Maximizing the Optimality Streak of Deferred Data Structuring (a.k.a. Database Cracking)

Yufei Tao
The Chinese University of Hong Kong, China

Abstract

This paper studies how to minimize the total cost of answering r queries over n elements in an online manner (i.e., the next query is given only after the previous query's result is ready) when the value $r \leq n$ is unknown in advance. Traditional indexing, which first builds a complete index on the n elements before answering queries, may be unsuitable because the index's construction time – usually $\Omega(n \log n)$ – can become the performance bottleneck. In contrast, for many problems, a lower bound of $\Omega(n \log(1+r))$ holds on the total cost of r queries for every $r \in [1, n]$. Matching this lower bound is a primary objective of *deferred data structuring* (DDS), also known as *database cracking* in the system community. For a wide class of problems, we present generic reductions to convert traditional indexes into DDS algorithms that match the lower bound for a long range of r. For a decomposable problem, if a data structure can be built in $O(n \log n)$ time and has $Q(n)$ query search time, our reduction yields an algorithm that runs in $O(n \log(1+r))$ time for all $r \leq \frac{n \log n}{Q(n)}$, where the upper bound $\frac{n \log n}{Q(n)}$ is asymptotically the best possible under mild constraints. In particular, if $Q(n) = O(\log n)$, then the $O(n \log(1+r))$-time guarantee extends to all $r \leq n$, with which we optimally settle a large variety of DDS problems. Our results can be generalized to a class of "spectrum indexable problems", which subsumes the class of decomposable problems.

2012 ACM Subject Classification Theory of computation → Data structures and algorithms for data management

Keywords and phrases Deferred Data Structuring, Database Cracking, Data Structures

Digital Object Identifier 10.4230/LIPIcs.ICDT.2025.10

Funding This work was partially supported by GRF projects 4203421 and 14222822 from HKRGC.

1 Introduction

Traditional indexing first creates an index on the whole dataset before starting to answer queries. For example, after building a binary search tree (BST) on an (unsorted) set S of n real values in $O(n \log n)$ time, we can use the tree to answer each predecessor query[1] in $O(\log n)$ time. This paradigm, however, falls short when the dataset S will only be searched a small number of times. In the extreme, if only one query needs to be answered, the best approach is to scan S in full, which requires only $O(n)$ time. More generally, if we are to answer r queries in an *online* manner – that is, the next query is given after the result of the previous query has been output – it is possible to pay a total cost of $O(n \log(1+r))$ [19]. When $r \ll n$ (e.g., $r = \text{polylog}\, n$ or $2^{\sqrt{\log n}}$), the cost $O(n \log(1+r))$ breaks the barrier of $\Omega(n \log n)$, suggesting that sorting is unnecessary.

Situations like the above occur in many big-data applications where large volumes of data are collected but only sparingly queried. The phenomenon has motivated a line of research under the name *database cracking* in the system community; see [12, 13, 16–18, 23, 29, 31, 32] and the references therein. In these environments, the problem size n is huge such that even sorting is considered expensive and should be avoided as much as possible. Furthermore, it is

[1] Given a search value q, a predecessor query returns the largest element in S that does not exceed q.

impossible to predict how many queries – namely, the integer r mentioned earlier – will need to be supported. Instead of constructing a complete index right away, database cracking carries out only a calculated portion of the construction during each query. If r eventually reaches a certain threshold, the index will be created in full, but it is more likely that r will stop at some value significantly lower than that threshold. The challenge of database cracking is to ensure "smoothness": as r grows, the total cost of all the r queries *so far* ought to increase as slowly as possible.

In the theory field, the data-structure problems at the core of database cracking had already been studied in 1988 by Karp, Motwani, and Raghavan [19] under the name *deferred data structuring* (DDS). For a selection of problems, they explained how to answer $r \in [1, n]$ queries on n elements with a total cost of $O(n \log(1+r))$, without knowing r in advance. For every $r \in [1, n]$, they proved a matching lower bound of $\Omega(n \log(1+r))$ on those problems. Through reductions, the same lower bound has been shown to hold on many other DDS problems.

This work explores the following topic: how to design a generic reduction that, given a (conventional) data structure, can turn it *automatically* into an efficient DDS algorithm? Such reductions may significantly simplify the design of DDS algorithms and shed new light on the intricate connections between traditional indexing and DDS.

Math Conventions. We use $\mathbb{N}^+$ to denote the set of positive integers. For any integer $x \geq 1$, let $[x]$ represent the set $\{1, 2, ..., x\}$. Every logarithm has base 2 by default. Define $\text{Log}(x) = \log(1+x)$ for any $x \geq 1$.

1.1 Problem Definitions

This subsection will formalize the problems to be investigated. Let S, called the *dataset*, be a set of n elements drawn from a domain $\mathbb{D}$. Let $\mathbb{Q}$ be a (possibly infinite) set where the elements are called *predicates*. Given a predicate $q \in \mathbb{Q}$, a *query* issued on S returns an *answer*, represented as $\text{Ans}_q(S)$. We consider that, for any $q \in \mathbb{Q}$, the answer $\text{Ans}_q(S)$ can be represented using $O(1)$ words and can be computed in $O(n)$ time (which essentially means that the query can be answered using a brute-force algorithm such as exhaustive scan).

Deferred Data Structuring (DDS). We now formalize the DDS problem. Initially, an algorithm $\mathcal{A}$ is provided with the dataset S in an array, where the elements are arbitrarily permuted. An adversary first chooses a predicate q_1 for the first query, and solicits the answer $\text{Ans}_{q_1}(S)$ from $\mathcal{A}$. Iteratively, for each $i \geq 2$, after the answer of the $(i-1)$-th query has been obtained, the adversary either decides to terminate the whole process, or chooses the predicate q_i for the next query and solicits $\text{Ans}_{q_i}(S)$ from $\mathcal{A}$. The adversary is permitted to observe the execution of $\mathcal{A}$ and, thus, capable of selecting a "bad" q_i for $\mathcal{A}$.

The algorithm $\mathcal{A}$ is said to guarantee *running time* $\text{Time}(n, r)$ for t queries if, for *every* $r \leq t$, the first r queries are processed with a total cost at most $\text{Time}(n, r)$. We will concentrate on $t \leq n$ because this is the scenario important for database cracking.

Streak of Optimality. While the above setup is "standard" for DDS, as far as database cracking is concerned, it makes sense to carry out investigation from another fresh perspective. As database cracking is useful mainly in the scenario where the dataset receives relatively few queries, it is imperative to design DDS algorithms that are particularly efficient when the number of queries is small.

To formalize the notion of "particularly efficient", we leverage the fact that, for many DDS problems (including the ones considered in this work), there exist hardness barriers dictating $\textsc{Time}(n, r) = \Omega(n \operatorname{Log} r)$ for *every* $r \in [1, n]$ under a relevant computation model. Motivated by this, we say that an algorithm $\mathcal{A}$ guarantees a $\log(r)$-*streak* of $\textsc{LogStreak}(n)$ if its running time satisfies $\textsc{Time}(n, r) = O(n \operatorname{Log} r)$ for all $r \leq \textsc{LogStreak}(n)$.

The worst $\log(r)$-streak guarantee is $\textsc{LogStreak}(n) = O(1)$ – this is trivial because any query can be answered in $O(n)$ time. Ideally, we would like to have $\textsc{LogStreak}(n) = n$, but this is not always possible as argued later in the paper. For practical use, however, it would suffice for an algorithm to ensure $\textsc{LogStreak}(n) = \Omega(n^\epsilon)$ for some constant $\epsilon > 0$ because $\Omega(n^\epsilon)$ queries are perhaps already too many for database cracking to be appealing.

Classes of Problems. The above definition framework can be specialized into various problem instances that differ in the data domain $\mathbb{D}$ or the predicate domain $\mathbb{Q}$. Next, we introduce two problem classes relevant to our discussion:

- A problem instance is *decomposable* if, for any disjoint sets $S_1, S_2 \subseteq \mathbb{D}$ and any predicate $q \in \mathbb{Q}$, it is possible to derive $\textsc{Ans}_q(S_1 \cup S_2)$ from $\textsc{Ans}_q(S_1)$ and $\textsc{Ans}_q(S_2)$ in constant time.
- We define a problem instance to be $(B(n), Q(n))$ *spectrum indexable* if the following property holds on every dataset $S \subseteq \mathbb{D}$: for every integer $s \in [|S|]$, it is possible to construct a data structure on S in $O(|S| \cdot B(s))$ time that can answer any query in $O(\frac{|S|}{s} \cdot Q(s))$ time. The term "spectrum indexable" is chosen to reflect the ability to build a "good" index structure – as far as functions $B(n)$ and $Q(n)$ are concerned – for the whole spectrum of the parameter s.

Two observations are important about the above definitions:

- $(B(n), Q(n))$ *spectrum indexability* implies that we can build a data structure on any dataset $S \subseteq \mathbb{D}$ in $O(|S| \cdot B(|S|))$ time to answer a query in $O(Q(|S|))$ time (for this purpose, simply set $s = |S|$).
- Consider any decomposable problem instance with the following property: for any dataset $S \subseteq \mathbb{D}$, we can build a data structure $\mathcal{T}$ in $O(|S| \cdot B(|S|))$ time to answer a query in $O(Q(|S|))$ time. Then, the problem instance must be $(B(n), Q(n))$ spectrum indexable. To see why, given an integer $s \in [|S|]$, divide S arbitrarily into $m = \lceil |S|/s \rceil$ disjoint subsets $S_1, S_2, ..., S_m$ where all subsets have size s except S_m. For each $i \in [m]$, create a structure $\mathcal{T}(S_i)$ in $O(|S_i| \cdot B(s))$ time; the total time to create all the m structures is $O(m \cdot s \cdot B(s)) = O(|S| \cdot B(s))$. To answer a query q, simply search every $\mathcal{T}(S_i)$ to obtain $\textsc{Ans}_q(S_i)$ in $O(Q(s))$ time and then combine $\textsc{Ans}_q(S_1), \textsc{Ans}_q(S_2), ..., \textsc{Ans}_q(S_m)$ into $\textsc{Ans}_q(S)$ using $O(m)$ time. The total query time is therefore $O(m \cdot Q(s))$.

1.2 Related Work

Motwani and Raghavan introduced the concept of DDS in a conference paper [24], which together with Karp they extended into a journal article [19]. The article [19] presented algorithms for the following DDS problems:

- *Predecessor search*, where S consists of n real values, and each query is given an arbitrary value q and returns the predecessor of q in S.
- *Halfplane containment*, where S consists of n halfplanes in $\mathbb{R}^2$, and each query is given an arbitrary point $q \in \mathbb{R}^2$ and returns whether q is covered by all the halfplanes in S.
- *Convex hull containment*, where S consists of n points in $\mathbb{R}^2$, and each query is given an arbitrary point $q \in \mathbb{R}^2$ and returns whether q is covered by the convex hull of S.

- *2D linear programming*, where S consists of n halfplanes in $\mathbb{R}^2$, and each query is given a 2D vector $\boldsymbol{u}$ and returns the point p in the intersection of all the n halfplanes that maximizes the dot product $\boldsymbol{u} \cdot \boldsymbol{p}$.
- *Orthogonal range counting*, where S consists of n points in $\mathbb{R}^d$ with the dimensionality d being a fixed constant, and a query is a given an arbitrary d-rectangle q – namely, an axis-parallel box the form $[x_1, y_1] \times [x_2, y_2] \times ... \times [x_d, y_d]$ – and returns the number of points of S that are covered by q.

For the first four problems, Karp, Motwani, and Raghavan presented algorithms achieving $\textsc{Time}(n, r) = O(n \operatorname{Log} r)$ for all $r \leq n$. For orthogonal range counting, they presented two algorithms, the first of which ensures $\textsc{Time}(n, r) = O(n \operatorname{Log}^d r)$ for all $r \leq n$, whereas the other one ensures $\textsc{Time}(n, r) = O(n \log n + n \operatorname{Log}^{d-1} r)$ for all $r \leq n$. For all these problems, they proved that, under the comparison model and/or the algebraic model, the running time of any algorithm must satisfy $\textsc{Time}(n, r) = \Omega(n \operatorname{Log} r)$ for every $r \in [1, n]$.

Aggarwal and Raghavan [1] presented a DDS algorithm with $\textsc{Time}(n, r) = O(n \operatorname{Log} r)$ for all $r \leq n$ for *nearest neighbor search*, where S consists of n points in $\mathbb{R}^2$, and a query is given an arbitrary point $q \in \mathbb{R}^2$ and returns the point in S closest to q. This running time is optimal under the algebraic model for all $r \leq n$.

A "success story" can be told about DDS on *range median*. In the problem's offline version, we are given a set S of n real values in an (unsorted) array A. For any $1 \leq x \leq y \leq n$, let $A[x : y]$ represent the set of elements $\{A[x], A[x+1], ..., A[y]\}$. In addition, we are given r integer pairs $(x_1, y_1), (x_2, y_2), ..., (x_r, y_r)$ such that $1 \leq x_i \leq y_i \leq n$ for each $i \in [n]$. The goal is to find the median of the set $A[x_i : y_i]$ for all $i \in [r]$. In [15], Har-Peled and Muthukrishnan explained how to solve the problem in $O(n \operatorname{Log} r + r \log n \cdot \operatorname{Log} r)$ time and proved a lower bound of $\Omega(n \operatorname{Log} r)$ under the comparison model for all $r \leq n$. In [11] (see also [5]), Gfeller and Sanders considered the problem's DDS version, where S is as defined earlier, and a query is given an arbitrary pair (x, y) with $1 \leq x \leq y \leq n$ and returns the median of $A[x : y]$. They designed an algorithm achieving $\textsc{Time}(n, r) = O(n \operatorname{Log} r)$ for all $r \leq n$. It is easy to see that any DDS algorithm can be utilized to solve also the offline problem in $\textsc{Time}(n, r)$ time. Hence, the algorithm of Gfeller and Sanders improves the offline solution of [15] and is optimal (for DDS) under the comparison model.

More remotely related to our work is [8], where Ching, Mehlhorn, and Smid considered a dynamic DDS problem where, besides queries, an algorithm is also required to support updates on the dataset S. Towards another direction, Barbay et al. [2] studied the DDS version of predecessor search but analyzed their algorithm using more fine-grained parameters called "gaps" (rather than using only n and r). This direction has also been extended to dynamic DDS; see the recent works [26, 27].

As mentioned, DDS has been extensively studied in the system community under the name database cracking. The focus there is to engineer efficient heuristics to accelerate query workloads encountered in various practical scenarios, rather than establishing strong theoretical guarantees. Interested readers may refer to the representative works of [12, 13, 16–18, 23, 31, 32] as entry points into the literature.

1.3 Our Results

Our main result is a generic reduction with the following guarantee:

▶ **Theorem 1.** *Suppose that $B(n)$ and $Q(n)$ are non-decreasing functions such that $B(n) = O(\log n)$ and $Q(n) = O(n^{1-\epsilon})$ where $\epsilon > 0$ is an arbitrarily small constant. Every $(B(n), Q(n))$ spectrum indexable problem admits a DDS algorithm with*

$$\textsc{LogStreak}(n) = \min\left\{n, \frac{c \cdot n \log n}{Q(n)}\right\} \quad (1)$$

for an arbitrarily large constant c, namely, the algorithm achieves $\textsc{Time}(n, r) = O(n \cdot \text{Log}\, r)$ *for all* $r \leq \min\{n, \frac{c \cdot n \log n}{Q(n)}\}$.

Under mild constraints, the streak bound in (1) is the best possible, as we argue in Section 3. As an important special case, when $Q(n) = O(\log n)$, the DDS algorithm produced by our reduction achieves $\textsc{LogStreak}(n) = n$, namely, $\textsc{Time}(n,\, r) = O(n \cdot \text{Log}\, r)$ for all $r \leq n$. For many decomposable problems with high importance to database systems, data structures with $O(n \log n)$ construction time and $O(\log n)$ search time are already known (e.g., predecessor search and nearest neighbor search). As such problems must be $(\text{Log}\, n, \text{Log}\, n)$ spectrum indexable (as explained in Section 1.1), Theorem 1 immediately produces excellent solutions to the DDS versions of all those problems, which are usually optimally under the comparison/algebraic model. A selection of those problems will be given in Section 4.1.

Theorem 1 has a pleasant message for database cracking: even data structures with slow query time can be useful for cracking! For example, for orthogonal range counting (defined in Section 1.2) in 2D space, the kd-tree, which takes $O(n \log n)$ time to build, answers a query in $Q(n) = O(\sqrt{n})$ time. Theorem 1 shows that the structure can be utilized to answer any $r = O(\sqrt{n} \log n)$ queries in $O(n \,\text{Log}\, r)$ time, which is probably more than enough for database cracking in reality. This nicely manifests our motivation (stated in Section 1.1) for studying "DDS algorithms particularly efficient for small r".

Theorem 1 has an instructive implication to the *design* of DDS algorithms – we should study *how spectrum indexable* the underlying problem really is. Exploration in this direction can get fairly interesting, as we will demonstrate in Section 4.2 for halfplane containment, convex hull containment, range median, and 2D linear programming (see Section 1.2 for their definitions). We can prove that each of those problems is $(\text{Log}\, n, Q(n))$ spectrum indexable for an appropriately selected function $Q(n)$, and then leverage Theorem 1 to obtain an algorithm solving it with $\textsc{Time}(n, r) = O(n \,\text{Log}\, r)$ for all $r \leq n$.

What happens to structures that take $\omega(n \log n)$ time to build, or equivalently, $B(n) = \omega(\log n)$? Section 5 will show that, under mild constraints, no generic reductions can use such a structure to obtain DDS algorithms with $\textsc{LogStreak}(n) = \omega(1)$. In other words, these algorithms can achieve $\textsc{Time}(n, r) = O(n \,\text{Log}\, r)$ only in the trivial scenario where $r = O(1)$. Nevertheless, if one accepts algorithms with guarantees of the form "$\textsc{Time}(n, r) \leq n \,\text{Log}^{O(1)}\, r$ for all $r \leq n$", our reduction underlying Theorem 1 can be extended to produce such algorithms as long as $B(n)$ and $Q(n)$ are $\text{Log}^{O(1)}\, n$, as will be discussed in Section 5.

2 Warm Up: Predecessor Search

Predecessor search is the problem that has received the most attention from the DDS and database-cracking communities. This section will review two approaches developed in [19] for achieving $\textsc{Time}(n, r) = O(n \,\text{Log}\, r)$ on this problem. These approaches form the basis of nearly all the existing DDS algorithms.

Bottom-Up. Recall that the dataset S consists of n real values. We assume, w.l.o.g., that n is a power of 2. At all times, the set S is arbitrarily partitioned into disjoint subsets – referred to as *runs* – each having the same size $s = 2^i$ for some $i \geq 0$. Every run is sorted and stored in an array. Initially, $s = 1$, i.e., a run contains an individual element of S. Over time, the run size s increases monotonically. Whenever s needs to go from 2^i to 2^j for some value $j > i$, an *overhaul* is carried out to build the new runs. As a run of size 2^j can be

obtained by merging 2^{j-i} runs of size 2^i in $O(2^i \cdot (j-i))$ time, the overhaul can be completed in $O(n \cdot (j-i))$ time. Therefore, if the current run size is s, the overall cost of producing all the runs in history is $O(n \operatorname{Log} s)$.

A (predecessor) query is answered simply by performing binary search on every run. To control the cost, however, the algorithm makes sure that the current size s is at least $i \operatorname{Log} i$ before processing the i-th ($i \geq 1$) query. If the condition is not met, an overhaul is invoked to increase s to the nearest power of 2 at least $i \operatorname{Log} i$. After that, the query entails a cost of $O(\frac{n}{s} \operatorname{Log} s) = O(\frac{n}{i \operatorname{Log} i} \cdot \log(i \operatorname{Log} i)) = O(n/i)$ time. If we add this up for all r queries, the sum becomes $O(n \sum_{i=1}^{r} \frac{1}{i}) = O(n \operatorname{Log} r)$. As the final run size s is $O(r \operatorname{Log} r)$, we can conclude that the algorithm processes r queries in $O(n \operatorname{Log} r)$ time. Note that this holds only if $r \operatorname{Log} r \leq n$ (the maximize run size is n). However, when r has reached $\lceil n/\log n \rceil$, the algorithm can afford to sort the entire S in $O(n \log n) = O(n \log r)$ time and answer every subsequent query in $O(\log n) = O(\log r)$ time. Thus, the algorithm achieves $\textsc{Time}(n, r) = O(n \operatorname{Log} r)$ for all $r \leq n$.

Top-Down. This approach mimics the following strategy for building a binary search tree (BST) $\mathcal{T}$ on S: (i) find the median of S, and splits S into S_1 and S_2 at the median; (ii) store the median as the root's key, and then build the root's left (resp., right) subtree recursively on S_1 (resp., S_2). Rather than doing a full construction outright, the algorithm builds $\mathcal{T}$ on an "as-needed" basis during query processing.

In the beginning, only the root exists and it is put in the *unexpanded* mode. In general, an unexpanded node u has no children yet, but is associated with the subset $S_u \subseteq S$ of elements that ought to be stored in its subtree. A query is answered in the same manner as in a normal BST, by traversing a root-to-leaf path π of $\mathcal{T}$. The main difference is that as the search comes to an unexpanded node u on π, the algorithm must *expand* it first. If $|S_u| \geq 2$, expanding u means creating two child nodes for u, splitting S_u at the median (the key of u), dividing S_u at the median into two parts, and associating each part with a child. After that, u becomes *expanded* with its children put in the unexpanded mode. If, on the other hand, $|S_u| = 1$, expanding u simply means making u a leaf of $\mathcal{T}$ and marking u in the *expanded* mode. In any case, the expansion takes $O(|S_u|)$ time (finding the median of a set takes linear time [4]).

After r queries, the BST $\mathcal{T}$ is partially built because only the nodes on the r root-to-leaf paths traversed during query processing are expanded. The nodes at the first $\operatorname{Log} r$ levels[2] can incur a total expansion cost of $O(n \operatorname{Log} r)$. For each root-to-leaf path π, the node u at level $\operatorname{Log} r$ has expansion cost $O(n/r)$, which dominates the total expansion cost of the descendants of u on π. Therefore, other than the nodes at the first $\operatorname{Log} r$ levels, all the other nodes in $\mathcal{T}$ have an expansion cost of $r \cdot O(\frac{n}{r}) = O(n)$ in total. The algorithm therefore achieves $\textsc{Time}(n, r) = O(n \operatorname{Log} r)$ for all $r \leq n$.

3 Reductions from Data Structures to DDS Algorithms

Section 3.1 will extend the bottom-up approach (reviewed in the previous section) into a generic reduction, which can yield DDS algorithms with large $\log(r)$-streak bounds, provided that a crucial requirement – linear mergeability (to be defined shortly) – is met. Although this reduction will be superseded by our final reduction (presented in Section 3.2) underneath Theorem 1, its discussion (i) deepens the reader's understanding of the approach's power

[2] The *level* of a node is the number of edges on the path from the node to the root.

and limitations, and (ii) elucidates the necessity for new ideas in the absence of linear mergeability. Finally, Section 3.3 will establish a hardness result to show that the streak bound in Theorem 1 can no longer be improved significantly.

3.1 The First Reduction: Generalizing the Bottom-Up Approach

This subsection will focus on decomposable problems. For any dataset $S \subseteq \mathbb{D}$, we assume the existence of a data structure $\mathcal{T}(S)$ that can answer any query in $O(Q(n))$ time. Furthermore, the structure is *linearly mergeable*, namely, for any disjoint $S_1, S_2 \subseteq \mathbb{D}$, the structure on $S_1 \cup S_2$ can be constructed from $\mathcal{T}(S_1)$ and $\mathcal{T}(S_2)$ in $O(|S_1| + |S_2|)$ time. Note that this implies $\mathcal{T}(S)$ can be built in $O(n \log n)$ time.

▶ **Lemma 2.** *For a decomposable problem on which there is a linear-mergeable structure with query time $Q(n) = O(n^{1-\epsilon})$ where $\epsilon > 0$ is a constant, we can design a DDS algorithm to guarantee $\textsc{LogStreak}(n) = \min\{n, \frac{c \cdot n \log n}{Q(n)}\}$ for an arbitrarily large constant c.*

Proof. We assume, w.l.o.g., that n is a power of 2. As with the bottom-up approach, at any moment, we divide S arbitrarily into runs with the same size $s = 2^i$ for some $i \geq 0$. For each run, build a structure on the elements therein. The initial run size s is 1. Every time s grows from 2^i to 2^j for some value $j > i$, an *overhaul* is performed to construct the runs of size 2^j. By linear mergeability, we can build the structure of a size-2^j run by merging the structures of 2^{j-i} runs of size 2^i in $O(2^j \cdot (j - i))$ time. By an analysis similar to the one in Section 2, if the current run size is s, the overall cost of producing the structures for all the runs that ever appeared in history is $O(n \operatorname{Log} s)$.

A query with predicate q is answered by searching the structure of every run, and then combining the answers from all the runs into $\textsc{Ans}_q(S)$. The query cost is $O(\frac{n}{s} \cdot Q(s))$. We require that, before answering the i-th ($i \geq 1$) query, the run size s must satisfy

$$Q(s)/s \leq 1/i. \tag{2}$$

If this requirement is not met, we launch an overhaul to increase s to the least power of 2 fulfilling (2). This ensures that the i-th query is answered with a cost $O(n/i)$. Hence, the total cost of processing r queries is $O(n \operatorname{Log} r)$.

What remains is to bound the cost of the overhauls. The final run size s is the least power of 2 satisfying

- $s \leq n$ (the run size cannot exceed n) and
- $s/Q(s) \geq r$ (because of (2)).

Rather than solving s precisely, we instead aim to find an upper bound for it. As $Q(s) = O(s^{1-\epsilon})$, we know $Q(s) \leq \alpha \cdot s^{1-\epsilon}$ for some constant $\alpha > 0$. Hence, $s/Q(s) \geq s^\epsilon/\alpha$. Let $\hat{s}$ be the least power of 2 satisfying

$$\hat{s} \leq n \text{ and } \hat{s}^\epsilon/\alpha \geq r.$$

Since the condition $s^\epsilon/\alpha \geq r$ implies $s/Q(s) \geq r$, it holds that $\hat{s} \geq s$.

The condition $\hat{s}^\epsilon/\alpha \geq r$ can be rewritten as $\hat{s} \geq (\alpha \cdot r)^{1/\epsilon}$. Therefore, if $(\alpha \cdot r)^{1/\epsilon} \leq n/2$, then $\hat{s}$ is simply the least power of 2 at least $(\alpha \cdot r)^{1/\epsilon}$. This means $\hat{s} = O(r^{1/\epsilon})$ and thus $O(\operatorname{Log} s) = O(\operatorname{Log} \hat{s}) = O(\operatorname{Log} r)$, in which case all the overhauls require $O(n \operatorname{Log} s) = O(n \operatorname{Log} r)$ time overall.

The above argument does not work if $(\alpha \cdot r)^{1/\epsilon} > n/2$. However, in this case, $r > n^\epsilon/(2^\epsilon \alpha)$, that is, r is already a polynomial of n. This motivates the following brute-force strategy. When r reaches $\lceil n^\epsilon/(2^\epsilon \alpha) \rceil$ – a moment we call the *snapping point* – we simply create a structure on the whole S in $O(n \log n) = O(n \log r)$ time, and use it to answer every

subsequent query in $O(Q(n))$ time until $r = \min\{n, \frac{c \cdot n \log n}{Q(n)}\}$. All the queries after the snapping point have a total cost of $O(n \log n) = O(n \log r)$. We thus have obtained an algorithm that guarantees $\textsc{Time}(n, r) = O(n \operatorname{Log} r)$ for all $r \leq \min\{n, \frac{c \cdot n \log n}{Q(n)}\}$. ◀

The above reduction crucially relies on the fact that the structure $\mathcal{T}$ is linearly mergeable. Otherwise, the overhaul for creating size-2^i runs would become $\Omega(n \cdot \log(2^i))$, in which case all the overhauls would end up with a total cost of $\Omega(n \cdot \operatorname{Log}^2 r)$. Next, we will present another reduction that can shave a $\operatorname{Log} r$ factor.

3.2 The Second Reduction: No Linear Mergeability

We will now drop the linear mergeability requirement in Section 3.1 and establish Theorem 1. Recall that the underlying problem is $(B(n), Q(n))$ spectrum indexable with $B(n) = O(\log n)$ and $Q(n) = O(n^{1-\epsilon})$ for some constant $\epsilon > 0$. The goal is to design an algorithm with $\textsc{Time}(n, r) = O(n \operatorname{Log} r)$ for all $r \leq \min\{n, \frac{c \cdot n \log n}{Q(n)}\}$, where $c > 0$ can be any constant.

Assume, w.l.o.g., that n is a power of 2. Our algorithm executes in *epochs*. At the beginning of the i-th ($i \geq 1$) epoch, we set

$$s = 2^{2^i}.$$

For now, let us consider that s does not exceed n, a condition that will be guaranteed, as discussed later. The epoch starts by creating a structure $\mathcal{T}$ – the structure promised by $(B(n), Q(n))$ spectrum indexability – on S in $O(n \cdot B(s))$ time. The structure allows us to answer any query in $O(\frac{n}{s} \cdot Q(s))$ time. The i-th epoch finishes after $\lceil s/Q(s) \rceil$ queries[3] are answered *during* the epoch. These queries demand a total cost of

$$\left\lceil \frac{s}{Q(s)} \right\rceil \cdot O\Big(\frac{n}{s} \cdot Q(s)\Big) \quad = \quad O(n).$$

It is clear from the above that the total computation time of the i-th epoch is $O(n \cdot B(s)) = O(n \cdot 2^i)$. As $n \cdot 2^i$ doubles when i increases by 1, the overall cost of answering r queries is $O(n \cdot 2^h)$, where h is the number of epochs needed. Precisely, the value of h is the smallest integer $x \geq 1$ satisfying two conditions:

- **C1:** $2^{2^x} \leq n$ (the value of s must not exceed n);
- **C2:** $\sum_{i=1}^{x} \left\lceil \frac{2^{2^i}}{Q(2^{2^i})} \right\rceil \geq r$ (the number of queries that can be answered by h epochs must be at least r).

Rather than solving h precisely, we aim to find an upper bound for it. Define $\hat{h}$ to be the smallest integer $x \geq 1$ satisfying

- **C1':** $2^{2^x} \leq n$ (same as **C1**);
- **C2':** $2^{2^x}/Q(2^{2^x}) \geq r$.

It is clear that $\hat{h} \geq h$.

Still, we do not solve $\hat{h}$ precisely but instead will find an upper bound for it. For this purpose, we leverage the fact that $Q(n) = O(n^{1-\epsilon})$, which means $Q(n) \leq \alpha \cdot n^{1-\epsilon}$ for some constant $\alpha > 0$. Define H to be the smallest integer $x \geq 1$ satisfying

- **C1":** $2^{2^x} \leq n$ (same as **C1** and **C1'**);
- **C2":** $\frac{2^{2^x}}{\alpha \cdot (2^{2^x})^{1-\epsilon}} = \frac{(2^{2^x})^{\epsilon}}{\alpha} \geq r$, or equivalently, $2^{2^x} \geq (\alpha \cdot r)^{1/\epsilon}$.

[3] In practice, our algorithm may be slightly improved by making this number $\lceil s \cdot B(s)/Q(s) \rceil$, but this will not be necessary for the purpose of proving Theorem 1

Because condition **C2"** implies condition **C2'**, we must have $H \geq \hat{h} \geq h$.

Therefore, if $(\alpha \cdot r)^{2/\epsilon} \leq \sqrt{n}$ (namely, $r \leq n^{\epsilon/4}/\alpha$), the value of H is simply the least integer $x \geq 1$ satisfying $2^{2^x} \geq (\alpha \cdot r)^{1/\epsilon}$. This means

$$\begin{aligned} 2^{2^{H-1}} < (\alpha \cdot r)^{1/\epsilon} &\Leftrightarrow 2^{2^H} < (\alpha \cdot r)^{2/\epsilon} \\ &\Rightarrow 2^H = O(\text{Log}\, r). \end{aligned} \tag{3}$$

In this case, all epochs incur a total cost of $O(n \cdot 2^h) = O(n \cdot 2^H)$, which is $O(n \,\text{Log}\, r)$ by (3).

The above argument does not work if $(\alpha \cdot r)^{2/\epsilon} > \sqrt{n}$. However, when this happens, we must have $r > n^{\epsilon/4}/\alpha$. As soon as r reaches $\lceil n^{\epsilon/4}/\alpha \rceil$ – the *snapping point* – we create a structure $\mathcal{T}$ on the whole S in $O(n \log n) = O(n \log r)$ time, and use it to answer every subsequent query in $O(Q(n))$ time until $r = \min\{n, \frac{c \cdot n \log n}{Q(n)}\}$. The queries after the snapping point require a total cost of $O(n \log n) = O(n \log r)$. We thus have obtained an algorithm that guarantees $\text{Time}(n, r) = O(n \,\text{Log}\, r)$ for all $r \leq \min\{n, \frac{c \cdot n \log n}{Q(n)}\}$, completing the proof of Theorem 1.

3.3 Tightness of the Streak Bound

This section will explain why the streak bound $\Omega(\frac{n \log n}{Q(n)})$ in Theorem 1 is asymptotically the best possible for reduction algorithms with "reasonable" behavior.

Black-box Reductions on Decomposable Problems with Restricted Structures. For proving hardness results, we can *specialize* the problem class at will, and we do so by considering only decomposable problems. A reduction algorithm $\mathcal{A}$ is required to work on any decomposable problem that is $(B(n), Q(n))$ spectrum indexable. As explained in Section 1.1, this implies a data structure $\mathcal{T}$ that can be built on any $S \subseteq \mathbb{D}$ in $O(|S| \cdot B(|S|))$ time and answer any query on S in $O(Q(|S|))$ time.

As long as $(B(n), Q(n))$ spectrum indexability is retained, we can restrict the functionality of $\mathcal{T}$ to make it hard for $\mathcal{A}$. Specifically, $\mathcal{T}$ provides only the following "services" to $\mathcal{A}$:

- $\mathcal{A}$ can create a structure $\mathcal{T}(S')$ on any subset $S' \subseteq S$;
- given a predicate $q \in \mathbb{Q}$, $\mathcal{A}$ can use $\mathcal{T}(S')$ to find the answer $\text{Ans}_q(S')$ on S';
- given a predicate q, after $\mathcal{A}$ already has obtained $\text{Ans}_q(S'_1)$ and $\text{Ans}_q(S'_2)$ for disjoint subsets $S'_1, S'_2 \subseteq S$, it can combine the answers into $\text{Ans}_q(S'_1 \cup S'_2)$ in constant time (the combining algorithm is provided by $\mathcal{T}$).

Despite its limited functionalities, $\mathcal{T}$ still makes the problem $(B(n), Q(n))$ spectrum indexable because the problem is decomposable; see the explanation in Section 1.1.

So far no restriction has been imposed on the behavior of $\mathcal{A}$, but we are ready to do so now. To answer a query, the algorithm $\mathcal{A}$ needs to search the structures on a number (including zero) of subsets of S, say, $\mathcal{T}(S'_1), \mathcal{T}(S'_2), ..., \mathcal{T}(S'_t)$. The algorithm can choose any $t \geq 0$ and any $S'_1, \mathcal{S}'_2, ..., S'_t$ (they do not need to be disjoint). In addition, $\mathcal{A}$ is also allowed to examine another subset $S'_{scan} \subseteq S$ by paying $\Omega(|S'_{scan}|)$ time. With all these, the algorithm must ensure

$$S'_{scan} \cup S'_1 \cup S'_2 \cup ... \cup S'_t = S. \tag{4}$$

The above constraint is natural because otherwise at least one element of S is absent from $S'_{scan} \cup S'_1 \cup S'_2 \cup ... \cup S'_t$. If the algorithm $\mathcal{A}$ "dares" to return the query answer anyway, it must have acquired certain special properties of the underlying problem. In this work, we are interested in generic reductions that are unaware of problem-specific properties.

We will refer to reduction algorithms obeying the above requirements as *black-box reductions*. Our algorithms in Lemma 2 and Theorem 1 belong to the black-box class.

Tightness of Theorem 1. We will prove that any black-box reduction can answer only $r = O(\frac{n \log n}{Q(n)})$ queries in $O(n \operatorname{Log} r)$ time for any function $Q(n) : \mathbb{N}^+ \to \mathbb{N}^+$ that

- satisfies $Q(n) = O(n)$, and $Q(n) = \Theta(Q(c \cdot n))$ for any constant $c > 0$;
- is sub-additive, i.e., $Q(x + y) \leq Q(x) + Q(y)$ holds for any $x, y \geq 1$.

This will confirm the tightness of Theorem 1 on the black-box class.

We will contrive a decomposable problem and an accompanying data structure, both of which make no sense in reality but nonetheless are sound in mathematics. The dataset S consists of n arbitrary elements; given any predicate, a query on S always returns $|S|$ (the concrete forms of elements and predicates are irrelevant). Whenever asked to "build" a data structure on S, we waste on purpose $|S| \log |S|$ time and then simply output an arbitrary permutation of S in an array. Whenever asked to "answer" a query, we waste on purpose $Q(|S|)$ time and then return $|S|$. The problem is clearly decomposable.

We argue that any black-box reduction algorithm $\mathcal{A}$ needs $\Omega(Q(n))$ time to answer *every* query. Consider an arbitrary query and suppose that $\mathcal{A}$ processes it by searching the structures $\mathcal{T}(S'_1), ..., \mathcal{T}(S'_t)$ for some $t \geq 0$ and scanning S'_{scan}. By the design of our structure, the query cost is at least

$$\begin{aligned}
\Omega(|S'_{scan}|) + \sum_{i \in [t]} Q(|S'_i|) &\geq \Omega(|S'_{scan}|) + Q\Big(\sum_{i \in [t]} |S'_i|\Big) \quad \text{(by sub-additivity)} \\
&= \Omega\Big(Q(|S'_{scan}|) + Q\Big(\sum_{i \in [t]} |S'_i|\Big)\Big) \quad \text{(by } Q(n) = O(n), Q(n) = \Theta(Q(cn))\text{)} \\
&= \Omega\Big(Q\Big(|S'_{scan}| + \sum_{i \in [t]} |S'_i|\Big)\Big) \quad \text{(by sub-additivity)} \\
&= \Omega(Q(n)). \quad \text{(by (4))}
\end{aligned}$$

By definition of $\log(r)$-streak, algorithm $\mathcal{A}$ must process $r = \textsc{LogStreak}(n)$ queries within a total cost of $O(n \log r)$, which obviously cannot exceed $O(n \log n)$ (remember $r \leq n$). It thus follows that $\mathcal{A}$ can process only $O(\frac{n \log n}{Q(n)})$ queries.

4 New DDS Algorithms for Concrete Problems

We now deploy Theorem 1 to develop algorithms for concrete DDS problems, focusing on decomposable problems in Section 4.1 and non-decomposable problems in Section 4.2.

4.1 Applications to Decomposable Problems

As mentioned, if a decomposable problem has a data structure $\mathcal{T}$ that can be built in $O(n \log n)$ time (i.e., $B(n) = O(\log n)$) and supports a query in $Q(n) = O(\log n)$ time, Theorem 1 directly yields a DDS algorithm with $\textsc{Time}(n, r) = O(n \operatorname{Log} r)$ for all $r \leq n$. We enumerate below a partial list of such problems with importance to database systems, of which no algorithms with the same guarantee were known previously.

- *2D Orthogonal range counting.* See Section 1.2 for the problem definition. The structure $\mathcal{T}$ can be a persistent "aggregate" binary search tree (BST) [30].
- *Orthogonal range counting on rectangles.* The dataset S is a set of n 2-rectangles (i.e., axis-parallel boxes) in $\mathbb{R}^2$. Given an arbitrary 2-rectangle q, a query returns how many rectangles of S intersecting with q. The problem can be reduced to four queries of the previous problem (orthogonal range counting on *points*) [33]. $\mathcal{T}$ can once again be a persistent aggregate BST [30].

- *Point location.* The dataset S is a planar subdivision of $\mathbb{R}^2$ defined by n line segments, where each segment is associated with the ids of the two faces incident to it. Given an arbitrary point $q \in \mathbb{R}^2$, a query returns the face of the subdivision containing q, which boils down to finding the segment immediately above q (i.e., the first segment "hit" by a ray shooting upwards from q). The structure $\mathcal{T}$ can be a persistent BST [28] or Kirkpatrick's structure [20].
- $k = O(1)$ *nearest neighbor search in 2D.* The dataset S is a set of n points in $\mathbb{R}^2$. Fix a constant integer $k \geq 1$. Given a point $q \in \mathbb{R}^2$, a query returns the k points in P closest to q. The structure $\mathcal{T}$ can be a point location structure [20, 28] built on an order-k Voronoi diagram (an order-k Voronoi diagram can be computed in $O(n \log n)$ time [6]). For $k = 1$, a DDS algorithm with $\textsc{Time}(n, r) = O(n \operatorname{Log} r)$ for all $r \leq n$ has been found [1]. However, the algorithm of [1] heavily relies on the ability to merge two (order-1) Voronoi diagrams in linear time, and thus, cannot be extended to higher values of k easily.
- *Approximate nearest neighbor search in metric space.* The dataset S consists of n objects in a metric space with a constant doubling dimension (this encapsulates any Euclidean space with a constant dimensionality). Let $dist(o_1, o_2)$ represents the distance between two objects o_1 and o_2 in the space. Given an arbitrary object q in the space, a query returns an object $o \in S$ such that $dist(o, q) \leq (1 + \epsilon) \cdot dist(o', q)$ for all $o' \in S$, where ϵ is a constant. A structure $\mathcal{T}$ fulfilling our purpose can be found in [14, 22].

For orthogonal range counting in $\mathbb{R}^d$ where $d \geq 3$ is a fixed constant (as defined in Section 1.2), one can apply Theorem 1 to achieve a somewhat unusual result. It is possible [3] to build a structure $\mathcal{T}$ in $O(n \log n)$ time that answers a query in $Q(n) = O(n^\epsilon)$ time where the constant $\epsilon > 0$ can be made arbitrarily small. Theorem 1 thus produces a DDS algorithm with $\textsc{Time}(n, r) = O(n \operatorname{Log} r)$ for all $r \leq n^{1-\epsilon}$. As ϵ can be arbitrarily close to 0, the $\log(r)$-streak bound $\textsc{LogStreak}(n) = n^{1-\epsilon}$ is lower than the maximum value n only by a factor sub-polynomial in n. A remark is in order about the significance if this gap could be closed for all constant dimensionalities. If a DDS algorithm with $\textsc{LogStreak}(n) = n$ could be discovered, then the algorithm would also settle the following *offline* version in $O(n \log n)$ time: we are given a set P of n points in $\mathbb{R}^d$ and a set Q of n d-rectangles; the goal is to report, for each d-rectangle $q \in Q$, how many points in P are covered in q. This offline problem has been extensively studied, and yet the fastest algorithm still runs in $n \log^{\Theta(d)} n$ time to our knowledge.

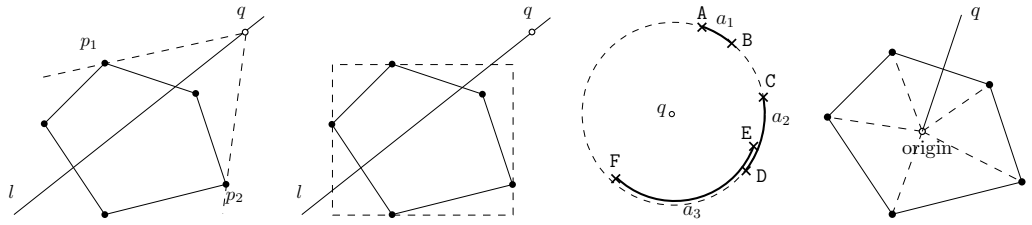
4.2 Non-Decomposable but Spectrum-Indexable Problems

This subsection will utilize Theorem 1 to deal with problems that are not decomposable problems (at least not obviously). The key is to show that the problem is $(\operatorname{Log} n, Q(n))$ spectrum indexable for a suitable $Q(n)$. This is an interesting topic on its own, as we demonstrate next by developing new DDS algorithms with $\textsc{Time}(n, r) = O(n \operatorname{Log} r)$ for all $r \leq n$ on halfplane containment, convex hull containment, range median, and 2D linear programming. The original algorithms [11, 19] for these problems were all designed using the top-down approach reviewed in Section 2. Our algorithms present a contrast that illustrates how Theorem 1 can facilitate the design of DDS algorithms.

Halfplane Containment. The problem can be converted [19] to the following equivalent form by resorting to geometry duality [7]:

- *Line through convex hull.* S is a set of n points in $\mathbb{R}^2$. Given any line l in $\mathbb{R}^2$, a query determines if l intersects with the convex hull of S, denoted as $\text{CH}(S)$.

We will concentrate on the above problem instead.



(a) Tangent rays. (b) MBR. (c) Arcs. (d) Dissection.

Figure 1 Illustrations of key concepts in Section 4.2.

Suppose, for now, that we are given a point q on l that falls outside $\text{CH}(S)$. From q, we can shoot two *tangent rays*, each touching $\text{CH}(S)$ but not entering into the interior of $\text{CH}(S)$. In Figure 1(a) where S is the set of black points, the first ray passes point $p_1 \in S$ while the second passes $p_2 \in S$. The two rays form a "wedge" enclosing $\text{CH}(S)$ within (note that the angle of the wedge is less than 180°); we will call it the *wedge of q on $CH(S)$*. Line l passes through $\text{CH}(S)$ if and only if l goes through the wedge. If the vertices of $\text{CH}(S)$ have been stored in clockwise order, the two tangent rays can be found in $O(\log n)$ time [25].

We will prove that the "line through convex hull" problem is $(\text{Log}\, n, \text{Log}\, n)$ spectrum indexable. Take any integer $s \in [1, n]$ and set $m = \lceil n/s \rceil$. Divide S arbitrarily into S_1, S_2, ..., S_m such that $|S_i| = s$ for $i \in [m-1]$ and $|S_m| = n - s(m-1)$. To build a structure $\mathcal{T}$, use $O(s \,\text{Log}\, s)$ time to compute $\text{CH}(S_i)$ for each $i \in [m]$ and store its vertices in clockwise order. The structure's construction time is $O(n \,\text{Log}\, s)$. Let us see how to answer a query with line l. Suppose, once again, that a point q on l outside $\text{CH}(S)$ is given. For each $i \in [m]$, compute the wedge of q on $\text{CH}(S_i)$ in $O(\text{Log}\, s)$ time. From these m wedges, it is a simple task to obtain in $O(m)$ time the wedge of q on $\text{CH}(S)$ (we will deal with a more general problem shortly in discussing "convex hull containment"). Now, whether l intersects with $\text{CH}(S)$ can be determined easily. The query time so far is $O(m \,\text{Log}\, s)$.

It remains to explain how to find q. This can be done in $O(1)$ time if we already have the minimum axis-parallel bounding rectangle of S, denoted as $\text{MBR}(S)$. Note that $\text{MBR}(S)$ must contain $\text{CH}(S)$; see Figure 1(b). It is clear that $\text{MBR}(S)$ can be obtained from $\text{MBR}(S_1)$, $\text{MBR}(S_2)$, ..., $\text{MBR}(S_m)$ in $O(m)$ time, while each $\text{MBR}(S_i)$ ($i \in [m]$) can be computed in $O(s)$ time during the construction of the structure $\mathcal{T}$. We thus conclude that the problem is $(\text{Log}\, n, \text{Log}\, n)$ spectrum indexable and can now apply Theorem 1.

Convex Hull Containment. In this problem, S is a set of n points in $\mathbb{R}^2$. Given any point $q \in \mathbb{R}^2$, a query determines if q is covered by $\text{CH}(S)$. We will prove that the problem is $(\text{Log}\, n, \text{Log}\, n)$ spectrum indexable.

Take any integer $s \in [1, n]$ and set $m = \lceil n/s \rceil$. Divide S arbitrarily into S_1, S_2, ..., S_m such that $|S_i| = s$ for $i \in [m-1]$ and $|S_m| = n - s(m-1)$. To build a structure $\mathcal{T}$, compute $\text{CH}(S_i)$ for each $i \in [m]$ and store its vertices in clockwise order; this takes $O(n \,\text{Log}\, s)$ time as explained before. Let us see how to answer a query given point q. For each $i \in [m]$, whether q is covered by $\text{CH}(S_i)$ can be checked in $O(\text{Log}\, s)$ time [25]. If the answer yes for any i, point q must be covered by $\text{CH}(S)$, and we are done. The subsequent discussion assumes that q is outside $\text{CH}(S_i)$ for all i. In $O(m \,\text{Log}\, s)$ time, compute the wedge of q on $\text{CH}(S_i)$ for all $i \in [m]$ as described in the previous problem.

It remains to determine from the m wedges whether q is covered by $\text{CH}(S)$. This can be re-modeled as the following problem. Place an arbitrary circle centered at q. For each wedge, its two bounding rays intersect the circle into an arc of less than 180°. Let $a_1, a_2, ..., a_m$ be

the arcs produced this way, and define a^* to be the smallest arc on the circle covering them all. Figure 1(c) shows an example with $m = 3$. Arc a_1 is subtended by points `A` and `B`, arc a_2 by points `C` and `D`, and arc a_3 by points `E` and `F`. Here, the smallest arc a^* goes clockwise from point `A` to `F`. Crucially, q is covered by $\text{CH}(S)$ if and only if a^* spans at least 180° (this is the case in Figure 1(c)) – note that if q is outside $\text{CH}(S)$, then a^* must be subtended by the wedge of q on $\text{CH}(S)$, which must be less than 180°. Therefore, the goal is to determine whether a^* is at least 180°.

It is possible to achieve the purpose in $O(m)$ time (even if the m arcs are given to us in an arbitrary order). For this purpose, we process the arcs one by one, maintain the smallest arc a^* covering the arcs already processed, and stop the algorithm when it becomes clear that a^* must be at least 180°. Specifically, for $i = 1$, simply set a^* to a_1. Iteratively, given the next arc a_i ($i \geq 2$), check in constant time if an arc of less than 180° can cover both a_i and a^*. If so, update a^* to that arc; otherwise, stop the algorithm. In Figure 1(c), for example, after a_2 is processed, the a^* we maintain goes clockwise from `A` to `D`. When processing a_3, the algorithm realizes that a^* must be at least 180° and hence terminates.

We thus conclude that the convex hull containment problem is $(\text{Log}\, n, \text{Log}\, n)$ spectrum indexable and can now apply Theorem 1.

Range Median. In this problem, S is a set of n real values stored in an array A. Given an arbitrary integer pair (x, y) satisfying $1 \leq x \leq y \leq n$, a query returns the median of $A[x : y]$. We will show that the problem is $(\text{Log}\, n, \text{Log}\, n)$ spectrum indexable. Fix any integer $s \leq n$ and $m = \lceil n/s \rceil$. Define $S_i = A[(i-1)s + 1 : i \cdot s]$ for each $i \leq m - 1$ and $S_m = A[(m-1)s + 1 : n]$. Next, we will assume $s \leq \sqrt{n}$; otherwise, simply use $O(n \log n) = O(n \log s)$ time to create a structure of [11] on the whole S, which is able to answer any query on S in $O(\log n) = O(\log s)$ time.

To build a structure $\mathcal{T}$, for each $i \in [m]$, store S_i in ascending order, but each element of S_i should be associated with its original position index in A. It is clear that $\mathcal{T}$ can be built in $O(n\, \text{Log}\, s)$ time. Let us see how to answer a query with predicate (x, y). First, determine the values $a, b \in [m]$ such that $A[x] \in S_a$ and $A[y] \in S_b$, which can be trivially done in $O(m)$ time. Scan S_a and identify the subset $S'_a = S_a \cap A[x : y]$ (for each element in S_a, check if its original index in A falls in $[x, y]$). Because S_a is sorted, we can produce S'_a in the sorted order in $O(s)$ time. In the same fashion, compute $S'_b = S_b \cap A[x : y]$ in $O(s)$ time. At this moment, all the elements of $A[x : y]$ have been partitioned in $b - a + 1$ sorted arrays: $S'_a, S_{a+1}, S_{a+2}, ..., S_{b-1}, S'_b$. The goal now is to find the $\lfloor (y - x + 1)/2 \rfloor$-th smallest element in the union of these arrays. Frederickson and Johnson [10] described an algorithm to select the element of a given rank from the union of sorted arrays. Their algorithm runs in $O(m\, \text{Log}\, \frac{n}{m}) = O(m\, \text{Log}\, s)$ time in our scenario. Overall the query time is $O(s + m\, \text{Log}\, s) = O(m\, \text{Log}\, s)$ because $s \leq \sqrt{n}$.

We now conclude that the range median problem is $(\text{Log}\, n, \text{Log}\, n)$ spectrum indexable and are ready to apply Theorem 1.

2D Linear Programming. By resorting to geometry duality [7], the problem can be converted [19] to "line through convex hull" (which we already solved) and the following:

- *Ray exiting convex hull.* Here, S is a set of n points in $\mathbb{R}^2$ such that $\text{CH}(S)$ covers the origin. Given any ray emanating q from the origin, a query returns the edge e_{exit} of $\text{CH}(S)$ from which q exits $\text{CH}(S)$.

We will concentrate on the above problem instead. Before proceeding, let us state two facts about the problem:

- Given any ray q, it is possible to find e_{exit} in $O(n)$ time, using an algorithm in [21]; we will call this the *basic algorithm*.
- We can create a structure in $O(n \log n)$ time to answer any query in $O(\log n)$ time. First compute $\text{CH}(S)$ and then "dissect" it using the segments connecting the origin to all the vertices; see Figure 1(d). A query with ray q can then be answered by looking for the triangle in the dissection that q goes through. We will call this the *basic structure*.

Unlike all the problems discussed so far, currently we cannot prove that "ray exiting convex hull" is $(\text{Log}\, n, \text{Log}\, n)$ spectrum indexable. However, by virtue of Theorem 1, we do *not* have to! It suffices to show that the problem is $(\text{Log}\, n, n^c)$ spectrum indexable for any positive constant $c < 1$. Theorem 1 then allows us to answer $r \leq n^{1-c} \log n$ queries in $O(n \,\text{Log}\, r)$ time. After r has reached n^{1-c}, we can afford to build the basic structure in $O(n \log n) = O(n \log r)$ time to answer every subsequent query in $O(\log n) = O(\log r)$ time. This allow us achieve $\textsc{Time}(n, r) = O(n \,\text{Log}\, r)$ for all $r \leq n$.

We will prove that the "ray exiting convex hull" problem is $(\text{Log}\, n, \sqrt{n})$ spectrum indexable. We will achieve the purpose by resorting to a result of [19], where Karp, Motwani, and Raghavan used the top-down approach to build a binary tree $\mathcal{T}$ with the following properties.

- If a node u is at level ℓ of $\mathcal{T}$, then u is associated with a set $S(u)$ of $n/2^\ell$ points in S.
- The first ℓ levels of the tree can be built in $O(n \cdot \ell)$ time.
- A query is answered by traversing at most a root-to-leaf path of $\mathcal{T}$. If the search process descendants to a node u, then the target edge e_{exit} can be found in $O(|S(u)|)$ time by running the basic algorithm [21] on $S(u)$.

Back to our scenario, fix any integer $s \leq n$. Build the first $1 + \lceil \text{Log}\, \sqrt{s} \rceil$ levels of $\mathcal{T}$ on S in $O(n \,\text{Log}\, \sqrt{s}) = O(n \,\text{Log}\, s)$ time. To answer a query, we descend a path of $\mathcal{T}$ to a node u of level $\lceil \text{Log}\, \sqrt{s} \rceil$ if the edge e_{exit} has not already been found. The set $|S(u)|$ has at most $n/2^{\text{Log}\, \sqrt{s}} = n/\sqrt{s}$ points. We can thus run the basic algorithm on $S(u)$ to find e_{exit} in $O(n/\sqrt{s}) = O(\frac{n}{s} \cdot \sqrt{s})$ time. Therefore, "ray exiting convex hull" problem is $(\text{Log}\, n, \sqrt{n})$ spectrum indexable.

We close this section with a remark, as is revealed by the above discussion, about an inherent connection between the top-down approach and our reduction. In essence, we build the structure of [19] incrementally: the i-th ($i \geq 1$) "epoch" (in the proof of Section 3.2) re-builds the first 2^{2^i} levels of $\mathcal{T}$. This is different from [19] where the nodes are "expanded upon first touch" in the way described in Section 2. In fact, all existing DDS algorithms designed based on the top-down approach can be encapsulated into our reduction framework through the bridge of "spectrum indexability", in the manner we demonstrated for "ray exiting convex hull".

5 DDS Using Structures with $\omega(n \log n)$ Construction Time

Our discussion so far has focused on data structures with $B(n) = O(\log n)$. In this section, we will first show the necessity of this condition for black-box reductions to guarantee even a non-constant $\log(r)$-streak bound. As a second step, we present an extension of Theorem 1 that permits the deployment of a structure with $\max\{B(n), Q(n)\} = \text{polylog}\, n$ to produce a DDS algorithm with good $\textsc{Time}(n, r)$ for all $r \leq n$.

Constant Streak Bounds for $B(n) = \omega(\log n)$. Our subsequent hardness argument requires $n \cdot B(n)$ to be a convex function. Consider any black-box reduction algorithm $\mathcal{A}$. Recall that $\mathcal{A}$ is required to work on any decomposable problem with a restricted data

structure (the reader may wish to review Section 3.3 before proceeding). Suppose that $\mathcal{A}$ can guarantee a $\log(r)$-streak bound of $\textsc{LogStreak}(n)$ for all such problems, namely, $\mathcal{A}$ can always answer r queries in $O(n \operatorname{Log} r)$ time for all $r \leq \textsc{LogStreak}(n)$. We will show that, if $B(n) = \omega(\log n)$, then $\textsc{LogStreak}(n)$ must be $O(1)$.

In a way similar to Section 3.3, we will contrive a decomposable problem and an accompanying data structure. The dataset S contains n arbitrary elements; given any predicate, a query on S always returns $|S|$. Whenever asked to build a data structure $\mathcal{T}(S)$ on S, we waste on purpose $|S| \cdot B(|S|)$ time and then output an arbitrary permutation of S in an array. Whenever asked to answer a query, we immediately return $|S|$ in constant time. In other words, the function $Q(n)$ is fixed to 1.

Henceforth, we will fix r to the value of $\textsc{LogStreak}(n)$ that $\mathcal{A}$ can ensure when it is given our contrived data structure. We will assume $r = \omega(1)$; otherwise, $\textsc{LogStreak}(n) = O(1)$ and our job is done. As it answers r queries in $O(n \operatorname{Log} r)$ time, at least one of the queries must have a cost of $O(\frac{n \operatorname{Log} r}{r})$. We will concentrate on this particular query in the rest of the argument.

Recall from Section 3.3 that, to answer this query, algorithm $\mathcal{A}$ needs to search a number (including 0) of structures $\mathcal{T}(S'_1), \mathcal{T}(S'_2), ..., \mathcal{T}(S'_t)$ and scan a subset $S'_{scan} \subseteq S$. As $\mathcal{A}$ needs to pay a cost of $\Omega(|S'_{scan}|)$ to scan S'_{scan}, it must hold that $|S'_{scan}| \leq \alpha \cdot \frac{n \operatorname{Log} r}{r}$ for some constant $\alpha > 0$. We will consider r, which we know is $\omega(1)$, to be large enough to make $\alpha \cdot \frac{n \operatorname{Log} r}{r} \leq n/2$. Because $\mathcal{A}$ must obey (4), we can assert that

$$|S'_1 \cup S'_2 \cup ... \cup S'_t| \geq |S| - |S'_{scan}| \geq n/2. \tag{5}$$

This implies $t \geq 1$. Since algorithm $\mathcal{A}$ must pay a constant time to search each structure $\mathcal{T}(S'_i)$ $(i \in [t])$, we must have

$$t \quad = \quad O((n/r) \cdot \operatorname{Log} r). \tag{6}$$

However, by how we design $\mathcal{T}$, the total cost of constructing $\mathcal{T}(S'_1), \mathcal{T}(S'_2), ..., \mathcal{T}(S'_t)$ is

$$\sum_{i \in [t]} |S'_i| \cdot B(|S'_i|). \tag{7}$$

Set $\lambda = \sum_{i \in [t]} |S'_i|$; from (5), we know $\lambda \geq n/2$. As $n \cdot B(n)$ is a convex function and $B(n)$ is non-decreasing, we know that (7) is minimized when $|S'_i| = \lambda/t$ for all $i \in [t]$. Therefore:

$$(7) \quad \geq \quad \lambda \cdot B(\lambda/t) \geq \frac{n}{2} \cdot B\Big(\frac{n}{2t}\Big). \tag{8}$$

From (6), we have $n/(2t) = \Omega(r/\log r)$. Because $B(n) = \omega(\log n)$, rudimentary asymptotic analysis shows that $B(n/(2t))$ must be $\omega(\log r)$.

We now conclude that (8), and hence (7), must be $\omega(n \operatorname{Log} r)$. But this contradicts that algorithm $\mathcal{A}$ can process r queries in $O(n \operatorname{Log} r)$ time. Therefore, our assumption that $r = \omega(1)$ must be wrong.

DDS Algorithms with $B(n) = \omega(\log n)$. Data structures having $\omega(n \log n)$ construction time are still useful for DDS as long as we are not obsessed with answering r queries in $O(n \operatorname{Log} r)$ time. To make this formal, we modify the techniques behind Theorem 1 to obtain another generic reduction with the following guarantees.

▶ **Theorem 3.** *Suppose that $B(n)$ and $Q(n)$ are non-decreasing functions that are both $O(\log^\gamma n)$ where $\gamma \geq 1$ is a constant. Every $(B(n), Q(n))$ spectrum indexable problem admits a DDS algorithm with $\textsc{Time}(n, r) = O(n \operatorname{Log}^\gamma r)$ for all $r \leq n$.*

The proof is similar to what was presented in Section 3.2 and is moved to Appendix A. An interesting application of the above is orthogonal range counting/reporting in $\mathbb{R}^d$ where d is a fixed constant at least 3 (see the problem definition in Section 1.2). The range tree augmented with fractional cascading [9], constructable in $O(n \log^{d-1} n)$ time on n points, answers a counting/reporting query also in $O(n \log^{d-1} n)$ time. The problem is decomposable and thus $(\text{Log}^{d-1} n, \text{Log}^{d-1} n)$ spectrum indexable. Theorem 3 directly gives a DDS algorithm with $\textsc{Time}(n, r) = O(n \, \text{Log}^{d-1} r)$ for all $r \leq n$, which strictly improves the results of [19] as mentioned in Section 1.2.

References

1 Alok Aggarwal and Prabhakar Raghavan. Deferred data structure for the nearest neighbor problem. *Information Processing Letters (IPL)*, 40(3):119–122, 1991. doi:10.1016/0020-0190(91)90164-D.

2 Jérémy Barbay, Ankur Gupta, Srinivasa Rao Satti, and Jonathan Sorenson. Near-optimal online multiselection in internal and external memory. *J. Discrete Algorithms*, 36:3–17, 2016. doi:10.1016/J.JDA.2015.11.001.

3 Jon Louis Bentley and Hermann A. Maurer. Efficient worst-case data structures for range searching. *Acta Inf.*, 13:155–168, 1980. doi:10.1007/BF00263991.

4 Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *Journal of Computer and System Sciences (JCSS)*, 7(4):448–461, 1973. doi:10.1016/S0022-0000(73)80033-9.

5 Gerth Stolting Brodal, Beat Gfeller, Allan Gronlund Jorgensen, and Peter Sanders. Towards optimal range medians. *Theoretical Computer Science*, 412(24):2588–2601, 2011. doi:10.1016/J.TCS.2010.05.003.

6 Timothy M. Chan and Konstantinos Tsakalidis. Optimal deterministic algorithms for 2-d and 3-d shallow cuttings. *Discrete & Computational Geometry*, 56(4):866–881, 2016. doi:10.1007/S00454-016-9784-4.

7 Bernard Chazelle, Leonidas J. Guibas, and D. T. Lee. The power of geometric duality. *BIT Numerical Mathematics*, 25(1):76–90, 1985. doi:10.1007/BF01934990.

8 Yu-Tai Ching, Kurt Mehlhorn, and Michiel H. M. Smid. Dynamic deferred data structuring. *Information Processing Letters (IPL)*, 35(1):37–40, 1990. doi:10.1016/0020-0190(90)90171-S.

9 Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008.

10 Greg N. Frederickson and Donald B. Johnson. The complexity of selection and ranking in x+y and matrices with sorted columns. *Journal of Computer and System Sciences (JCSS)*, 24(2):197–208, 1982. doi:10.1016/0022-0000(82)90048-4.

11 Beat Gfeller and Peter Sanders. Towards optimal range medians. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 475–486, 2009. doi:10.1007/978-3-642-02927-1_40.

12 Goetz Graefe, Felix Halim, Stratos Idreos, Harumi A. Kuno, and Stefan Manegold. Concurrency control for adaptive indexing. *Proceedings of the VLDB Endowment (PVLDB)*, 5(7):656–667, 2012. doi:10.14778/2180912.2180918.

13 Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment (PVLDB)*, 5(6):502–513, 2012. doi:10.14778/2168651.2168652.

14 Sariel Har-Peled and Nirman Kumar. Approximate nearest neighbor search for low-dimensional queries. *SIAM Journal on Computing*, 42(1):138–159, 2013. doi:10.1137/110852711.

15 Sariel Har-Peled and S. Muthukrishnan. Range medians. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 503–514, 2008. doi:10.1007/978-3-540-87744-8_42.

16 Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *Proceedings of Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 68–78, 2007. URL: http://cidrdb.org/cidr2007/papers/cidr07p07.pdf.

17 Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column-stores. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 297–308, 2009. doi:10.1145/1559845.1559878.

18 Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment (PVLDB)*, 4(9):585–597, 2011. doi:10.14778/2002938.2002944.

19 Richard M. Karp, Rajeev Motwani, and Prabhakar Raghavan. Deferred data structuring. *SIAM Journal on Computing*, 17(5):883–902, 1988. doi:10.1137/0217055.

20 David G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal of Computing*, 12(1):28–35, 1983. doi:10.1137/0212002.

21 David G. Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(1):287–299, 1986. doi:10.1137/0215021.

22 Robert Krauthgamer and James R. Lee. Navigating nets: simple algorithms for proximity search. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 798–807, 2004. URL: http://dl.acm.org/citation.cfm?id=982792.982913.

23 Konstantinos Lampropoulos, Fatemeh Zardbani, Nikos Mamoulis, and Panagiotis Karras. Adaptive indexing in high-dimensional metric spaces. *Proceedings of the VLDB Endowment (PVLDB)*, 16(10):2525–2537, 2023. doi:10.14778/3603581.3603592.

24 Rajeev Motwani and Prabhakar Raghavan. Deferred data structuring: Query-driven preprocessing for geometric search problems. In *Proceedings of Symposium on Computational Geometry (SoCG)*, pages 303–312, 1986. doi:10.1145/10515.10548.

25 Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences (JCSS)*, 23(2):166–204, 1981. doi:10.1016/0022-0000(81)90012-X.

26 Bryce Sandlund and Sebastian Wild. Lazy search trees. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 704–715, 2020. doi:10.1109/FOCS46700.2020.00071.

27 Bryce Sandlund and Lingyi Zhang. Selectable heaps and optimal lazy search trees. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1962–1975, 2022. doi:10.1137/1.9781611977073.78.

28 Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *Communications of the ACM (CACM)*, 29(7):669–679, 1986. doi:10.1145/6138.6151.

29 Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. An experimental evaluation and analysis of database cracking. *The VLDB Journal*, 25(1):27–52, 2016. doi:10.1007/S00778-015-0397-Y.

30 Yufei Tao and Dimitris Papadias. Range aggregate processing in spatial databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(12):1555–1570, 2004. doi:10.1109/TKDE.2004.93.

31 Fatemeh Zardbani, Peyman Afshani, and Panagiotis Karras. Revisiting the theory and practice of database cracking. In *Proceedings of Extending Database Technology (EDBT)*, pages 415–418, 2020. doi:10.5441/002/EDBT.2020.46.

32 Fatemeh Zardbani, Nikos Mamoulis, Stratos Idreos, and Panagiotis Karras. Adaptive indexing of objects with spatial extent. *Proceedings of the VLDB Endowment (PVLDB)*, 16(9):2248–2260, 2023. doi:10.14778/3598581.3598596.

33 Donghui Zhang, Vassilis J. Tsotras, and Dimitrios Gunopulos. Efficient aggregation over objects with extent. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 121–132, 2002. doi:10.1145/543613.543629.

A Proof of Theorem 3

Our algorithm executes in *epochs*. At the beginning of the i-th ($i \geq 1$) epoch, we set $s = 2^{2^i}$ and create a structure $\mathcal{T}$ – the structure promised by $(B(n), Q(n))$ spectrum indexability – on S in $O(n \cdot B(s)) = O(n \cdot 2^{i \cdot \gamma})$ time. The structure allows us to answer any query in $O(\frac{n}{s} \cdot Q(s))$ time. The i-th epoch finishes after s queries are answered *during* the epoch. These queries have a total cost of

$$s \cdot O\Big(\frac{n}{s} \cdot Q(s)\Big) = O(n \cdot Q(s)) = O(n \cdot 2^{i \cdot \gamma}).$$

Hence, the total computation time of the i-th epoch is $O(n \cdot 2^{i \cdot \gamma})$. Because $\gamma \geq 1$, we know that $n \cdot 2^{i \cdot \gamma}$ at least doubles when i increases by 1. Hence, the total cost of all epochs is asymptotically dominated by $O(n \cdot 2^{h \cdot \gamma})$, where h is the number of epochs needed.

Precisely, the value of h is the smallest integer $x \geq 1$ satisfying two conditions:

- **C1:** $2^{2^x} \leq n$ (the value of s must not exceed n);
- **C2:** $\sum_{i=1}^{x} 2^{2^i} \geq r$ (the number of queries answerable by h epochs must be at least r).

Let H represent the smallest integer $x \geq 1$ such that $2^{2^x} \geq r$. This means

$$2^{2^{H-1}} < r \quad \Leftrightarrow \quad 2^{2^H} < r^2. \tag{9}$$

When $2^{2^H} \leq n$, the definitions of h and H imply $h \leq H$. In this case, all the epochs incur a total cost of $O(n \cdot 2^{h \cdot \gamma}) = O(n \cdot 2^{H \cdot \gamma}) = O(n \operatorname{Log}^{\gamma} r)$.

The above argument does not work if $2^{2^H} > n$. However, when this happens, we know from (9) that $r^2 > 2^{2^H} > n$, leading to $r > \sqrt{n}$. As soon as r reaches $\lceil \sqrt{n} \rceil$ – the *snapping point* – we create a structure $\mathcal{T}$ on the whole S in $O(n \log^{\gamma} n)$ time, and use it to answer every subsequent query in $O(Q(n)) = O(\log^{\gamma} n)$ time until $r = n$. The queries after the snapping point require a total cost of $O(n \log^{\gamma} n) = O(n \log^{\gamma} r)$. We thus have obtained an algorithm that guarantees $\textsc{Time}(n, r) = O(n \operatorname{Log}^{\gamma} r)$ for all $r \leq n$, completing the proof of Theorem 3.