An Index for Set Intersection with Post-Filtering

Ru Wang, Shangqi Lu, and Yufei Tao

Abstract—This paper studies how to design an index structure on a collection of sets $S_1, S_2, ..., S_n$ to answer the following queries: given distinct set ids $a, b \in [1, n]$, report $F(S_a \cap S_b)$ where $F(.)$ is a filtering function. We present a solution that can support a great variety of filtering functions — range research, skyline, convex hull, nearest neighbor search, quantile (to name just a few) — with attractive performance guarantees. The guarantees are sensitive to the set collection's *pseudoarboricity*, a new notion for quantifying the density of $\{S_1, S_2, ..., S_n\}$. Our index structures are simple to understand and implement.

1 Introduction

In the traditional *set intersection* problem, we want to build an index structure on a collection of sets $S_1, S_2, ..., S_n$ to answer the following query efficiently: given distinct set ids $a, b \in [1, n]$, report $S_a \cap S_b$. In practice, however, this "raw" intersection is rarely the final result demanded by a user, who instead often needs to apply certain post-processing on $S_a \cap S_b$. Next, we will illustrate this with some examples.

1.1 Motivation

Consider a web search scenario: *find the webpages containing keywords* `database` *and* `algorithm`. The query returns — in the information retrieval terminology — the intersection of `database`'s and `algorithm`'s inverted lists. Typically, the intersection encompasses thousands of pages, but only a limited few can be displayed in a user's browser. Hence, a query in reality also contains predicates like `page_rank` $\geq p$ (for some constant p) to select only the best pages. For a similar example in relational databases, consider the conjunctive query: *select * from* `people` *where* `job` = '`prof`' *and* `state` = '`CA`' *and* $x \leq$ `salary` $\leq y$ (for some constants x and y). Conceptually, the query applies a range condition on the `salary` values of the people that carry both of the labels `prof` and `CA`.

The previous examples apply post-filtering with a predicate supplied by the user at run time, but filtering may also be performed using a fixed function. An interesting example is the *skyline* function. Consider a classical scenario [11] where P is a set of hotels each having two attributes: `price` and (beach) `distance`. A hotel p_1 "dominates" another p_2 if p_1.`price` $\leq p_2$.`price` and p_1.`distance` $\leq p_2$.`distance`, with at least one inequality being strict. The "skyline" of P includes all the hotels in P not dominated by others. For example, if P consists of the 8 points in Figure 1, its skyline is $\{p_1, p_3, p_5\}$. The skyline is important because it captures the top-1 hotel under any scoring function monotone in `price` and `distance` [11].

In reality, hotel facilities are also an important factor in hotel selection. Define $P($`bar`$)$ as the set of hotels with bars, $P($`pet`$)$ as the set of pet-friendly hotels, and $P($`wifi`$)$ as the set of hotels offering free wifi. The query "*find the skyline of the hotels that have bars and free WiFi*" returns

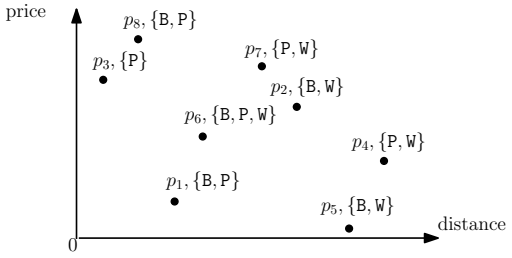


Fig. 1. Set intersection with skyline composition

the skyline of $P($`bar`$) \cap P($`wifi`$)$. In general, the skyline may vary significantly depending on the facilities requested. Figure 1 uses `B`, `P`, and `W` to denote `bar`, `pet`, and `wifi`, respectively (e.g., hotel p_1 has a bar and is pet-friendly). The skyline of $P($`bar`$) \cap P($`wifi`$)$ is $\{p_5, p_6\}$, the skyline of $P($`pet`$) \cap P($`wifi`$)$ is $\{p_4, p_6\}$, while that of $P($`bar`$) \cap P($`pet`$)$ is $\{p_1, p_8\}$. Note that these skylines may contain hotels outside the skyline of the complete hotel set P.

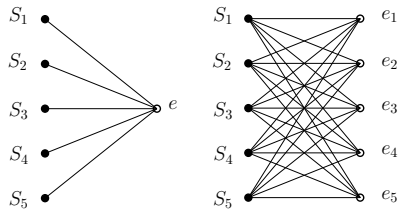
1.2 Our Contributions

Inspired by the above applications, we study how to index a set collection $S_1, S_2, ..., S_n$ to answer queries of the form: given distinct set ids $a, b \in [1, n]$, report $F(S_a \cap S_b)$ where $F(.)$ is a filtering function. We refer to the problem as *set intersection with post-filtering* (SIPF), which will be more formally defined in Section 2.

Our main contribution is an indexing scheme for tackling SIPF under a great variety of filtering functions. The scheme is simple to understand and implement but offers non-trivial performance guarantees. By resorting to a commonly-accepted conjecture on set intersection, we can even show that the proposed scheme no longer admits significant theoretical improvements.

To analyze our scheme's performance, we introduce a new concept called *set pseudoarboricity*. Traditionally, pseudoarboricity is a graph-theoretic notion used to quantify the density of a graph, but we adapt it to measure the density of a set collection $S_1, S_2, ..., S_n$. To gain intuition about how "set density" can be measured, observe a sense of symmetry between sets and elements: if a set S_i ($i \in [1, n]$) contains an element e, imagine *assigning* this relationship to either e or S_i. Assign all the set-element relationships in a manner as *balanced* as possible, i.e., each set/element gets roughly the same number of relationships. What is the largest number of

The authors are with The Chinese University of Hong Kong, Hong Kong. Their email addresses are {rwang21, sqlu, taoyf}@cse.cuhk.edu.hk. *This work was partially supported by GRF projects 14207820, 14203421, and 14222822.*



(a) Sparse (b) Dense

Fig. 2. Two extreme set collections

relationships assigned to a set or an element? This number is precisely the set collection's pseudoarboricity.

To further illustrate the above, consider a set collection where $S_1 = S_2 = ... = S_5 = \{e\}$, as pictorially shown in the graph of Figure 2(a). We can assign every edge (a.k.a. relationship) to its set vertex such that each set vertex gets assigned one edge (the element e is assigned none). Figure 2(b) illustrates another set collection where $S_1 = S_2 = ... = S_5 = \{e_1, e_2, ..., e_5\}$. As can be verified, any assignment must allocate at least 3 edges to a set or element vertex. The difference between the values 1 and 3 reflects the fact that the latter set collection is denser than the former.

If α is the pseudoarboricity of the input set collection, the indexes produced by our scheme use $O(\alpha N \operatorname{polylog} N)$ space (where $N = \sum_{i=1}^{n} |S_i|$) and answer a query in $O(\alpha \operatorname{polylog} N + \text{OUT})$ time, where OUT is the number of elements returned, namely, $\text{OUT} = |F(S_a \cap S_b)|$ (where a and b are the queried set IDs). Note that OUT can be far less than the size of $S_a \cap S_b$.

The value of α ranges between 1 and $O(\sqrt{N})$. For practical use, our structures are suitable mainly for sparse set collections with small pseudoarboricity values. We can show, however, that this is likely inevitable for the filtering functions considered. Specifically, unless the *strong set-intersection conjecture* [29] (see Section 3.1 for details) is wrong, both statements below are true.

- No structures of $O(\alpha N \operatorname{polylog} N)$ space can ensure $O(\alpha^{0.99} \operatorname{polylog} N + \text{OUT})$ query time.
- No structures of $O(\alpha \operatorname{polylog} N + \text{OUT})$ query time can use only $O(\alpha^{0.99} N \operatorname{polylog} N)$ space.

We also conducted an experimental evaluation to validate our theoretical findings on real-world data. Our results indicate that in practice, our structures perform better than what was predicted in theory, which is not surprising given the conservative nature of theoretical analysis. Additionally, we discovered useful heuristics from the experiments that can further optimize the performance of our structures in practical applications.

The paper is structured as follows: Section 2 formally defines the problems to be studied. Section 3 provides an overview of the existing research that is closely related to our work. In Section 4, we introduce the concept of "set pseudoarboricity" and prove its basic properties. Section 5 describes the core of our structure in the foundational scenario where no post-filtering is required on an intersection result. We then explain in Section 6 how to extend the core index to SIPF problems. Our experimental evaluation is presented in Section 7, and the paper concludes in Section 8 with a summary of our findings.

2 Problem Definitions

Let us first define the *set intersection with post-filtering* (SIPF) problem. $S_1, S_2, ..., S_n$ are n non-empty sets, where the elements are drawn from a common domain $\mathbb{D}$. A *query* reports $F(S_a \cap S_b)$ where

- a and b are distinct set ids from 1 to n;
- $F(.)$ is a *filtering function* that takes a set $D \subseteq \mathbb{D}$ as the parameter and returns a subset of D.

Our goal is to create an index structure that can answer all queries efficiently. Define $N = \sum_{i=1}^{n} |S_i|$, the problem's input size. We denote by $\mathcal{F}$ the set of all functions F that can be given at query time.

This paper will discuss several representative SIPF problems defined below.

Identity. We say that $F(.)$ is an *identity function* if $F(D) = D$ for any $D \subseteq \mathbb{D}$. When $\mathcal{F}$ includes only the identity function, SIPF degenerates into traditional set intersection.

Predicate-Based Filtering. In this case, $F(S)$ filters S with a predicate π, which maps each element $e \in \mathbb{D}$ to either true or false. Specifically:

$$F(D) = \{e \in D \mid \pi(e) = \text{true}\}.$$

Besides set IDs i and j, each query can select an arbitrary π from a class of predicates, thereby defining its own filtering function F.

Various predicate classes create different SIPF instances. Consider that each element $e \in \mathbb{D}$ is associated with a d-dimensional point $p_e \in \mathbb{R}^d$. To formulate π, a user selects an arbitrary axis-parallel rectangle q in $\mathbb{R}^d$ and decides $\pi(e) =$ true if and only if p_e falls in q. The set $\mathcal{F}$ includes all the filtering functions obtained this way. We refer to this SIPF instance as *SIPF-range*.

Alternatively, a user may formulate π by selecting a *halfspace* q in $\mathbb{R}^d$. Specifically, a halfspace includes all the points $p \in \mathbb{R}^d$ satisfying $\sum_{i=1}^{d} c_i \cdot p[i] \geq c_{d+1}$, where $p[i]$ is the i-th coordinate of p. The user decides $\pi(e) =$ true if and only if p_e falls in q. All the filtering functions thus constructed constitute $\mathcal{F}$. We call this SIPF instance *SIPF-halfspace*.

Reducible Filtering. In the second class of SIPF problems we consider, $\mathcal{F}$ consists of only a single filtering function $F(.)$, which is reducible. Specifically, $F(.)$ is *reducible* when it has the following property

$$\forall D' \subseteq D \subseteq \mathbb{D}, \text{if } F(D) \subseteq D', \text{then } F(D') = F(D). \quad (1)$$

Many filtering functions in database applications are reducible. To see this, consider, again, that every element $e \in \mathbb{D}$ is associated with a d-dimensional point $p_e \in \mathbb{R}^d$. The function

$$F(D) \quad = \quad \text{the skyline of } \{p_e \mid e \in D\} \quad (2)$$

is reducible.[1] Similarly, the function

$$F(D) = \text{the vertex set of the convex hull of } \{p_e \mid e \in D\} \quad (3)$$

1. Formally, the *skyline* of a set P of points in $\mathbb{R}^d$ contains every point $p \in P$ that is not dominated by any other point in P, namely, no point $p' \in P$ satisfies $p'[i] \leq p[i]$ for all $i \in [1, d]$ with the inequality being strict for at least one i.

is also reducible[2] We use the name *SIPF-skyline* for the instance where $\mathcal{F}$ contains a single function as given in (2), and *SIPF-CH* for the instance where $\mathcal{F}$ contains a single function as given in (3).

Filtering Functions Friendly to Deflation. Section 6.3 will present a technique we name *deflation filtering*, which enables us to devise efficient indexes for many SIPF problems. Next, we define two problems, SIPF-NN and SIPF-quantile, that will be used to illustrate the technique.

In *SIPF-NN*, each element $e \in \mathbb{D}$ is associated with a 2D point p_e. The set $\mathcal{F}$ contains a single filtering function:

$$F(D) = \text{the } k \text{ nearest neighbors of } q \text{ in } \{p_e \mid e \in D\} \quad (4)$$

namely, the k points in $\{p_e \mid e \in D\}$ closest to q. A user can freely select the integer $k \geq 1$ and the query point $q \in \mathbb{R}^2$.

In *SIPF-quantile*, each element $e \in \mathbb{D}$ carries a distinct real value p_e. The set $\mathcal{F}$ contains only one filtering function:

$$F(D) = \text{the } k\text{-quantiles of } \{p_e \mid e \in D\} \quad (5)$$

namely, the $\lfloor i \cdot |D|/k \rfloor$-th smallest value in $\{p_e \mid e \in D\}$ for $i = 1, 2, ..., k$; in the special case where $k > |D|$, $F(D)$ returns the entire D. The value of k is fixed in advance.

Remark. We aim to design a generic indexing scheme that works for all the above SIPF problems. As a baseline approach, one can resort to conventional set intersection: first retrieve $S_a \cap S_b$ and then compute $F(S_a \cap S_b)$. The primary concern with this approach is its failure to leverage the fact that the output size OUT can be significantly less than $|S_a \cap S_b|$ (recall that OUT counts how many elements in $S_a \cap S_b$ survive filtering). The worst query time is $\Omega(N)$.

Our indexing scheme is purposed for static data, i.e., no updates to the set collection $\{S_1, ..., S_n\}$ are permitted.

Math Conventions. Given an integer $x \geq 1$, we use $[x]$ to represent the set $\{1, 2, ..., x\}$. Denote by $\mathbb{Z}^+$ the set of positive integers. A function $f(.)$ from $\mathbb{Z}^+$ to $\mathbb{Z}^+$ is said to grow *at least linearly* if $f(x_1) + f(x_2) \leq f(x_1 + x_2) + O(1)$ for all $x_1, x_2 \in \mathbb{Z}^+$, where the constant factor in $O(1)$ does not depend on the selection of x_1 and x_2.

3 Related Work

Section 3.1 discusses the previous research most relevant to SIPF. Then, Section 3.2 reviews the notion of pseudoarboricity in graph theory.

3.1 Existing SIPF-Related Results

We will first survey the existing results on set intersection (where a query returns $S_a \cap S_b$), which, as mentioned, serves as a generic solution to SIPF indexing. Then, we will discuss the existing non-generic indexes for special SIPF instances.

Set Intersection. This fundamental problem has drawn considerable attention from the theory community. With perfect hashing, it is rudimentary to obtain an index of $O(N)$ space that returns $S_a \cap S_b$ (for any set IDs a and b) in $O(\min\{|S_a|, |S_b|\})$ time. Bille et al. [8] reduced the query time to $O(\text{OUT} + \min\{|S_a|, |S_b|\} \cdot (\log\log N)^2 / \log N)$, where $\text{OUT} = |S_a \cap S_b|$. Eppstein et al. [27] further improved the query time to $O(\text{OUT} + \min\{|S_a|, |S_b|\} \cdot \log\log N / \log N)$. In the worst case where both $|S_a|$ and $|S_b|$ are $\Omega(N)$, the query time of [27] becomes $O(N \log\log N / \log N + \text{OUT})$.

In [21], Cohen and Porat described a structure of $O(N)$ space that answers any (set-intersection) query in $O(\sqrt{N} + \sqrt{N \cdot \text{OUT}})$ time. Kopelowitz et al. [38] generalized the result by showing that, if $spc(N) \geq N$ space is allowed, there is a structure with query time $\tilde{O}(N/\sqrt{spc(N)} \cdot (1 + \sqrt{\text{OUT}}))$, where $\tilde{O}(.)$ hides a factor polylogarithmic to N.

Several commonly-accepted conjectures exist on the hardness of set intersection [29], [30], [39], [40], [48]. Most relevant to our work is the *strong set-intersection conjecture* [29], which states that any structure answering a query in $qry(N) + O(\text{OUT})$ time must use $\tilde{\Omega}(N^2/qry(N))$ space, where the notation $\tilde{\Omega}(.)$ hides a factor polylogarithmic to N. In fact, this conjecture has already been proved [2] on the class of pointer-machine structures[3]

The reader may refer to [5]–[7], [17], [24], [34] and the references therein for additional results that are relevant but dominated by those mentioned earlier.

The system community has studied extensively how to maximize the practical efficiency of set intersection. The past research has pursued two orthogonal directions: software vs. hardware. The software direction aims to discover effective implementation-level heuristics; see, for example, [12], [25], [32], [36], [42], [45], [49], [50], [53], [54], [57], [58], [61] for techniques related to compression, cacheline-aware programming, source code fine-tuning, etc. The hardware direction, on the other hand, aims to benefit from specialized computing devices, such as multi-core CPUs [12], [45], GPUs [4], [52], and SIMD processors [9], [33], [35], [37], [44], [60].

Special SIPF Problems. We are aware of no generic indexing solutions to SIPF (other than the approach that resorts to set intersection by ignoring filtering). Even for the special instances defined in Section 2, the only theoretical result we are aware of is a structure by Goodrich [31] for 1D SIPF-range. His structure uses $O(N)$ space and has $O(N(\log\log N)/\log N + \text{OUT})$ expected query time; note that OUT here can be much smaller than the size of $S_a \cap S_b$.

In the system community, the SIPF-range and SIPF-NN problems (Section 2) have been studied in the *keyword search* area; see representative works [3], [14]–[16], [18], [19], [28], [43], [46], [51], [55], [56], [59] and the references therein. The indexes in those works have robust performance on the problems they are designed for, but require substantial non-trivial modifications to work for other SIPF instances. Designing a generic indexing scheme to minimize such modifications is a primary goal of this paper.

3.2 Graph Pseudoarboricity

Let $G = (V, E)$ be an undirected simple graph. By giving a direction to each edge in E, we obtain a directed graph G^+, called an *orientation* of G. The *pseudoarboricity* of G is the smallest number α such that we can find an orientation of G where the maximum out-degree[4] is α. For example, the

2. Formally, the *convex hull* of a set P of points in $\mathbb{R}^d$ is the smallest convex polytope covering P.

3. In simple words, a pointer-machine structure is an index where navigation must be performed based on pointers. Most indexes in database systems, e.g., binary search trees, B-trees, R-trees, and so on, are pointer-based.

4. In a directed graph, an edge pointing from vertex u to vertex v is denoted as (u, v). Accordingly, v is an *out-neighbor* of u, and u is an *in-neighbor* of v. The *out-degree* of a vertex is the number of its out-neighbors.

(a) A graph G (b) The orientation G^+ output by **orient**

Fig. 3. Illustration of the **orient** algorithm

graph G in Figure 3(a) has pseudoarboricity 2, and Figure 3(b) shows an orientation of G with maximum out-degree 2.

Identifying an orientation of G whose maximum out-degree matches the pseudoarboricity α is computationally intensive: the fastest algorithm to our knowledge takes $O(|E|^{1.5}\sqrt{\log\log|V|})$ time [10]. However, for our theoretical results to hold, we can accommodate any orientation of G with maximum out-degree $O(\alpha)$, i.e., allowing a constant-factor approximation. Such relaxed orientations are much easier to find. One approach, for instance, is to apply the following algorithm developed by Matula and Beck [47]:

orient ($G = (V, E)$)
1. $E^+ = \emptyset$
2. **while** G not empty **do**
3. $u \leftarrow$ a vertex currently with the smallest degree
4. **for** each edge $\{u, v\}$ in E **do**
   add a directed edge (u, v) into E^+
5. remove u and its edges from G
6. **return** $G^+ = (V, E^+)$

Let us illustrate the algorithm on the graph G in Figure 3(a). In the first round, both vertices B and D have the smallest degree 2; either can be chosen as the vertex u at Line 2. Suppose that u is set to B, after which both edges of B are directed away from B (Line 4), as shown in Figure 3(b). Then, B and its edges are removed from G (Line 5), which now has only three edges $\{A, D\}, \{A, C\}$ and $\{C, D\}$. In the second round, all remaining vertices have the same degree 2. Suppose that u is set to A this time. Accordingly, the edges $\{A, D\}$ and $\{A, C\}$ are directed away from A; see Figure 3(b). The round ends by deleting A and its two edges. This leaves G with only one edge. Assuming that the third round sets u to C, we obtain the orientation G^+ in Figure 3(b).

The next lemma summarizes the algorithm's guarantees.

Lemma 1. *The* **orient** *algorithm can be implemented in $O(|V| + |E|)$ time and returns an orientation G^+ with maximum out-degree at most 2α, where α is the pseudoarboricity of G.*

Proof. The efficiency claim is due to Matula and Beck [47]. They also proved that the maximum out-degree of G^+ equals the so-called *degeneracy* of G. Eppstein [26] showed that, in general, the degeneracy of any undirected simple graph is at most twice the graph's pseudoarboricity. □

4 Pseudoarboricity of a Set Collection

We will extend the notion of pseudoarboricity to a collection of non-empty sets $S_1, S_2, ..., S_n$. Define $U = \bigcup_{i=1}^n S_i$ and, as before, $N = \sum_{i=1}^n |S_i|$. Note that $N \geq n$ because we do not allow empty sets.

Build a graph $G = (V, E)$ — the *companion graph* of the set collection — as follows. First, for every set S_i ($i \in [n]$), add a vertex to V, which we call a *set vertex* or *the vertex of S_i*. Second, for every element $e \in U$, add a vertex to V, which we call an *element vertex* or *the vertex of e*. To construct E, create an edge between an element vertex e and a set vertex S_i if and only if $e \in S_i$. This finishes the creation of G, which is a bipartite graph. Note that $|V| = n + |U|$ and $|E| = N$.

Definition 2. *The* **pseudoarboricity** *of a set collection is defined as the pseudoarboricity of its companion graph.*

Example 1. Figure 4(a) shows the companion graph G of the set collection S_1, S_2, S_3 in Example 1. The graph has pseudoarboricity 2. To see why, first note that the pseudoarboricity of G must be greater than 1 (there are 16 edges but only 11 vertices). Then, let us witness an orientation G^+ of G that has maximum out-degree 2. For clarity, we present the edges of G^+ in two parts: Figure 4(b) shows the edges oriented from left to right, while 4(c) shows the edges oriented from right to left. □

Let α be the pseudoarboricity of $S_1, S_2, ..., S_n$. Next, we give several properties of α.

Property 3. $\alpha = O(\sqrt{N})$.

Proof. In general, the pseudoarboricity of any undirected graph with m edges is bounded by $O(\sqrt{m})$ [20]. The property follows because the companion graph has N edges. □

Property 4. *The value α is at most the smaller value between*

- *the largest size of $S_1, ..., S_n$, and*
- *the largest number of sets having a non-empty intersection.*

Proof. We prove the property by considering only the scenario where the quantity of the first bullet is at most that of the second bullet (the opposite case is symmetric). Construct an orientation of the companion graph G by directing every edge from a set vertex to an element vertex. This orientation has a maximum out-degree equal to $\max_{i=1}^n |S_i|$ which, by definition of pseudoarboricity, must be at least α. □

To explain the next property, let G^+ be the orientation of G obtained from the **orient** algorithm in Section 3.2. For each set S_i ($i \in [n]$), define

$$S_i^- = \{e \in S_i \mid \text{the vertex of } e \text{ is an in-neighbor of the vertex of } S_i \text{ in } G^+\}. \quad (6)$$

Property 5.

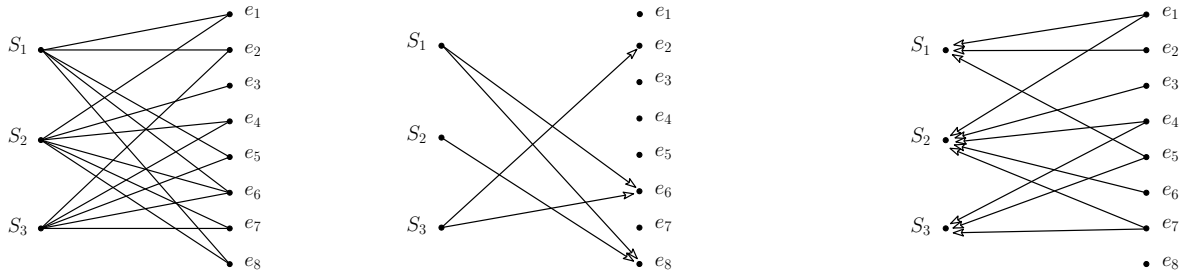
$$\sum_{i,j:1\leq i<j\leq n} |S_i^- \cap S_j^-| \leq 2\alpha N.$$

Proof. For each element $e \in U$, define $deg^+(e)$ to be the out-degree of the vertex of e in G^+. By Lemma 1, $deg^+(e) \leq 2\alpha$. Observe:

$$\sum_{i,j:1\leq i<j\leq n} |S_i^- \cap S_j^-| = \sum_e \binom{deg^+(e)}{2}. \quad (7)$$

As $\binom{deg^+(e)}{2} < (deg^+(e))^2 \leq 2deg^+(e) \cdot \alpha$, the right hand side of (7) is bounded by $\sum_e 2deg^+(e) \cdot \alpha \leq 2\alpha N$. □

Example 2. Continuing on the previous example, $S_1^- = \{e_1, e_2, e_5\}$, $S_2^- = \{e_1, e_3, e_4, e_6, e_7\}$, and $S_3^- = \{e_4, e_5, e_7\}$. The left hand side of (7) equals 4. Note that this is (much) less than $2\alpha N = 2 \cdot 2 \cdot 16 = 64$. □



(a) Companion graph G (b) Edges of G^+ directed from left to right (c) Edges of G^+ directed from right to left

Fig. 4. A running example

5 A Set Intersection Index

This section will describe a data structure to solve the (classical) set intersection problem, or equivalently, SIPF where $\mathcal{F}$ has only the identity filtering function (see Section 2). The structure will serve as the core of our other SIPF indexes presented in the next section.

5.1 Structure

As before, let $S_1, S_2, ..., S_n$ be the input set collection, G be its companion graph, and G^+ the orientation of G output by the **orient** algorithm (Section 3.2). In a manner similar to S_i^- in (6), we define for each $i \in [n]$

$$S_i^+ = \{e \in S_i \mid \text{the vertex of } e \text{ is an out-neighbor of the vertex of } S_i \text{ in } G^+\}.$$

For any $i \in [n]$, $S_i^- \cap S_i^+ = \emptyset$ and $S_i^- \cup S_i^+ = S_i$.

Our structure has the following components.

- **C1** for each $i \in [n]$, store S_i in a hash table that, given an element e, can decide if $e \in S_i$ in $O(1)$ time;
- **C2** for each $i \in [n]$, store S_i^+ in an array;
- **C3** for every 2-tuple (i, j) where $1 \leq i < j \leq n$, store $S_i^- \cap S_j^-$ in an array if $S_i^- \cap S_j^-$ is non-empty.

Example 3. Continuing on Example 2, $S_1^+ = \{e_6, e_8\}$, $S_2^+ = \{e_8\}$, and $S_3^+ = \{e_2, e_6\}$ (see Figure 2(c)). Component **C3** stores $S_1^- \cap S_2^- = \{e_1\}$, $S_1^- \cap S_3^- = \{e_5\}$, and $S_2^- \cap S_3^- = \{e_4, e_7\}$. □

5.2 Query

Consider a query that returns $S_a \cap S_b$ where $1 \leq a < b \leq n$. Each element $e \in S_a \cap S_b$ belongs to exactly one of the following categories:

- **(Cat. 1)** $e \in S_a^- \cap S_b^-$;
- **(Cat. 2)** $e \in S_a^+$ or S_b^+.

Elements of Cat. 1 have been explicitly stored in component **C3** and are output directly. To report Cat. 2, we perform the following steps:

report-Cat-2 (a, b)
1. **for** each $e \in S_a^+$ **do**
2. **if** $e \in S_b$ **then** report e
3. **for** each $e \in S_b^+$ **do**
4. **if** $e \in S_a$ **then** report e

Example 4. Let us apply the algorithm to compute $S_2 \cap S_3$ on our running example. First, output $S_2^- \cap S_3^- = \{e_4, e_7\}$ directly from component **C3**. Then, run **report-Cat-2**. As S_2^+ has only one element e_8, the algorithm probes the hash table on S_3 to find out $e_8 \notin S_3$. Next, **report-Cat-2** examines the elements of S_3^+ and realizes $e_6 \in S_2$ but $e_2 \notin S_2$. The final answer is $\{e_4, e_6, e_7\}$. □

5.3 Analysis

It is easy to verify that components **C1** and **C2** use $O(N)$ space. By Property 5, component **C3** occupies $O(\alpha N)$ space, where α is the pseudoarboricity of the input set collection. We thus conclude that the overall space consumption is $O(\alpha N)$. Given that **orient** finishes in $O(|V| + |E|)$ time (Lemma 1), it is rudimentary to build our structure in $O(\alpha N)$ time.

Let us now turn to query time. All the elements in Cat. 1 can be reported in linear time because they are explicitly stored. Algorithm **report-Cat-2** takes time $O(|S_a^+| + |S_b^+|)$, observing that Line 2 (resp., 4) requires only constant time by resorting to the hash table on S_b (resp., S_a). Lemma 1 assures us that $|S_a^+| + |S_b^+| = O(\alpha)$. We thus conclude that the overall time complexity is $O(\alpha + \text{OUT})$, where $\text{OUT} = |S_a \cap S_b|$.

Theorem 6. *For the set intersection problem, there is a structure of $O(\alpha N)$ space that answers a query in $O(\alpha + \text{OUT})$ time, where N is the total size of all the input sets, α is the pseudoarboricity of the input set collection, and OUT is the number of elements reported. The structure can be built in $O(\alpha N)$ time.*

5.4 Measuring Hardness with Pseudoarboricity

Next, we explain why the space-query tradeoff in Theorem 6 is unlikely to admit significant improvement.

Theorem 7. *Unless the strong set intersection conjecture is wrong, the following statements are true about the set intersection problem.*

- *No structure of $O(\alpha N \operatorname{polylog} N)$ space can answer a query in $O(\alpha^{1-\delta} + \text{OUT})$ time.*
- *No structure that answers a query in $O(\alpha + \text{OUT})$ time can use $O(\alpha^{1-\delta} N \operatorname{polylog} N)$ space.*

In the above statements, $\delta > 0$ is an arbitrarily small constant, and the meanings of N, α, and OUT follow those in Theorem 6.

Proof. Consider an arbitrary set-intersection structure that uses $spc(N)$ space and answers a query in $qry(N) + O(\text{OUT})$ time. The strong set intersection conjecture, as reviewed in Section 3.1, states that $spc(N) \cdot qry(N)$ needs to be $\tilde{\Omega}(N^2)$.

By Property 3, the pseudoarboricity of the input set collection must be $O(\sqrt{N})$. Suppose that there exists a structure with $spc(N) = O(\alpha N \operatorname{polylog} N)$ and $qry(N) = O(\alpha^{1-\delta})$. Then, $spc(N) \cdot qry(N) = O(\alpha^{2-\delta} N \operatorname{polylog} N) = O(N^{2-\delta/2} \operatorname{polylog} N)$, which defies the strong set intersection conjecture. This proves the first bullet of the theorem. A similar argument proves the second bullet. □

6 SIPF INDEXES

We will proceed to explain how to adapt the index in the previous section to tackle SIPF in general. Section 6.1 (resp., 6.2) will focus on predicate-based (resp., reducible) filtering. Section 6.3 will deal with other types of filtering functions.

6.1 Predicate-Based Filtering

Denote by $\mathcal{F}$ the set of possible filtering functions. Each function is defined by a predicate π that maps an element $e \in \mathbb{D}$ to either true or false. Given distinct set ids i, j and a predicate π, a query reports all the elements $e \in S_i \cap S_j$ with $\pi(e) =$ true.

Let us temporarily depart from the SIPF context to talk about what we call the *foundation problem*. Let D be a set of elements from $\mathbb{D}$. Given a predicate π, a *foundation query* returns all the elements $e \in D$ satisfying $\pi(e) =$ true. We assume the availability of a *foundation structure* that stores D in $spc(|D|)$ space and can answer a foundation query in $qry(|D|) + O(\text{OUT})$ time where OUT is the number of elements reported. We also assume that $spc(.)$ grows at least linearly (see Section 2 for what this means).

Example 5. SIPF-range (defined in Section 2) has the following foundation problem. Let D be a set of points in $\mathbb{R}^d$. Given an axis-parallel rectangle q in $\mathbb{R}^d$, a foundation query returns all the elements $e \in D$ such that $e \in q$. If $d = 1$, a foundation structure can be a BST (binary search tree), which uses $spc(|D|) = O(|D|)$ space and answers a query in $O(\log |D| + \text{OUT})$ time, i.e., $qry(|D|) = O(\log |D|)$. □

Returning to the SIPF problem defined by $\mathcal{F}$, we modify component **C3** of the index in Section 5 as follows:

C3 For every 2-tuple (i, j) where $1 \leq i < j \leq n$, create a foundation structure on $S_i^- \cap S_j^-$ if $S_i^- \cap S_j^- \neq \emptyset$.

To answer a query with set IDs parameters a, b (with $a < b$) and predicate π, we apply two changes to the algorithm in Section 5. First, to retrieve the elements of Cat. 1, we use the foundation structure on $S_a^- \cap S_b^-$ to find all the elements $e \in S_a^- \cap S_b^-$ with $\pi(e) =$ true. Second, in **report-Cat-2**, Line 1 now becomes "**for** each $e \in S_a^+$ with $\pi(e) =$ true **do**", and Line 3 becomes "**for** each $e \in S_b^+$ with $\pi(e) =$ true **do**".

By adapting the analysis of Section 5 in a straightforward manner, we can show:

Theorem 8. *Consider SIPF with predicate-based filtering. Suppose that the foundation problem admits a foundation index that, given a set $D \subseteq \mathbb{D}$, uses $spc(|D|)$ space and answers a query in $O(qry(|D|) + \text{OUT})$ time. Then, for SIPF with predicate-based filtering, there is a structure of $O(N) + spc(O(\alpha N))$ space that can answer a query in $O(\alpha + qry(N) + \text{OUT})$ time, where N is the total size of all the input sets, α is the arboricity of the input set collection, and* OUT *is the number of elements reported.*

Applications. Let us apply the theorem to tackle SIPF-range. In the foundation problem, we have a set D of points in $\mathbb{R}^d$. Given an axis-parallel rectangle q in $\mathbb{R}^d$, a foundation query reports all the points of D that are covered by q.

- For $d = 1$, as mentioned, the BST serves as the foundation structure with $spc(|D|) = O(D)$ and $qry(|D|) = O(\log |D|)$. Theorem 8 yields an SIPF structure of $O(\alpha N)$ space that answers a query in $O(\alpha + \log N + \text{OUT})$ time.
- For $d \geq 2$, we can apply the *range tree* [23] as the foundation structure, which ensures $spc(|D|) = O(|D| \log^{d-1} |D|)$ and $qry(|D|) = O(\log^{d-1} |D|)$. Theorem 8 then yields an SIPF structure of $O(\alpha N \log^{d-1} N)$ space that answers a query in $O(\alpha + \log^{d-1} N + \text{OUT})$ time.

Consider now SIPF-halfspace (see Section 2). In the foundation problem, we have a set D of points in $\mathbb{R}^d$. Given a halfspace q in $\mathbb{R}^d$, a foundation query reports all the points of D covered by q. For $d \leq 3$, a structure of Afshani and Chan [1] can serve as the foundation index, which ensures $spc(|D|) = O(|D|)$ and $qry(|D|) = O(\log |D|)$. Theorem 8 yields an SIPF structure of $O(\alpha N)$ space that answers a query in $O(\alpha + \log N + \text{OUT})$ time.

Hardness Remark. Subject to the strong set intersection conjecture, Theorem 8 can no longer be significantly improved. Suppose, on the contrary, that someone can leverage the foundation index to obtain an SIPF structure of $spc(\alpha N)$ space and query time $O(\alpha^{1-\delta} + qry(N) + \text{OUT})$ for some constant $\delta > 0$. For 1D SIPF-range, this promises a structure of $O(\alpha N)$ space and $O(\alpha^{1-\delta} + \log N + \text{OUT})$ query time. But then we can defy the aforementioned conjecture as follows. Let $S_1, S_2, ..., S_n$ be the input set collection (for set intersection). Create an input for 1D SIPF-range by associating each element $e \in \bigcup_{i=1}^n S_i$ with an arbitrary real value p_e. Given any set IDs a and b, we issue a 1D SIPF-range query with the range $q = (-\infty, \infty)$, whose output is precisely $S_a \cap S_b$. By resorting to the SIPF-range structure, we obtain a set-intersection index of $O(\alpha N)$ space and $O(\alpha^{1-\delta} + \log N + \text{OUT})$ query time, breaking the conjecture.

It is less obvious why no one can leverage the foundation index to obtain an SIPF structure of $spc(\alpha N)$ space and $O(\alpha + \text{OUT}) + o(qry(N))$ query time. Suppose that such a guarantee is attainable. Then, whenever $spc(N) = \Omega(N)$, we can obtain an asymptotically better foundation index as follows. Given an input D to the foundation problem, we create two identical sets $S_1 = S_2 = D$, which make an SIPF input whose pseudoarboricity α is at most 2 (Property 4). Given any predicate π, we can answer a foundation query by issuing an SIPF query with set IDs $a = 1$, $b = 2$, and predicate π. The SIPF query returns exactly the same answer as the foundation query. By resorting to the SIPF structure, we have created an index for the foundation problem that uses $O(N) + O(spc(N)) = O(spc(N))$ space but promises $o(qry(N)) + O(\text{OUT})$ query time.

Similarly, one can also argue about the difficulty of obtaining an SIPF index of $O(spc(\alpha^{1-\delta} N))$ space and $O(\alpha + qry(N) + \text{OUT})$ query time.

6.2 Reducible Filtering

Let $\mathcal{F}$ be a singleton set comprising a single reducible filtering function F (see (1) for definition). Given distinct set ids a, b, a query reports $F(S_a \cap S_b)$. Similar to Section 5.2, we divide $S_a \cap S_b$ into two disjoint sets: $D_1 = S_a^- \cap S_b^-$ and $D_2 = (S_a^+ \cup S_b^+) \cap (S_a \cap S_b)$. As each element in $F(S_a \cap S_b)$ either belongs to D_1 or D_2, we can divide $F(S_a \cap S_b)$ into two disjoint sets: $D_1' = D_1 \cap F(D_1 \cup D_2)$ and $D_2' = D_2 \cap F(D_1 \cup D_2)$. As will be clear later, we will precompute D_1' and store it in

our index. The non-trivial part is to compute D'_2. For that purpose, we leverage the following lemma that is enabled by the reducible property.

Lemma 9. $F(D_1 \cup D_2) = F(D'_1 \cup D_2)$.

Proof. Set $D = D_1 \cup D_2$ and $D' = D'_1 \cup D_2$. We will first prove $F(D) \subseteq D'$. Consider an arbitrary element $e \in F(D)$; trivially, $e \in D_1 \cup D_2$. If $e \in D_1$, then $e \in D_1 \cap F(D) = D'_1 \subseteq D'$. Otherwise, e must belong to D_2 and thus also belong to D'. This indicates $F(D) \subseteq D'$. Combining this with the obvious property $D' \subseteq D$, we can now conclude $F(D) = F(D')$ from the reducible property given in (1). □

By Lemma 9, finding D'_2 is equivalent to computing $D_2 \cap F(D'_1 \cup D_2)$. To that end, we need to solve a general problem, defined below as the *shielding problem*.

The shielding problem involves two disjoint sets $D'_1 \subseteq \mathbb{D}$ and $D_2 \subseteq \mathbb{D}$ with the guarantee $D'_1 \subseteq F(D'_1 \cup D_2)$. There are two phases: preprocessing and shielding. In preprocessing, we are allowed to create a *shielding structure* on D'_1. The goal of the shielding phase is to find $D_2 \cap F(D'_1 \cup D_2)$, namely, eliminating those elements of D_2 that do not belong to $F(D'_1 \cup D_2)$. We assume the availability of a shielding structure that uses $spc(|D_1|)$ space and enables us to implement the shielding phase in $shield(|D'_1|, |D_2|)$ time. We also assume that $spc(.)$ grows at least linearly.

Example 6. The SIPF-skyline problem, where $\mathcal{F}$ has a single function given in (2), has the following shielding problem. Let D'_1 and D_2 be two sets of points in $\mathbb{R}^d$ ($d \geq 2$) such that D'_1 is a subset of the skyline of $D'_1 \cup D_2$. The goal of the shielding phase is to report all the points in D_2 that appear in the skyline of $D'_1 \cup D_2$. In preprocessing, we create a range tree on D'_1 as the shielding structure, which uses $spc(|D'_1|) = O(|D'_1| \log^{d-1} |D'_1|)$ space. Given any point $q \in \mathbb{R}^d$, the structure can decide in $O(\log^{d-1} |D'_1|)$ time if any point in D'_1 dominates q. In the shielding phase, we first compute the skyline of D_2 in $O(|D_2| \log^{d-2} |D_2|)$ time using an algorithm in [41]. For each point q in the skyline of D_2, use the range tree to check if any point in D'_1 dominates q. If not, q appears in the skyline of $D'_1 \cup D_2$ and is thus reported. The shielding phase takes $shield(|D'_1|, |D_2|) = O(|D_2| \log^{d-1}(|D'_1| + |D_2|))$ time in total. □

Now, we return to reducible filtering and introduce our index structure and the query algorithm formally. For every pair (i, j) satisfying $1 \leq i < j \leq n$, define

$$D^-(i,j) \quad = \quad \{e \mid e \in S_i^- \cap S_j^- \cap F(S_i \cap S_j)\} \qquad (8)$$

It is worth pointing out that $D^-(i, j)$ is not equivalent to $F(S_i^- \cap S_j^-)$. To construct an index, all we need is to modify component **C3** of the set-intersection index in Section 1 to:

C3 for every 2-tuple (i, j) where $1 \leq i < j \leq n$, if $D^-(i, j) \neq \emptyset$, store $D^-(i, j)$ in an array and create a shielding structure on $D^-(i, j)$.

To answer a query with set IDs a, b (with $a < b$), we

- retrieve $D'_1 = D^-(a, b)$;
- retrieve D_2, which is the set of elements in Cat. 2 defined in Section 5.2.

Next, use the shielding structure on D'_1 to obtain $D_2 \cap F(D'_1 \cup D_2)$. By Lemma 9, this is exactly D'_2. Finally, return $D'_1 \cup D'_2$.

By adapting the analysis in Section 5.3, we can conclude with the following theorem:

Theorem 10. *Consider SIPF with reducible filtering. Suppose that the shielding problem admits a shielding structure that uses $spc(|D_1|)$ space and allows us to perform the shielding phase in $shield(|D_1|, |D_2|)$ time. Then, there is a structure for SIPF with reducible filtering that uses $O(N) + spc(O(\alpha N))$ space and answers a query in $O(shield(\mathrm{OUT}, \alpha) + \mathrm{OUT})$ time, where N is the total size of all the input sets, α is the pseudoarboricity of the input set collection, and OUT is the number of elements reported.*

Applications. Let us apply the theorem to tackle SIPF-skyline in $\mathbb{R}^d$. In Example 6, we have described a shielding structure if $spc(|D_1|) = O(|D_1| \log^{d-1} |D_1|)$ space that can perform shielding in $shield(|D_1|, |D_2|) = O(|D_2| \log^{d-1}(|D_1|+|D_2|))$ time. The theorem yields an SIPF-skyline index of space $O(\alpha N \log^{d-1} N)$ and $O(\alpha \log^{d-1} N + \mathrm{OUT})$ query time.

Consider now the SIPF-CH problem in $\mathbb{R}^d$ with $d \leq 3$. The set $\mathcal{F}$ contains only one filtering function F, given in (3). In the corresponding shielding problem, D_1 and D_2 are two sets of points in $\mathbb{R}^d$ such that all the points in D_1 are vertices of the convex hull of $D_1 \cup D_2$. The goal of the online phase is to return the points in D_2 that are vertices of the convex hull of $D_1 \cup D_2$. In preprocessing, we store D_1 in $spc(|D_1|) = O(|D_1|)$ space. In the shielding phase, we first compute the convex hull of D_2 in $O(|D_2| \log |D_2|)$ time, after which the convex hull of $D_1 \cup D_2$ can be obtained by merging D_1 with the convex hull of D_2 in $O(|D_1| + |D_2|)$ time [13]. The points in D_2 on the convex hull of $D_1 \cup D_2$ are obtained as a side product. Hence, $shield(|D_1|, |D_2|) = O(|D_2| \log |D_2| + |D_1|)$. Theorem 10 then yields an SIPF-CH index with space $O(\alpha N)$ and query time $O(\alpha \log \alpha + \mathrm{OUT})$.

Hardness Remark. Our indexes no longer allow significant improvement unless the strong set-intersection conjecture is wrong. We will explain this for SIPF-skyline as the argument is similar for SIPF-CH. Suppose that someone can design an SIPF-skyline structure with $O(\alpha N \operatorname{polylog} N)$ space and $O(\alpha^{1-\delta} \operatorname{polylog} N + \mathrm{OUT})$ query time for some constant $\delta > 0$. We can utilize the structure to defy the conjecture. Let $S_1, S_2, ..., S_n$ be the input set collection for set intersection. Create an SIPF-skyline input by associating each element $e \in \bigcup_{i=1}^{n} S_i$ with a 2D point $p_e = (x, -x)$ where x is an integer uniquely assigned to e. Given set IDs a and b, the skyline of $S_a \cap S_b$ is exactly $S_a \cap S_b$. Thus, we obtain a set-intersection index of $O(\alpha \operatorname{polylog} N)$ space and $O(\alpha^{1-\delta} \operatorname{polylog} N + \mathrm{OUT})$ query time, breaking the conjecture. Likewise, one can also argue about the difficulty of obtaining an SIPF-skyline index with $O(\alpha^{1-\delta} N \operatorname{polylog} N)$ space and $O(\alpha \operatorname{polylog} N + \mathrm{OUT})$ query time.

6.3 Filtering by Deflation

This subsection will introduce a "deflation strategy", which, despite appearing straightforward initially, proves to be a powerful approach for addressing certain SIPF problems. To motivate the strategy, suppose that we want to compute $F(S_a \cap S_b)$ where a and b are distinct set ids and $F \in \mathcal{F}$ is a filtering function. As in Section 5.2, define $D_1 = S_a^- \cap S_b^-$ and $D_2 = (S_a^+ \cup S_b^+) \cap (S_a \cap S_b)$. Since $D_1 \cup D_2 = S_a \cap S_b$,

our goal is essentially to compute $F(D_1 \cup D_2)$. The *deflation strategy* fulfills the goal in two steps:

- First, compute certain information — denoted as D_3 — from some pre-computed data structure. Intuitively, D_3 serves as a "deflated version" of D_1 for the purpose of computing $F(D_1 \cup D_2)$.
- Then, obtain $F(D_1 \cup D_2)$ from D_2 and D_3.

To make the above strategy work efficiently, we need to impose some requirements, which give rise to the following *deflation problem*. Let $\mathcal{F}$ be an arbitrary set of filtering functions, and let $D_1 \subseteq \mathbb{D}$ and $D_2 \subseteq \mathbb{D}$ be two disjoint sets. There are two phases: preprocessing and online. In preprocessing, we are allowed to create a certain structure, called the *deflation structure*, on D_1 of space $O(|D_1| \operatorname{polylog} |D_1|)$. In the online phase, we are given a filtering function $F \in \mathcal{F}$ and need to compute some information, represented as D_3, in $O(\operatorname{polylog} |D_1| + \mathrm{OUT})$ time, where $\mathrm{OUT} = |F(D_1 \cup D_2)|$. The requirement for D_3 is that we must be able to compute $F(D_1 \cup D_2)$ from D_2 and D_3 — rather than from D_1 and D_2 — in $O(|D_2| \operatorname{polylog} |D_2| + \mathrm{OUT})$ time.

Example 7. Consider the SIPF-NN problem. As in (4), every filtering function $F \in \mathcal{F}$ has two parameters: an integer $k \geq 1$ and a query point $q \in \mathbb{R}^2$. Next, we will explain how to solve the underlying deflation problem, assuming $k \leq |D_1 \cup D_2|$ (the reader can then extend our discussion to the opposite scenario, which is trivial). The deflation problem includes two disjoint sets of points in $\mathbb{R}^2$: D_1 and D_2. In preprocessing, we store D_1 in a structure of [1] that uses $O(|D_1|)$ space and allows us to find the k nearest neighbors of any query point in D_1 in $O(\log |D_1| + k)$ time. In the online phase, given an integer k and a query point q, we compute D_3, which comprises the k nearest neighbors of q in D_1. Given D_3, we can compute $F(D_1 \cup D_2)$ — that is, k nearest neighbors of q in $D_1 \cup D_2$ — in $O(|D_2| + k) = O(|D_2| + \mathrm{OUT})$ time as follows. First, extract the k nearest neighbors of q in D_2 by scanning D_2 entirely in $O(|D_2|)$ time. Then, perform k-selection [22] to find the k nearest neighbors of q in $D_2 \cup D_3$. The cost of the k-selection is $O(|D_2 \cup D_3|) = O(|D_2| + \mathrm{OUT})$. □

To solve the original SIPF problem, we modify component **C3** of our set-intersection index in Section 5 as follows:

C3 for every 2-tuple (i, j) where $1 \leq i < j \leq n$, create a deflation structure for $D_1 = S_i^- \cap S_j^-$ and $D_2 = (S_i^+ \cup S_j^+) \cap (S_i \cap S_j)$.

The space consumption of the modified index is $O(\alpha N \operatorname{polylog} N)$.

Suppose that we are now given a query with by set IDs a, b, and filtering function $F \in \mathcal{F}$. Let $D_1 = S_a^- \cap S_b^-$ and $D_2 = (S_a^+ \cup S_b^+) \cap (S_a \cap S_b)$. By supplying F to the deflation structure created on (D_1, D_2), we obtain D_3 in $O(\operatorname{polylog} N + |D_3|) = O(\operatorname{polylog} N + \mathrm{OUT})$ time. After that, we compute $F(S_a \cap S_b) = F(D_1 \cup D_2)$ from D_2 and D_3 in $O(|D_2| \operatorname{polylog} N + \mathrm{OUT}) = O(\alpha \operatorname{polylog} N + \mathrm{OUT})$ time. This establishes the result below.

Theorem 11. *Consider any SIPF problem. If we manage to find a solution to the underlying deflation problem, then we have a structure for the SIPF problem that uses $O(\alpha N \operatorname{polylog} N)$ space and answers a query in $O(\alpha \operatorname{polylog} N + \mathrm{OUT})$ time, where N is the total size of all the input sets, α is the pseudoarboricity of the input set collection, and OUT is the number of elements reported.*

Applications. In Example 7, we have explained how to solve the deflation problem underlying SIPF-NN. The theorem thus yields an SIPF-NN index of space $O(\alpha N \operatorname{polylog} N)$ and $O(\alpha \operatorname{polylog} N + k)$ query time (the value of OUT is k).

Next, we will tackle the SIPF-quantile problem defined in Section 2. The deflation problem includes two disjoint sets of real values: D_1 and D_2. Let $t = |D_1 \cup D_2|$ and Q be the set of k-quantiles of $D_1 \cup D_2$. We want to build a structure of $O(|D_1| \operatorname{polylog} |D_1|)$ space in preprocessing. In the online phase, we use the structure to find in $O(\operatorname{polylog} |D_1| + k)$ time a set D_3 of size $O(k)$ that contains $D_1 \cap Q$. The set D_3 should allow us to find the k-quantiles of $D_1 \cup D_2$ in $O(|D_2| \operatorname{polylog} |D_2| + k)$ time. We will consider $k \leq |D_1 \cup D_2|$ (the opposite case is trivial).

Let $x_1, x_2, ..., x_k$ be the k-quantiles of $D_1 \cup D_2$ in ascending order. For each $i \in [k]$, define y_i to be (i) x_i if $x_i \in D_1$ or (ii) otherwise, the largest value in D_1 less than x_i. Furthermore, associate y_i with its *rank* r_i in $D_1 \cup D_2$ (i.e., y_i is the r_i-th smallest value in $D_1 \cup D_2$). Note that $y_1, y_2, ..., y_k$ may not be distinct: $y_i = y_{i-1}$ if D_1 contains no values in $(x_{i-1}, x_i]$. The number of distinct values among $y_1, y_2, ..., y_k$ is at most $\min\{|D_1|, k\}$. In preprocessing, we store these distinct values, as well as their ranks, as our structure, which uses $O(\min\{|D_1|, k\})$ space.

In the online phase, D_3 is precisely the set of distinct values among $y_1, y_2, ..., y_k$, which can trivially be obtained in $O(|D_3|)$ time (D_3 is stored directly). The size of D_3 is $O(k) = O(\mathrm{OUT})$. We then sort D_2 in $O(|D_2| \log |D_2|)$ time. For each $i \in [k]$, the $\lfloor i \cdot t/k \rfloor$-th smallest value in $D_1 \cup D_2$ can be found as follows. If $x_i \in D_1$, it is stored explicitly in D_3. Otherwise, x_i is the $(\lfloor i \cdot t/k \rfloor - r_i)$-th smallest among all the values in D_2 that are strictly greater than y_i. It is now rudimentary to find the k-quantiles of $D_1 \cup D_2$ in $O(|D_2| + |D_3|) = O(|D_2| + \mathrm{OUT})$ time by scanning D_2 and D_3 synchronously once.

Theorem 11 then yields an index for the SIPF-quantile problem that occupies $O(\alpha N \operatorname{polylog} N)$ space and $O(\alpha \operatorname{polylog} N + k)$ query time.

Hardness Remark. Our SIPF-NN and SIPF-quantile structures are difficult to improve substantially subject to the strong set-intersection conjecture. The arguments are analogous to those in the remarks of Sections 6.1 and 6.2.

7 Experiments

This section presents an experimental evaluation of the proposed techniques. We will start in Section 7.1 by examining our index in Section 5 for "pure" set intersection, i.e., no post filtering is necessary. Then, Sections 7.2-7.4 will present the results on predicate-based, reducible, and deflation filtering, respectively. All the experiments were performed on a machine equipped with an Intel CPU 3.6GHz and 82 Gbtyes of memory. The operating system was Ubuntu 20.04.

7.1 Set Intersection without Post Filtering

Our structure in Section 5 is designed for classical set intersection, where the dataset is a collect of sets $S_1, S_2, ..., S_n$, and a query returns $S_a \cap S_b$ for distinct $a, b \in [n]$. In this section,

name	number n of sets	N	max set size	max element frequency	max. out-degree by orient	cut-off set size
IMDB	351,109	4,647,097	10,407	24,107	89	306
Google	178,203	22,685,631	9,998	1,591	60	723
Amazon	2,775,193	47,523,975	58,147	7,303	71	618

TABLE 1
Statistics of the datasets deployed

we will analyze the structure's behavior on real-world data and lay the groundwork for understanding the behavior of the other indexes to be shown in the subsequent sections.

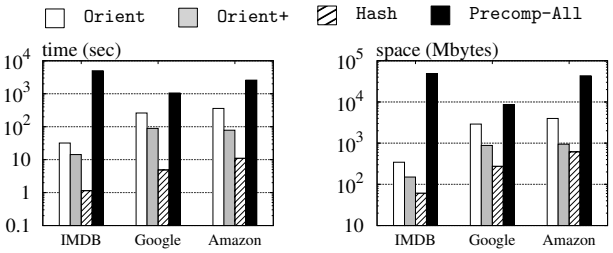
Competing Methods. We will refer to the structure in Section 5 as `Orient` (named after the fact that it works by orienting the edges in the companion graph; see Figure 4). We compared `Orient` with two benchmark solutions:

- `Precomp-All`: This method pre-computes the intersection $S_i \cap S_j$ for all $1 \leq i < j \leq n$. While `Precomp-All` is highly query-efficient, allowing for direct retrieval of the pre-stored $S_a \cap S_b$ given the set IDs a and b, it has the drawbacks of high space consumption and expensive pre-computation time.
- `Hash`: This method creates a hash table on each set S_i ($i \in [n]$). To answer a query with set IDs a and b (w.l.o.g., assume $|S_a| \leq |S_b|$), `Hash` reads S_a once and for each element $e \in S_a$ checks whether $e \in S_b$ by probing the hash table of S_b. `Hash` requires only linear space, but it may suffer from high query cost when $|S_a|$ and $|S_b|$ are both large.

We also developed a new method, called `Orient+`, which combines the concepts of `Orient` and `Hash`. Our rationale for this approach is that, when at least one of the input sets (S_a or S_b) is small, `Hash` has superior query efficiency, with a query time of $O(\min\{|S_a|, |S_b|\})$. Therefore, it is beneficial to construct our structure in Section 5 only on large sets. Let $s_1, s_2, ..., s_t$ — in descending order — denote the distinct sizes of the sets in the underlying dataset (note that t may be smaller than n because some sets may have identical sizes). We define a *cutoff set size* $s_{\lfloor \eta t \rfloor}$, where η is a parameter between 0 and 1. We build our structure only on the sets whose sizes are at least $s_{\lfloor \eta t \rfloor}$, which we refer to as *colossal sets*. Sets with sizes less than the cutoff value are considered *tail sets* and are pre-processed using only a hash table. In query processing, if either S_a or S_b is a tail set, we apply the query algorithm of `Hash`. Otherwise, we use the algorithm in Section 5.

The parameter η plays a crucial role in achieving a smooth transition between `Hash` and `Orient`, which represent the two extreme cases of $\eta = 0$ and 1, respectively. A higher value of η consumes more space but entails lower query cost. In all the experiments presented below, we set $\eta = 0.75$, which was empirically chosen to strike a good balance between the space and query efficiency of `Orient+`.

As noted in Section 3, the system community has developed an impressive array of techniques for answering set intersection queries. However, our experiments focus on comparing our approach with `Hash` for four reasons. First, our goal is to assess the proposed theoretical concepts using a standardized benchmark, rather than claiming practical superiority over the existing methods. Second, `Hash` is an excellent choice for the benchmark due to its simplicity and popularity in the literature. Third, many existing techniques, such as those based on compression like the *roaring bitmap* [11, 43], can be integrated into our structure to further enhance its performance. Hence, a comparison with those techniques would be unfair in the absence of such integration, but incorporating the integration would complicate the resulting structure and impede a clear discussion of its behavior. Fourth, in general, once we have obtained a thorough understanding of a new indexing paradigm's characteristics, it is often relatively easy to engineer heuristics for performance improvements. In this work, we have paid little attention to such heuristics.



(a) Preprocessing time (b) Space

data, method	S_i	$S_i^- \cap S_j^-$	S_i^+	total
IMDB, `Orient`	61.4	271	11.8	344
IMDB, `Orient+`	61.4	83.0	6.22	151
IMDB, `Hash`	61.4	–	–	61.4
Google, `Orient`	275	2610	16.5	2902
Google, `Orient+`	275	601	4.28	880
IMDB, `Hash`	275	–	–	275
Amazon, `Orient`	615	3270	89.7	3975
Amazon, `Orient+`	615	284	45.1	944
Amazon, `Hash`	615	–	–	615

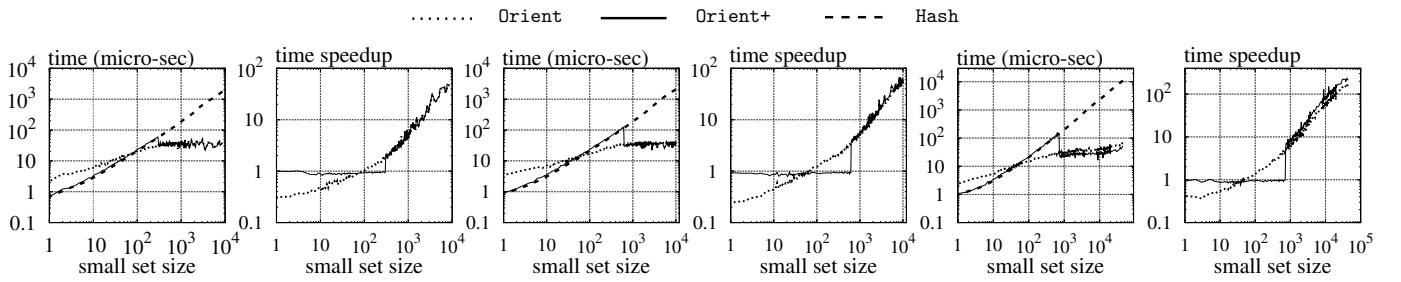
(c) Space breakdown (Mbytes)

Fig. 5. Preprocessing and space costs for pure set intersection

Data. We utilized three real datasets, as are introduced below:

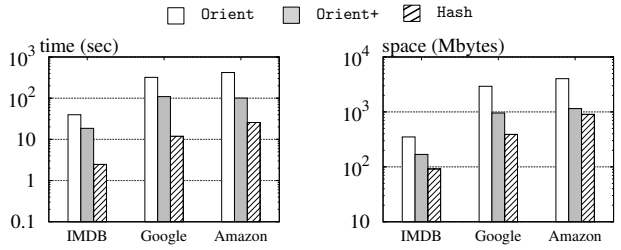
- IMDB: This dataset was downloaded from *ieee-dataport.org/open-access/imdb-movie-reviews-dataset* and includes $n = 351{,}109$ sets. Each set corresponds to a movie and contains the IDs of the users that have rated the movie.
- Google: This dataset was downloaded from *jiachengli1995.github.io/google/index.html* and includes $n = 178{,}203$ sets. Each set corresponds to a business listed on Google Maps and contains the IDs of the users that have written a review about the business.
- Amazon: This dataset was downloaded from *nijianmo.github.io/amazon/index.html#subsets* and includes $n = 2{,}775{,}193$ sets. Each set corresponds to a product sold on Amazon and contains the IDs of the users that have ever commented on the product.

Table 1 summarizes the main statistics of the datasets introduced above. The third column shows the total size of all the sets in each dataset, while the fourth column gives the maximum size of a set within the dataset. For a given dataset, the *frequency* of an element e is defined as the number of sets containing e; the fifth column of Table 1 presents the maximum frequency of all elements across the



(a) Query time (IMDB) (b) Speedup over Hash (IMDB) (c) Query time (Google) (d) Speedup over Hash (Google) (e) Query time (Amazon) (f) Speedup over Hash (Amazon)

Fig. 6. Query time vs. small set size (pure set intersection)



(a) Preprocessing time (b) Space

data, method	S_i	$S_i^- \cap S_j^-$	S_i^+	map	total
IMDB, Orient	61.4	271	11.8	6.00	350
IMDB, Orient+	73.0	83.0	6.22	6.00	168
IMDB, Hash	85.6	–	–	6.00	91.6
Google, Orient	275	2610	16.5	21.5	2923
Google, Orient+	328	601	4.28	21.5	955
Google, Hash	369	–	–	21.5	391
Amazon, Orient	615	3270	89.7	56.9	4032
Amazon, Orient+	763	284	45.1	56.9	1152
Amazon, Hash	849	–	–	56.9	906

(c) Space breakdown

Fig. 7. Preprocessing and space costs for SIPF-range

dataset. For each dataset, we applied the **orient** algorithm in Section 3.2 on its companion graph; the sixth column gives the maximum out-degree of the orientation obtained. The last column shows the dataset's cutoff set size for Orient+.

Workloads. We define a *workload* as a set of 10,000 queries, where each query is generated independently as follows. Let $s_1, s_2, ..., s_t$ denote the distinct sizes of the sets in the underlying dataset, sorted in descending order. First, we randomly choose a size x from $\{s_2, ..., s_t\}$. Then, we select (i) a set S_a uniformly at random from all the sets in the dataset whose sizes are exactly x, and (ii) another set S_b uniformly at random from all the sets whose sizes are greater than x. The query's parameters are the set IDs a and b. The number x will be called the query's *small-set size*.

Results. Figure 5(a) compares the preprocessing time of different methods. Figure 5(b) compares their space consumption, followed by a breakdown in Figure 5(c). Specifically, for Orient and Orient+, the second column in the table of Figure 5(c) is the total space of the hash tables on all S_i ($i \in [n]$), the third column is the total storage space of all $S_i^- \cap S_j^-$ ($1 \leq i < j \leq n$), and the fourth column is the total space of all S_i^+ ($i \in [n]$). The last column sums up the previous columns. For Hash, the space usage includes only the hash tables, as shown in the second column.

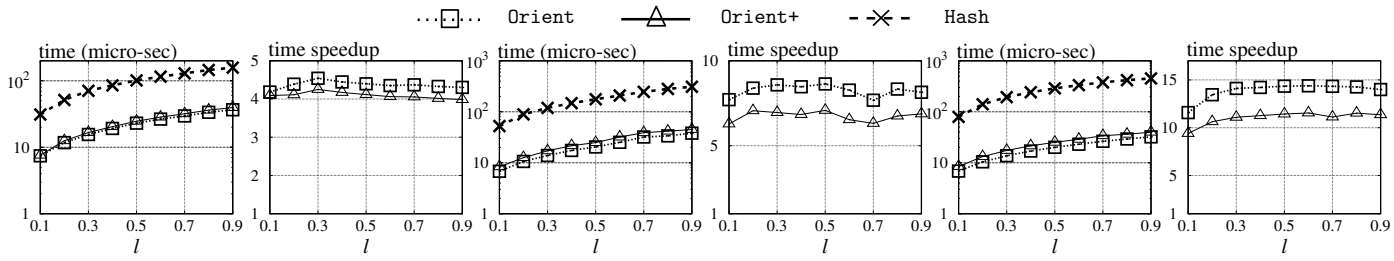
Among the methods compared, Precomp-All required by far the longest precomputation time and the highest space consumption. This underscores the importance of designing efficient index structures for set intersection queries. As expected, Hash was the fastest in precomputation and occupied the least space. The overhead of Orient and Orient+ fell between that of Precomp-All and Hash. It is evident from Figure 5(c) that a significant portion of the space of Orient and Orient+ was used for storing the intersection $S_i^- \cap S_j^-$ of $1 \leq i < j \leq n$. Orient+ significantly reduced the preprocessing and space costs of Orient.

Figure 6 presents the results on query efficiency. Let us start with Figures 6(a) and 6(b), which concern the IMDB dataset. For each method, we ran a workload and measured the cost of every query. A crucial factor affecting query cost is the small-set size, which as defined before is the smaller size of the two sets queried. In Figure 6(a), we plot the query cost of a method as a function of the small-set size. If multiple queries in a workload have the same small-set size x, the query cost plotted at x is the average of those queries. Figure 6(b) shows the speedup ratio achieved by Orient and Orient+ over Hash, also as a function of x. The speedup of Orient, for instance, at x is the query time of Orient at x divided by that of Hash in Figure 6(a). The other figures for Google and Amazon follow the same style. We do not include Precomp-All in these figures because the purpose of that method was to demonstrate the prohibitive preprocessing and space overhead of brute-force pre-computation.

Orient+ behaved similarly to Hash for small-set sizes up to a certain cutoff size. Both Orient+ and Hash outperformed Orient when the small-set size was exceedingly low. To explain this behavior, let S_a and S_b be the two sets queried, and assume that $|S_a| \leq |S_b|$. The query complexity of Orient+ and Hash is $O(|S_a|)$, while that of Orient is $O(|S_a^+| + |S_b^+|)$ because it must scan both S_a^+ and S_b^+. However, as the small-set size increased, Orient quickly outperformed the other two methods and converged with Orient+ after the small-set size reached the cutoff. The power of Orient and Orient+ is reflected in scenarios where the small-set size is large (which is also the most challenging scenario for set intersection). Their speedup ratios over Hash increased rapidly with the small-set size and even exceeded two orders of magnitude.

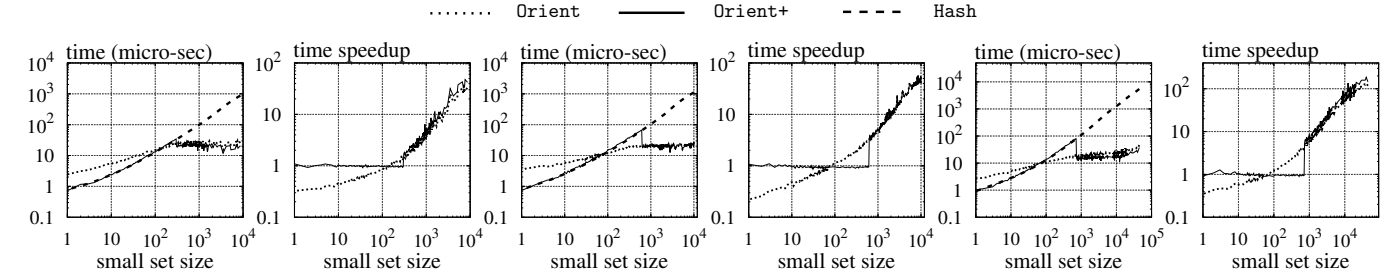
7.2 SIPF-Range

We now proceed to evaluate the index proposed in Section 6.1 for the SIPF-range problem in 1D space. As before, let $\{S_1, ..., S_n\}$ be the input set collection. Define $D = \bigcup_{i=1}^{n} S_i$. Each element $e \in D$ is associated with a 1D point $p_e \in \mathbb{R}$; a query returns the set of elements $e \in S_a \cap S_b$, for some



(a) Time (IMDB) (b) Speedup (IMDB) (c) Time (Google) (d) Speedup (Google) (e) Time (Amazon) (f) Speedup (Amazon)

Fig. 8. Query time vs. query range length (SIPF-range)



(a) Time (IMDB) (b) Speedup (IMDB) (c) Time (Google) (d) Speedup (Google) (e) Time (Amazon) (f) Speedup (Amazon)

Fig. 9. Query time vs. small set size (SIPF-range, $l = 0.5$)

distinct $a, b \in [n]$, satisfying the predicate that p_e falls in a specified interval q.

Competing Methods. For consistency, we use the name `Orient` to refer to the structure proposed in Section 6.1. Specifically, `Orient` stores

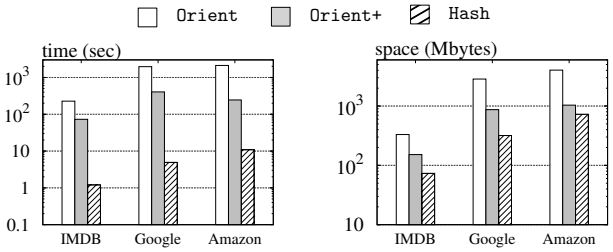
- a hash table on each S_i ($i \in [n]$);
- an array where the elements $e \in S_i^+$ ($i \in [n]$) are sorted by p_e;
- for each $1 \leq i < j \leq n$, an array where the elements $e \in S_i^- \cap S_j^-$ are sorted by p_e.

We compared the method with `Hash`, which in this context builds on each S_i ($1 \leq i \leq n$) (i) a hash table and (ii) an array where the elements $e \in S_i$ are sorted by p_e. To answer a query with set IDs a and b and an interval q, (assuming $|S_a| \leq |S_b|$) `Hash` first performs binary search on the sorted array of S_a to retrieve each element $e \in S_a$ with $p_e \in q$, and then probes the hash table of S_b to determine if $e \in S_b$. In addition, we also examined `Orient+`, which builds our structure of Section 6.1 only on the colossal sets and preprocesses the tail sets in the same way as `Hash` (see Section 7.1 for the definitions of colossal set and tail set). For all methods, the 1D points p_e are stored only once in an array — called the *map* — that, given e, permits fetching p_e in constant time.

Data. We used the set collections IMDB, Google, and Amazon that were introduced earlier. However, we augmented the datasets so that each element $e \in \mathbb{D}$ is associated with a 1D point uniformly generated in the domain $[0, 10^8]$.

Workloads. Our workload consists of 10,000 queries created using the method described in Section 7.1. However, we introduce a new parameter $l \in [0, 1]$ that is used to generate the query range $q = [x, y]$. Specifically, we randomly generate the value of x uniformly from the interval $[0, (1-l) \cdot 10^8]$, and set $y = x + l \cdot 10^8$. This ensures that all queries in the workload have the same length of q, which is equal to $l \cdot 10^8$.

Results. Figure 7(a) compares the preprocessing time of the different methods for the range query workload. The preprocessing time is generally higher than that in Figure 5(a) due to the additional step of creating sorted arrays. Figure 7(b) gives the comparison on space consumption, with a breakdown in Figure 7(c). For `Orient`, the meaning of each column should be straightforward as it follows directly from our earlier description. For `Orient+`, the meanings are the same (as `Orient`) except that the second column includes both the hash tables on all the sets and the sorted arrays on the tail sets. For `Hash`, the second column covers the hash tables and sorted arrays on all sets.



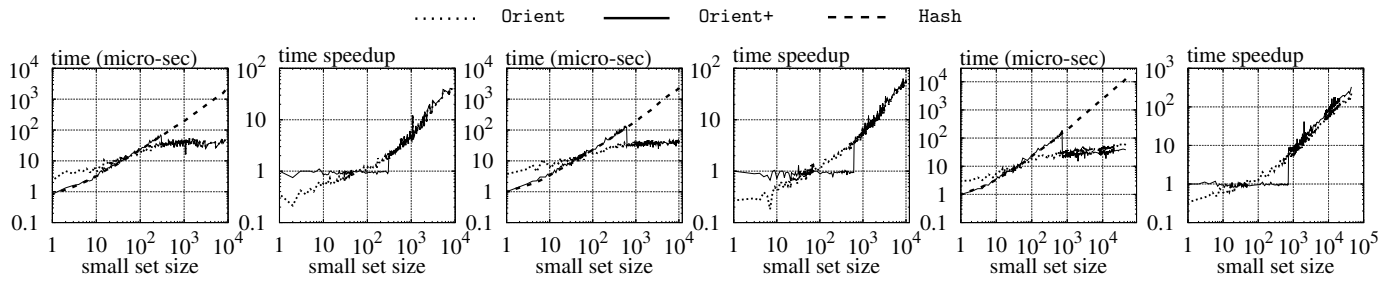
(a) Preprocessing time (b) Space

data, method	S_i	$S_i^- \cap S_j^-$	S_i^+	map	total
IMDB, `Orient`	61.4	246	11.8	12.0	331
IMDB, `Orient+`	61.4	71.0	6.22	12.0	151
IMDB, `Hash`	61.4	–	–	12.0	73.4
Google, `Orient`	275	2517	16.5	43.0	2852
Google, `Orient+`	275	546	4.28	43.0	868
Google, `Hash`	275	–	–	43.0	318
Amazon, `Orient`	615	3212	89.7	113	4031
Amazon, `Orient+`	615	262	45.1	113	1035
Amazon, `Hash`	615	–	–	113	728

(c) Space breakdown

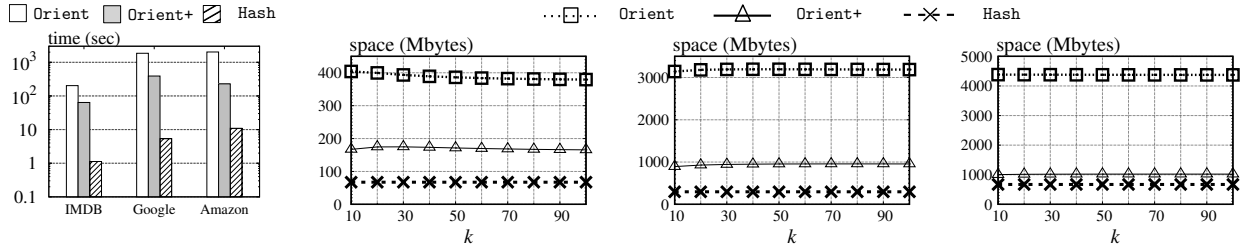
Fig. 10. Preprocessing and space costs for SIPF-skyline

The next experiment investigated the impact of a query's selectivity level on the algorithms' performance. For that purpose, we used each method to process workloads of various l and measured, for each workload, the average cost of all the queries in the workload. Focusing on IMDB, Figure 8(a) plots the workload average as a function of l for



(a) Time (IMDB) (b) Speedup (IMDB) (c) Time (Google) (d) Speedup (Google)(e) Time (Amazon)(f) Speedup (Amazon)

Fig. 11. Query time vs. small set size (SIPF-skyline)



(a) Preprocessing time (b) Space (IMDB) (c) Space (Google) Space (Amazon)

Fig. 12. Preprocessing and space costs for SIPF-quantile

all methods, and Figure 8(b) plots the corresponding speedup ratios of `Orient` and `Orient+` over `Hash` observed in Figure 8(a). The other figures in Figure 8 for Google and Amazon follow the same style. As expected, the average query cost generally increased as l became larger. `Orient` and `Orient+` retained significant speedups over `Hash` across all values of l. Between `Orient` and `Orient+`, the former had better query efficiency on average, because `Orient+` trades off some query performance for reduced space consumption and preprocessing time.

In Figure 9, we fixed l to 0.5 and examined the influence of a query's small-set size using the same approach as in Figure 6. The speedups of `Orient` and `Orient+` represent how much times faster they were over `Hash`. The general behavior was very similar to that observed in Figure 6.

7.3 SIPF-Skyline

Next, we will evaluate the index proposed in Section 6.2 for the SIPF-skyline problem in 2D space. Let $\{S_1, ..., S_n\}$ be the input set collection and define $D = \bigcup_{i=1}^{n} S_i$. Each element $e \in D$ is associated with a 2D point $p_e \in \mathbb{R}^2$; a query returns the skyline of $\{p_e \mid e \in S_a \cap S_b\}$ for some distinct $a, b \in [n]$.

Competing Methods. `Orient`, which refers to the structure proposed in Section 6.2, stores

- a hash table on each S_i ($i \in [n]$);
- an array where the elements $e \in S_i^+$ ($i \in [n]$) are sorted by the x-coordinate of p_e;
- for each $1 \leq i < j \leq n$, an array storing the elements of $S_i^- \cap S_j^-$ that contribute to the skyline of $\{p_e \mid e \in S_i \cap S_j\}$. The elements e in the array are sorted by the x-coordinate of p_e.

We compared the method with `Hash`, which builds only a hash table on each S_i ($1 \leq i \leq n$). To answer a query with set IDs a and b, `Hash` first finds $S_a \cap S_b$ in the way described in Section 7.1 and then computes the skyline of $\{p_e \mid e \in S_a \cap S_b\}$ using an algorithm in [41]. We also examined `Orient+`, which builds our structure of Section 6.2 only on the colossal sets in the dataset and preprocesses the tail sets like `Hash`. For all methods, the 2D points p_e are stored only once in an array — the *map* — that, given e, permits fetching p_e in constant time.

Data. We used the same set collections as before, namely IMDB, Google, and Amazon. However, we augmented these datasets by associating each element $e \in \mathbb{D}$ with a 2D point, which was independently generated according to the anti-correlated distribution described in [11].

Workloads. The same workloads as those in Section 7.1 were deployed.

Results. Figure 10(a) shows the preprocessing time for all methods, while Figure 10(b) compares their space consumption, with a detailed breakdown in Figure 10(c). The meaning of all columns in the table should be clear to the reader. It is worth noting that the space usage of `Orient` and `Orient+` was lower than what was reported in Figure 5(c) because only part of $S_i^- \cap S_j^-$ needs to be stored (the part contributing to the skyline of $\{p_e \mid e \in S_i \cap S_j\}$) for $1 \leq i < j \leq n$.

Regarding query performance, Figure 11 shows the query time of each method as a function of small set size, using a style similar to that in Figure 6. The relative performance of the methods was consistent with that presented in Figure 6.

7.4 SIPF-Quantile

The last set of experiments was designed to evaluate the index proposed in Section 6.3 for the SIPF-quantile problem. Let $\{S_1, ..., S_n\}$ be the input set collection and define $D = \bigcup_{i=1}^{n} S_i$. Each element $e \in D$ is associated with a 1D point $p_e \in \mathbb{R}$; a query returns the k-quantiles of $\{p_e \mid e \in S_a \cap S_b\}$ for some specified $a, b \in [n]$, where the integer k is fixed for all queries.

Competing Methods. `Orient`, the SIPF-quantile structure in Section 6.3, stores

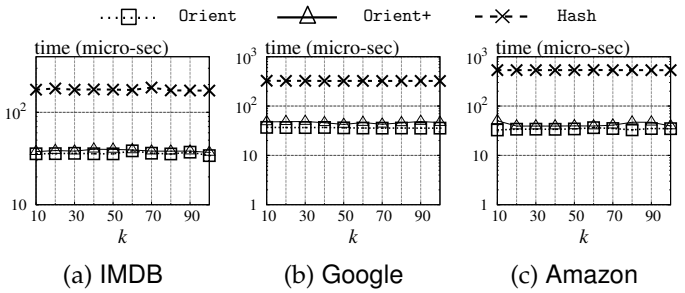


Fig. 13. Query time vs. k (SIPF-quantile)

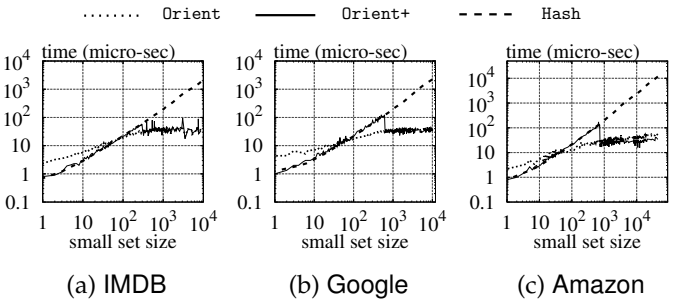


Fig. 14. Query time vs. small set size (SIPF-quantile)

- a hash table on each S_i ($i \in [n]$);
- an array where the elements $e \in S_i^+$ ($i \in [n]$) are sorted by p_e;
- for each $1 \leq i < j \leq n$, an array storing $\min\{|S_i^- \cap S_j^-|, k\}$ elements, plus an additional rank value per element (see Section 6.3). The elements e in the array are sorted by p_e. As a heuristic, we can reduce the space when $k \geq |S_i \cap S_j|$, observing that in this scenario computing the k-quantiles of $\{p_e \mid e \in S_i \cap S_j\}$ is reduced to finding $\{p_e \mid e \in S_i \cap S_j\}$. Specifically, if $k \geq |S_i \cap S_j|$, we stored only $S_i^- \cap S_j^-$, saving the space of rank values.

We compared our method with `Hash`, which builds only a hash table on each S_i ($1 \leq i \leq n$). To answer a query with set IDs a and b, `Hash` first finds $S_a \cap S_b$ in the way described in Section 7.1 and then computes the k-quantiles of $\{p_e \mid e \in S_a \cap S_b\}$ with sorting. We also examined `Orient+`, which builds our structure of Section 6.2 only on the colossal sets and preprocesses the tail sets like `Hash`. For all methods, the 1D points p_e are stored only once in an array.

Data and Workloads. We used exactly the same datasets and workloads as in Section 7.2.

Results. Figure 12(a) shows the preprocessing time for all methods, while Figures 12(b)-12(d) present their space consumption as a function of k. The space usage was similar to what was presented in Figure 7 (and hence, we omit a detailed breakdown) and changed only slightly with k. It is worth noting that the space of `Orient` and `Orient+` did not increase with k due to the heuristic mentioned earlier.

In the next experiment, we examined the effect of varying k on query performance. For each method, we measured its average cost for answering all queries in a workload under different values of k. Figure 13 shows the workload average as a function of k for all methods. It turned out that the parameter had little impact on the query cost. `Orient` was slightly faster than `Orient+`, as is consistent with the trends in Figure 8.

Finally, in Figure 14, we fixed k to 50 and examined the influence of a query's small-set size following the approach of in Figure 6. We observed the same trends as in Figure 6.

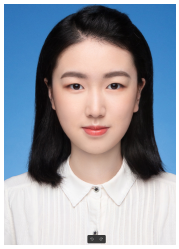
8 Conclusions

A set intersection query allows a user to designate two arbitrary sets from a set collection and computes their intersection. While such queries frequently arise in database applications, in practice users are rarely interested in the raw intersection result, but instead typically need to apply certain post-filtering to it. Motivated by this, in the current paper we develop an indexing scheme that can efficiently support a wide variety of filtering functions. The scheme is easy to understand and implement, yet it has solid theoretical guarantees sensitive to the "pseudoarboricity" of the underlying set collection, which is a new concept developed in this paper to quantify the density of a set collection. Our theoretical findings are confirmed by an empirical evaluation using real-world data.

References

[1] P. Afshani and T. M. Chan, "Optimal halfspace range reporting in three dimensions," in *SODA*, 2009, pp. 180–186.
[2] P. Afshani and J. S. Nielsen, "Data structure lower bounds for document indexing problems," in *ICALP*, vol. 55, 2016, pp. 93:1–93:15.
[3] A. Anand, S. J. Bedathur, K. Berberich, and R. Schenkel, "Efficient temporal keyword search over versioned text," in *CIKM*, 2010, pp. 699–708.
[4] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin, "Efficient parallel lists intersection and index compression algorithms using graphics processing units," *PVLDB*, vol. 4, no. 8, pp. 470–481, 2011.
[5] R. A. Baeza-Yates, "A fast set intersection algorithm for sorted sequences," in *CPM*, 2004, pp. 400–408.
[6] J. Barbay and C. Kenyon, "Alternation and redundancy analysis of the intersection problem," *ACM Transactions on Algorithms*, vol. 4, no. 1, pp. 4:1–4:18, 2008.
[7] J. L. Bentley and A. C. Yao, "An almost optimal algorithm for unbounded searching," *IPL*, vol. 5, no. 3, pp. 82–87, 1976.
[8] P. Bille, A. Pagh, and R. Pagh, "Fast evaluation of union-intersection expressions," in *ISAAC*, 2007, pp. 739–750.
[9] J. Blanuša, R. Stoica, P. Ienne, and K. Atasu, "Many-core clique enumeration with fast set intersections," *PVLDB*, vol. 13, no. 12, pp. 2676–2690, 2020.
[10] M. Blumenstock, "Fast algorithms for pseudoarboricity," in *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2016, pp. 113–126.
[11] S. Borzsonyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001, pp. 421–430.
[12] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with roaring bitmaps," *Softw. Pract. Exp.*, vol. 46, no. 5, pp. 709–719, 2016.
[13] T. Chan, "A minimalist's implementation of the 3-d divide-and-conquer convex hull algorithm," *Technical report, University of Waterloo*, 2003.
[14] G. Chen, J. Zhao, Y. Gao, L. Chen, and R. Chen, "Time-aware boolean spatial keyword queries (extended abstract)," in *ICDE*, 2018, pp. 1781–1782.
[15] L. Chen, Y. Cui, G. Cong, and X. Cao, "SOPS: A system for efficient processing of spatial-keyword publish/subscribe," *PVLDB*, vol. 7, no. 13, pp. 1601–1604, 2014.
[16] L. Chen, S. Shang, C. Yang, and J. Li, "Spatial keyword search: a survey," *GeoInformatica*, vol. 24, no. 1, pp. 85–106, 2020.
[17] Y. Chen and W. Shen, "An efficient method to evaluate intersections on big data sets," *Theo. Comp. Sci.*, vol. 647, pp. 1–21, 2016.
[18] Y.-Y. Chen, T. Suel, and A. Markowetz, "Efficient query processing in geographic web search engines," in *SIGMOD*, 2006, pp. 277–288.

[19] Z. Chen, L. Chen, G. Cong, and C. S. Jensen, "Location- and keyword-based querying of geo-textual data: a survey," *VLDB J.*, vol. 30, no. 4, pp. 603–640, 2021.
[20] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. of Comp.*, vol. 14, no. 1, pp. 210–223, 1985.
[21] H. Cohen and E. Porat, "Fast set intersection and two-patterns matching," *Theo. Comp. Sci.*, vol. 411, no. 40-42, pp. 3795–3800, 2010.
[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.
[23] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Springer-Verlag, 2008.
[24] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Adaptive set intersections, unions, and differences," in *SODA*, 2000, pp. 743–752.
[25] B. Ding and A. C. Konig, "Fast set intersection in memory," *PVLDB*, vol. 4, no. 4, pp. 255–266, 2011.
[26] D. Eppstein, "Arboricity and bipartite subgraph listing algorithms," *IPL*, vol. 51, no. 4, pp. 207–211, 1994.
[27] D. Eppstein, M. T. Goodrich, M. Mitzenmacher, and M. R. Torres, "2-3 cuckoo filters for faster triangle listing and set intersection," in *PODS*, 2017, pp. 247–260.
[28] I. D. Felipe, V. Hristidis, and N. Rishe, "Keyword search on spatial databases," in *ICDE*, 2008, pp. 656–665.
[29] I. Goldstein, T. Kopelowitz, M. Lewenstein, and E. Porat, "Conditional lower bounds for space/time tradeoffs," in *WADS*. Springer, 2017, pp. 421–436.
[30] I. Goldstein, M. Lewenstein, and E. Porat, "On the hardness of set disjointness and set intersection with bounded universe," in *ISAAC*, 2019, pp. 7:1–7:22.
[31] M. T. Goodrich, "Answering spatial multiple-set intersection queries using 2-3 cuckoo hash-filters," *CoRR*, vol. abs/1708.09059, 2017.
[32] G. Guzun and G. Canahuate, "Hybrid query optimization for hard-to-compress bit-vectors," *VLDB J.*, vol. 25, no. 3, pp. 339–354, 2016.
[33] S. Han, L. Zou, and J. X. Yu, "Speeding up set intersections in graph algorithms using SIMD instructions," in *SIGMOD*, 2018, pp. 1587–1602.
[34] F. K. Hwang and S. Lin, "A simple algorithm for merging two disjoint linearly-ordered sets," *SIAM J. of Comp.*, vol. 1, no. 1, pp. 31–39, 1972.
[35] H. Inoue, M. Ohara, and K. Taura, "Faster set intersection with SIMD instructions by reducing branch mispredictions," *PVLDB*, vol. 8, no. 3, pp. 293–304, 2014.
[36] S. Kim, J. Lee, S. R. Satti, and B. Moon, "SBH: super byte-aligned hybrid bitmap compression," *Information Systems*, vol. 62, pp. 155–168, 2016.
[37] S. Kim, T. Lee, S. Hwang, and S. Elnikety, "List intersection for web search: Algorithms, cost models, and optimizations," *PVLDB*, vol. 12, no. 1, pp. 1–13, 2018.
[38] T. Kopelowitz, S. Pettie, and E. Porat, "Dynamic set intersection," in *WADS*, 2015, pp. 470–481.
[39] ——, "Higher lower bounds from the 3sum conjecture," in *SODA*, 2016, pp. 1272–1287.
[40] T. Kopelowitz and V. V. Williams, "Towards optimal set-disjointness and set-intersection data structures," in *ICALP*, vol. 168, 2020, pp. 74:1–74:16.
[41] H. T. Kung, F. Luccio, and F. P. Preparata, "On finding the maxima of a set of vectors," *JACM*, vol. 22, no. 4, pp. 469–476, 1975.
[42] H. Lang, A. Beischl, V. Leis, P. A. Boncz, T. Neumann, and A. Kemper, "Tree-encoded bitmaps," in *SIGMOD*, 2020, pp. 937–967.
[43] T. Lee, J. Park, S. Lee, S. Hwang, S. Elnikety, and Y. He, "Processing and optimizing main memory spatial-keyword queries," *PVLDB*, vol. 9, no. 3, pp. 132–143, 2015.
[44] D. Lemire, L. Boytsov, and N. Kurz, "SIMD compression and the intersection of sorted integers," *Softw. Pract. Exp.*, vol. 46, no. 6, pp. 723–749, 2016.
[45] D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. S. Y. Kai, "Roaring bitmaps: Implementation of an optimized software library," *Softw. Pract. Exp.*, vol. 48, no. 4, pp. 867–895, 2018.
[46] Z. Li, K. C. K. Lee, B. Zheng, W. Lee, D. L. Lee, and X. Wang, "IR-tree: An efficient index for geographic document search," *TKDE*, vol. 23, no. 4, pp. 585–599, 2011.
[47] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *JACM*, vol. 30, no. 3, pp. 417–427, 1983.
[48] M. Patrascu, "Towards polynomial lower bounds for dynamic problems," in *STOC*, 2010, pp. 603–610.
[49] W. Rao, L. Chen, P. Hui, and S. Tarkoma, "Bitlist: New full-text index for low space cost and efficient keyword search," *PVLDB*, vol. 6, no. 13, pp. 1522–1533, 2013.
[50] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel, "Compression of inverted indexes for fast query evaluation," in *SIGIR*, 2002, pp. 222–229.
[51] Y. Tao and C. Sheng, "Fast nearest neighbor search with keywords," *TKDE*, vol. 26, no. 4, pp. 878–888, 2014.
[52] S. Tatikonda, F. Junqueira, B. B. Cambazoglu, and V. Plachouras, "On efficient posting list intersection with multicore processors," in *SIGIR*, 2009, pp. 738–739.
[53] D. Tsirogiannis, S. Guha, and N. Koudas, "Improving the performance of list intersection," *PVLDB*, vol. 2, no. 1, pp. 838–849, 2009.
[54] Q. Wang and T. Suel, "Document reordering for faster intersection," *PVLDB*, vol. 12, no. 5, pp. 475–487, 2019.
[55] D. Wu, G. Cong, and C. S. Jensen, "A framework for efficient spatial web object retrieval," *VLDB J.*, vol. 21, no. 6, pp. 797–822, 2012.
[56] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen, "Joint top-k spatial keyword query processing," *TKDE*, vol. 24, no. 10, pp. 1889–1903, 2012.
[57] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *TODS*, vol. 31, no. 1, pp. 1–38, 2006.
[58] H. Yan, S. Ding, and T. Suel, "Inverted index compression and query processing with optimized document ordering," in *WWW*, 2009, pp. 401–410.
[59] C. Zhang, Y. Zhang, W. Zhang, and X. Lin, "Inverted linear quadtree: Efficient top K spatial keyword search," *TKDE*, vol. 28, no. 7, pp. 1706–1721, 2016.
[60] J. Zhang, Y. Lu, D. G. Spampinato, and F. Franchetti, "FESIA: A fast and simd-efficient set intersection approach on modern cpus," in *ICDE*, 2020, pp. 1465–1476.
[61] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz, "Super-scalar RAM-CPU cache compression," in *ICDE*, 2006, p. 59.



Ru Wang obtained her B.S. degree in Computer Science and Technology from Fudan University in 2021. She is currently a PhD student at the Department of Computer Science and Engineering, The Chinese University of Hong Kong.

Shangqi Lu obtained his Ph.D. from the Department of Computer Science and Engineering, The Chinese University of Hong Kong, in 2022. His research focuses on algorithms and data structures in database systems with nontrivial theoretical guarantees.

Yufei Tao is a professor in the Department of Computer Science and Engineering, The Chinese University of Hong Kong. He served as the PC chair of PODS 2020, a PC co-chair of ICDE 2014, an associate editor of ACM TODS (2008-2015), an associate editor of IEEE TKDE (2012-2014), and an associate editor of Computer Science Review (2021-now). He was elected an ACM Fellow in 2020.