with more than one edge assigned to it, say edges e_i and e_j. Connect that point to the both ends of e_i (or e_j, the choice of the edge can be done arbitrarily), this creates exactly one new face (because the graph is planar). Assign e_i to this newly created face and remove it from its old assignment. Repeat this process until all faces have at most one point. For example, see t_2 in Fig. 14. For step (4) any bordering edge of a face with no points assigned can be removed. The resulting faces will have exactly one point left in them. For example, see t_8 in Fig. 14.

The property of this assignment is that for every vertex, there is a corresponding point in space where at least one edge of tiles corresponding to edges the vertex is incident on, ends at that point. This property helps us ensure that incident edges are neighbours. The second step is redrawing boundaries. Note that tiles need to have borders that are parallel to the x and y axes, while the current tiles do not satisfy this criterion. To do this, first, consider grid on the space whose vertical and horizontal lines pass through the vertices of the planar graph. Now, in each cell in this grid, a number of lines exist that potentially intersect at the corner of the cells and only at the corner. This is because the graph is planar, and the vertices can only be at the corner of the cells by construction.

Now in the cell, for each line consider its entrance point and its exit point from the cell. Our goal is to redraw the line such that for each cell, the line's entrance and exit points are the same, but it consists only of segments parallel to x and y axes. To do this, we first shift all the points whose entrance or exit is at a corner arbitrarily away from the corner and add a line segment from the corner to this new point. Now, for all the lines, we start perpendicular to their border and change direction right before, and enter the exit point perpendicular to the location as well. This way, we can ensure no two lines intersect unless they intersect at a corner because they correspond to incident edges.

Furthermore, observe that borders of all line segments corresponding to edges incident on a vertex intersect at the location of the vertex. Therefore, all the tiles ending at the vertex will be neighbours. However, for the tiles that shouldn't be, we again move their entrance point arbitrarily. Now, there is a path between tiles corresponding to incident edges.

□