Independent Range Sampling

Xiaocheng Hu Miao Qiao Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong
New Territories, Hong Kong
{xchu, mqiao, taoyf}@cse.cuhk.edu.hk

ABSTRACT

This paper studies the *independent range sampling* problem. The input is a set P of n points in $\mathbb{R}$. Given an interval $q = [x, y]$ and an integer $t \geq 1$, a query returns t elements uniformly sampled (with/without replacement) from $P \cap q$. The sampling result must be *independent* from those returned by the previous queries. The objective is to store P in a structure for answering all queries efficiently.

If P fits in memory, the problem is interesting when P is dynamic (i.e., allowing insertions and deletions). The state of the art is a structure of $O(n)$ space that answers a query in $O(t \log n)$ time, and supports an update in $O(\log n)$ time. We describe a new structure of $O(n)$ space that answers a query in $O(\log n + t)$ expected time, and supports an update in $O(\log n)$ time.

If P does not fit in memory, the problem is challenging even when P is static. The best known structure incurs $O(\log_B n + t)$ I/Os per query, where B is the block size. We develop a new structure of $O(n/B)$ space that answers a query in $O(\log^\star(n/B) + \log_B n + (t/B)\log_{M/B}(n/B))$ amortized expected I/Os, where M is the memory size, and $\log^\star(n/B)$ is the number of iterative $\log_2(.)$ operations we need to perform on n/B before going below a constant. We also give a lower bound argument showing that this is nearly optimal—in particular, the multiplicative term $\log_{M/B}(n/B)$ is necessary.

Categories and Subject Descriptors

F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; H.3.1 [**Information storage and retrieval**]: Content analysis and indexing—*indexing methods*

Keywords

Independent range sampling, range reporting, lower bound

PODS'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2594538.2594545.

1. INTRODUCTION

A *reporting* query, in general, retrieves from a dataset all the elements satisfying a condition. In the current big data era, such a query easily turns into a "big query", namely, one whose result contains a huge number of elements. In this case, even the simple task of enumerating all these elements can prove to be problematic. For example, assuming that a hard disk can write 4k bytes in one millisecond, it takes an hour to write a query result of 4 billion integers. Note that a query result of 4 billion elements is actually rather small on a tera-byte scale dataset, while a tera bytes of data could hardly be counted as "big" by today's standard.

This phenomenon naturally brings back the notion of *query sampling*, a classic concept that was introduced to the database community several decades ago. The goal of query sampling is to return, instead of an entire query result, only a random sample set of the elements therein. The usefulness of such a sample set has long been recognized even in the non-big-data days (see an excellent survey in [12]). The unprecedented gigantic data volume we are facing nowadays has only strengthened the importance of query sampling. Particularly, this is an effective technique in dealing with the big-query issue mentioned earlier in many scenarios where acquiring a query result in its entirety is not compulsory.

This work aims to endow query sampling with *independence*; namely, the samples returned by each query should be independent from the samples returned by the previous queries. In particular, we investigate how to achieve this purpose on *range reporting*, as it is a very fundamental query in the database and data structure fields. Formally, the problem we study can be stated as follows:

PROBLEM 1 (**Independent Range Sampling (IRS)**). *Let P be a set of n points in $\mathbb{R}$. Given an interval $q = [x, y]$ in $\mathbb{R}$ and an integer $t \geq 1$, we define two types of queries:*

- *A* **with replacement (WR) query** *returns a sequence of t points, each of which is taken uniformly at random from $P(q) = P \cap q$.*
- *Requiring $t \leq |P(q)|$, a* **without replacement (WoR) query** *returns a subset R of $P(q)$ with $|R| = t$, which is taken uniformly at random from all the size-t subsets of $P(q)$. The query may output the elements of R in an arbitrary order.*

In both cases, the output of the query must be independent from the outputs of all previous queries.

Guaranteeing independence among the sampling results of all queries ensures a strong sense of fairness: the elements

satisfying a query predicate always have the same chance of being reported (regardless of the samples returned previously), as is a desirable feature in battling the "big-query issue". Furthermore, the independence requirement also offers convenience in statistical analysis and algorithm design. In particular, it allows one to issue the *same* query multiple times to fetch different samples. This is especially useful when one attempts to test a property by sampling, but is willing to accept only a small failure probability of drawing a wrong conclusion. The independence guarantees that the failure probability decreases exponentially with the number of times the query is repeated.

Computation Models. We study IRS in both the scenarios where the input set P fits or does not fit in memory, respectively. In the former scenario, we discuss algorithms on a *random access machine* (RAM), where it takes constant time to perform a comparison, a + operation, and to access a memory location. For randomized algorithms, we make the standard assumption that it takes constant time to generate a random integer in $[0, 2^w - 1]$, where w is the length of a word.

In the latter scenario (where P does not fit in memory), we adhere to the standard *external memory* (EM) model [2], where a machine has M words of memory and a disk that has been formatted into blocks of size B words. It always holds that $M \geq 2B$. An I/O either reads a block of data into memory, or writes B words in memory to a disk block. The cost of an algorithm is the number of I/Os performed (CPU time is free), while the space of a structure is the number of disk blocks occupied.

Finally, we define $\log^{(0)} x = x$, and $\log^{(i+1)} x = \log_2(\log^{(i)} x)$ for any integer $i \geq 0$. Define $\log^\star x$ to be the smallest i such that $\log^{(i)} x \leq 2$.

Existing Results. Next we review the literature on IRS, assuming first the WR semantics. In internal memory, the problem is trivial when P is static. Specifically, we can simply store the points of P in ascending order using an array A. Given a query with parameters $q = [x, y]$ and t, we can first perform binary search to identify the subsequence in A that consists of the elements covered by q. Then, we can simply sample from the subsequence by generating t random ranks and accessing t elements. The total query cost is $O(\log n + t)$.

The problem becomes much more interesting when P is dynamic, namely, it admits insertions and deletions of elements. This problem was first studied more than two decades ago. The best solution to this date uses $O(n)$ space, answers a query in $O(t \log n)$ time, and supports an update in $O(\log n)$ time (see [12] and the references therein). This can be achieved by creating a "rank structure" on P that allows us to fetch the i-th (for any $i \in [1, n]$) largest element of P in $O(\log n)$ time. After this, we can then simulate the static algorithm described earlier by spending $O(\log n)$ time, instead of $O(1)$, fetching each sample.

In external memory, the IRS problem is challenging even when P is static. Note that accessing t random positions from a disk-resident array is expensive: it takes $O(t)$ I/Os when the array size is large. As a result, the RAM algorithm we discussed earlier incurs $O(\log_B n + t)$ I/Os in external memory, assuming a B-tree on the array A. This is nearly a factor of B higher than the $\Theta(t/B)$ cost needed to write t samples. Going back to the baseline solution, one can always retrieve the entire $P(q) = P \cap q$, and then sample t elements from $P(q)$ using a standard algorithm [7, 15]. Assuming $t \leq k$, the query cost is $O(\log_B n + (k/B) \log_{M/B}(k/B))$, where $k = |P(q)|$. This cost is not necessarily better than $O(\log_B n + t)$ because t can be arbitrarily smaller than k.

We are not aware of work that tackles specifically WoR queries. However, we will see later that a WoR query with parameters q, t can be answered by a constant number (in expectation) of WR queries having parameters $q, 2t$. Hence, the aforementioned performance guarantees also hold on WoR queries in expectation.

The above represent the current state of the art on the IRS problem. It is worth mentioning that if one does not require independence of the sampling results of different queries, query sampling can be supported as follows. For each $i = 0, 1, ..., \lceil \log n \rceil$, maintain a set P_i by independently including each element of P with probability $1/2^i$. Given a query with interval $q = [x, y]$, $P_i \cap q$ serves as a sample set where each element in $P(q)$ is taken with probability $1/2^i$. However, by issuing the same query again, one always gets back the same samples, thus losing the benefits of IRS mentioned before.

Also somewhat relevant is the recent work of Wei and Yi [16], in which they studied how to return various statistical summaries (e.g., quantiles) on the result of range reporting. They did not address the problem of query sampling, let alone how to enforce the independence requirement. At a high level, IRS may be loosely classified as a form of *online aggregation* [8], because most research on this topic has been devoted to the maintenance of a random sample set of a long-running query (typically, aggregation from a series of joins); see [10] and the references therein. As far as IRS is concerned, we are not aware of any work along this line that guarantees better performance than the solutions surveyed previously.

It is worth mentioning that sampling algorithms have been studied extensively in various contexts (for entry points into the literature, see [1, 4, 5, 6, 7, 11, 14, 15]). These algorithms aim at efficiently producing sample sets for different purposes over a static or evolving dataset. Our focus, on the other hand, is to design data structures for sampling the results of arbitrary range queries.

Our Results. We present several new results on the IRS problem. In Section 2, we give a dynamic RAM structure of $O(n)$ space that answers a WR or WoR query in $O(\log n + t)$ expected time, and supports an update in $O(\log n)$ time. All the expectations in this paper depend only on the random choices made by our algorithms.

In EM, we first establish in Section 3 a lower bound showing that one cannot hope to achieve a query cost such as $O(\log_B n + t/B)$ using a structure of reasonable space even when P is static. Specifically, we prove that, whenever $\log_{M/B}(n/B) \leq B$, any structure occupying $n^{O(1)}$ blocks must perform $\Omega((t/B) \log_{M/B}(n/B))$ expected I/Os answering a WR query with parameters $q = (-\infty, \infty)$ and $t \in [B, n]$. In fact, this is true even if query cost is *amortized*. That is, there is a sequence of queries such that every structure of $n^{O(1)}$ space must incur $\Omega(m \cdot (t/B) \log_{M/B}(n/B))$ I/Os in expectation processing the entire sequence, where m is the number of queries in the sequence. A similar lower

bound also holds on WoR queries, except for $t \in [B, n^{1-\epsilon}]$, where $\epsilon > 0$ is an arbitrarily small constant.

As a second step, in Section 4 we develop a structure that uses $O(n/B)$ space, and answers a WR or WoR query in $O(\log^\star(n/B)+\log_B n+(t/B)\log_{M/B}(n/B))$ amortized I/Os in expectation. Our lower bound shows that the query cost is optimal within only an additive factor of $O(\log^\star(n/B) + \log_B n)$.

Interestingly, our RAM and EM structures are based on different technical ideas. In RAM, our structure essentially stores the elements of the input set P in a number of arrays of various sizes. This idea, however, does not work in EM due to the expensive I/O overhead of sampling directly from an array. Instead, we develop an approach that precomputes independent samples even before a query comes. Those samples are stored inside our data structure, which is modified every time a query is answered so that we can ensure there are always enough samples for the next query. We show how to balance the work of sample computation among the queries so that the amortized query cost remains low.

The concept of *independent query sampling* can be integrated with any reporting queries (e.g., multidimensional range reporting, stabbing queries on intervals, half-plane reporting, etc.), and defines a new variant for every individual problem. All these variants are expected to play increasingly crucial roles in countering the big-query issue. The techniques developed in this paper pave the foundation for further studies in this line of research.

2. RAM STRUCTURES

Our discussion will assume WR queries by default because as we will see there is an efficient reduction from WoR to WR. Recall that a query specifies two parameters: a range $q = [x, y]$ and the number t of samples. We say that the query is *one-sided* if $x = -\infty$ or $y = \infty$; otherwise, the query is *two-sided*. Next, we will first describe a structure for one-sided queries, before attending to two-sided ones.

2.1 A One-Sided Structure

Structure. We build a *weight-balanced B-tree* (WBB-tree) [3] on the input set P with leaf capacity $b = 4$ and branching parameter $f = 8$. In general, a WBB-tree parameterized by b and f is a B-tree where

- data elements are stored in the leaves. We label the leaf level as level 0; if a node is at level i, then its parent is at level $i + 1$.
- a non-root node u at the i-th level has between $bf^i/4$ and bf^i elements stored in its subtree. We denote by $P(u)$ the set of those elements. This property implies that an internal node has between $f/4$ and $4f$ child nodes.

Each node u is naturally associated with an interval $I(u)$ defined as follows. If u is a leaf, then $I(u) = (e', e]$ where e (or e', resp.) is the largest element stored in u (or the leaf preceding u, resp.); specially, if no leaf precedes u, then $e' = -\infty$. If u is an internal node, then $I(u)$ unions the intervals of all the child nodes of u.

Let z_ℓ be the leftmost leaf (i.e., the leaf containing the smallest element of P). Denote by Π_ℓ the path from the root to z_ℓ. For every node u on Π_ℓ, store all the elements of $P(u)$ in an array $A(u)$. Note that the element ordering in $A(u)$ is arbitrary. The total space of all arrays is $O(n)$, noticing that the arrays' sizes shrink geometrically as we descend Π_ℓ.

Query. A one-sided query with parameters $q = (-\infty, y]$ and t is answered as follows. We first identify the lowest node u on Π_ℓ such that $I(u)$ fully covers q. If u is a leaf, we obtain the entire $P(q) = P \cap q$ from u in constant time, after which the samples can be obtained trivially in $O(t)$ time. If u is an internal node, we obtain a sequence $\mathcal{R}$ by repeating the next step until the length of $\mathcal{R}$ is t: select uniformly at random an element e from $A(u)$, and append e to $\mathcal{R}$ if e is covered by q. We return $\mathcal{R}$ as the query's output. Note that the $\mathcal{R}$ computed this way is independent from all the past queries.

We argue that the above algorithm runs in $O(\log\log n+t)$ expected time, focusing on the case where u is not a leaf. Let $k = |P(q)|$. Node u can be found in $O(\log\log n)$ time by creating a binary search tree on the intervals of the nodes on Π_ℓ. It is easy to see that the size of $A(u)$ is at least k but at most ck for some constant $c \geq 1$. Hence, a random sample e from $A(u)$ has at least $1/c$ probability of falling in q. This implies that we expect to sample no more than $ct = O(t)$ times before filling up $\mathcal{R}$.

Update. Recall the well-known fact that an array can be maintained in $O(1)$ time per insertion and deletion[1]—this is true even if the array's size needs to grow or shrink—provided that the element ordering in the array does not matter. The key to updating our structure lies in modifying the secondary arrays along Π_ℓ. Whenever we insert/delete an element e in the subtree of a node u on Π_ℓ, e must be inserted/deleted in $A(u)$ as well. Insertion is easy: simply append e to $A(u)$. To delete e, we first locate e in $A(u)$, swap it with the last element of $A(u)$, and then shrink the size of $A(u)$ by 1. The problem, however, is how to find the location of e; although hashing does this trivially, the update time becomes $O(\log n)$ expected.

The update time can be made worst case by slightly augmenting our structure. For each element $e \in P$, we maintain a linked list of all its positions in the secondary arrays. This linked list is updated in constant time whenever a position changes (this requires some proper bookkeeping, e.g., pointers between a position in an array and its record in a linked list). In this way, when e is deleted, we can find all its array positions in $O(\log n)$ time. Taking care of other standard details of node balancing (see [3]), we have arrived at:

THEOREM 1. *For the IRS problem, there is a RAM structure of $O(n)$ space that can answer a one-sided WR query in $O(\log\log n+t)$ expected time, and can be updated in $O(\log n)$ worst-case time per insertion and deletion.*

2.2 A 2-Sided Structure of $O(n \log n)$ Space

By applying standard range-tree ideas to the one-sided structure in Theorem 1, we obtain a structure for two-sided queries with space $O(n \log n)$ and query time $O(\log n + t)$ expected. However, it takes $O(\log^2 n)$ time to update the structure. Next, we give an alternative structure with improved update cost.

[1]A deletion needs to specify where the target element is in the array.

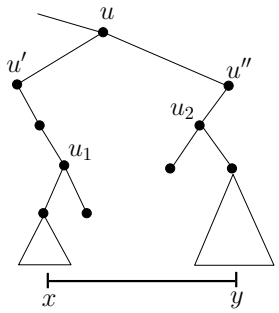

Figure 1: Answering a query at two nodes

Structure. Again, we build a WBB-tree T on the input set P with leaf capacity $b = 4$ and branching parameter $f = 8$. At each node u in the tree, we keep a count equal to $|P(u)|$, i.e., the number of elements in its subtree. We also associate u with an array $A(u)$ that stores all the elements of $P(u)$; the ordering in $A(u)$ does not matter. The overall space consumption is clearly $O(n \log n)$.

Query. We will see how to use the structure to answer a query with parameters $q = [x, y]$ and t. Let $k = |P(q)|$. Since we aim at query time of $\Omega(\log n)$, it suffices to consider only $k > 0$ (one can check whether $k > 0$ easily with a separate "range count" structure). The crucial step is to find at most two nodes u_1, u_2 satisfying two conditions:

- ***c*1** $I(u_1)$ and $I(u_2)$ are disjoint, and their union covers q;
- ***c*2** $|P(u_1)| + |P(u_2)| = O(k)$.

These nodes can be found as follows. First, identify the lowest node u in T such that $I(u)$ covers q. If u is a leaf node, setting $u_1 = u$ and $u_2 = nil$ satisfies both conditions.

Now, suppose that u is an internal node. If q spans the interval $I(u')$ of at least one child u' of u, then once again setting $u_1 = u$ and $u_2 = nil$ satisfies both conditions. Now, consider that q does not span the interval of any child of u. In this case, x and y must fall in the intervals of two consecutive child nodes u', u'' of u, respectively. Define $q_1 = q \cap I(u')$ and $q_2 = q \cap I(u'')$. We decide u_1 (u_2, resp.) as the lowest node in the subtree of u' (u'', resp.) whose interval covers q_1 (q_2, resp.); see Figure 1 for an illustration. The lemma below shows that our choice is correct.

LEMMA 1. *The u_1 and u_2 we decided satisfy conditions **c1** and **c2**.*

PROOF. We will focus on the scenario where u is an internal node. Let k_1 (k_2, resp.) be the number of elements in the subtree of u' (u'', resp.) covered by q. Clearly, $k = k_1 + k_2$. It suffices to show that $|P(u_1)| = O(k_1)$ and $|P(u_2)| = O(k_2)$. We will prove only the former due to symmetry. In fact, if u_1 is a leaf, then both k_1 and $|P(u_1)|$ are $O(1)$. Otherwise, q definitely spans the interval of a child node, say $\hat{u}$, of u_1. Hence, $|P(u_1)| = O(|P(\hat{u})|) = O(k_1)$. □

Let us continue the description of the query algorithm, given that u_1 and u_2 are already found. We conceptually append $A(u_1)$ to $A(u_2)$ to obtain a concatenated array A. Then, we repetitively perform the following step until an initially empty sequence $\mathcal{R}$ has length t: sample uniformly at random an element e from A, and append e to $\mathcal{R}$ if it lies in q. Note that since we know both $|A(u_1)|$ and $|A(u_2)|$, each sample can be obtained in constant time. Since A has size $O(k)$ and at least k elements covered by q, we expect to sample $O(t)$ elements before filling up $\mathcal{R}$. The total query cost is therefore $O(\log n + t)$ expected.

Update. The key to updating our structure is to modify the secondary arrays, as can be done using the ideas explained in Section 2.1 for updating our one-sided structure. The overall update time is $O(\log n)$.

LEMMA 2. *For the IRS problem, there is a RAM structure of $O(n \log n)$ space that can answer a two-sided WR query in $O(\log n + t)$ expected time, and can be updated in $O(\log n)$ worst-case time per insertion and deletion.*

2.3 A 2-Sided Structure of $O(n)$ Space

In this subsection, we improve the space of our two-sided structure to linear using a two-level sampling idea.

Structure. Let s be an integer between $\log_2 n - 1$ and $\log_2 n + 1$. We divide the domain $\mathbb{R}$ into a set $\mathcal{I}$ of $g = \Theta(n/\log n)$ disjoint intervals $\mathcal{I}_1, ..., \mathcal{I}_g$ such that each $\mathcal{I}_i$ ($1 \leq i \leq g$) covers between $s/2$ and s points of P. Define $\mathcal{C}_i = \mathcal{I}_i \cap P$, and call it a *chunk*. Store the points of each $\mathcal{C}_i$ in an array (i.e., one array per chunk).

We build a structure T of Lemma 2 on $\{\mathcal{I}_1, ..., \mathcal{I}_g\}$. T allows us to sample at the chunk level, when given a query range $q^* = [x^*, y^*]$ aligned with the intervals' endpoints (in other words, q^* equals the union of several consecutive intervals in $\mathcal{I}$). More specifically, given a query with such a range q^* and parameter t, we can use T to obtain a sequence S of t chunk ids, each of which is taken uniformly at random from the ids of the chunks whose intervals are covered by q^*. We slightly augment T such that whenever a chunk id i is returned in S, the chunk size $|\mathcal{C}_i|$ is always returned along with it. The space of T is $O(g \log g) = O(n)$.

We will also need a rank structure on P, which (as explained in Section 1) allows us to obtain t samples from any query range in $O(t \log n)$ time.

Query. We answer a query with parameters $q = [x, y]$ and t as follows. First, in $O(\log n)$ time, we can identify the intervals $\mathcal{I}_i$ and $\mathcal{I}_{i'}$ that contain x and y, respectively. If $i = i'$, we answer the query bruteforce by reading all the $O(\log n)$ points in $\mathcal{C}_i$.

If $i \neq i'$, we break q into three disjoint intervals $q_1 = [x, x^*]$, $q_2 = [x^*, y^*]$, and $q_3 = [y^*, y]$, where x^* (y^*, resp.) is the right (left, resp.) endpoint of $\mathcal{I}_i$ ($\mathcal{I}_{i'}$, resp.). In $O(\log n)$ time (using the rank structure on P), we can obtain the number of data points in the three intervals: $k_1 = |q_1 \cap P|$, $k_2 = |q_2 \cap P|$, and $k_3 = |q_3 \cap P|$. Let $k = k_1 + k_2 + k_3$.

We now determine the numbers t_1, t_2, t_3 of samples to take from q_1, q_2, and q_3, respectively. To do so, generate t random integers in $[1, k]$; t_1 equals how many of those integers fall in $[1, k_1]$, t_2 equals how many in $[k_1 + 1, k_1 + k_2]$, and t_3 how many in $[k_1 + k_2 + 1, k]$. We now proceed to take the desired number of samples from each interval (we will clarify how to do so shortly). Finally, we randomly permute the t samples in $O(t)$ time, and return the resulting permutation.

Sampling t_1 and t_3 elements from q_1 and q_3 respectively can be easily done in $O(\log n)$ time. Next, we concentrate on taking t_2 samples from q_2. If $t_2 \leq 6 \ln 2$, we simply obtain t_2 samples from the rank structure in $O(t_2 \log n) = O(\log n)$

time. For $t_2 > 6 \ln 2$, we first utilize T to obtain a sequence S of $4t_2$ chunk ids for the range $q_2 = [x^*, y^*]$. We then generate a sequence $\mathcal{R}$ of samples as follows. Take the next id j from S. Toss a coin with head probability $|\mathcal{C}_j|/s$.[2] If the coin tails, do nothing; otherwise, append to $\mathcal{R}$ a point selected uniformly at random from $\mathcal{C}_j$. The algorithm finishes as soon as $\mathcal{R}$ has collected t_2 samples. It is possible, however, that the length of $\mathcal{R}$ is still less than t_2 even after having processed all the $4t_2$ ids in S. In this case, we restart the *whole* query algorithm from scratch.

We argue that the expected cost of the algorithm is $O(\log n + t)$. As $|\mathcal{C}_j|/s \geq 1/2$ for any j, the coin we toss in processing S heads at least $4t_2/2 = 2t_2$ times in expectation. A simple application of Chernoff bounds shows that the probability it heads less than t_2 times is at most $1/2$ when $t_2 > 6 \ln 2$. This means that the algorithm terminates with probability at least $1/2$. Each time the algorithm is repeated, its cost is bounded by $O(\log n + t)$ (regardless of whether another round is needed). Therefore, overall, the expected running time is $O(\log n + t)$.

Update. T is updated whenever a chunk (either its interval or the number of points therein) changes. This can be done in $O(\log n)$ time per insertion/deletion of a point in P. A chunk overflow (i.e., size over s) or underflow (below $s/2$) can be treated in $O(s)$ time by a chunk split or merge, respectively. Standard analysis shows that each update bears only $O(1)$ time amortized. Finally, to make sure s is between $\log_2 n - 1$ and $\log_2 n + 1$, we rebuild the whole structure whenever n has doubled or halved, and set $s = \log_2 n$. Overall, the amortized update cost is $O(\log n)$. The amortization can be removed by standard techniques [13]. We have now established:

THEOREM 2. *For the IRS problem, there is a RAM structure of $O(n)$ space that can answer a two-sided WR query in $O(\log n + t)$ expected time, and can be updated in $O(\log n)$ worst-case time per insertion and deletion.*

2.4 Reduction from WoR to WR

We will need the fact below (see appendix for a proof):

LEMMA 3. *Let S be a set of k elements. Consider taking $2s$ samples uniformly at random from S with replacement, where $s \leq k/(3e)$. The probability that we get at least s distinct samples is at least $1/2$.*

A two-sided WoR query with parameters q, t on dataset P can be answered using a structure of Theorem 2 as follows. First, check whether $t \geq k/(3e)$ where $k = |P(q)|$ can be obtained in $O(\log n)$ time. If so, we run a sampling WoR algorithm (e.g., [15]) to take t samples from $P(q)$ directly, which requires $O(\log n + k) = O(\log n + t)$ time. Otherwise, we run a WR query with parameters $q, 2t$ to obtain a sequence $\mathcal{R}$ of samples in $O(\log n + t)$ expected time. If $\mathcal{R}$ has at least t distinct samples (which can be checked in $O(t)$ expected time using hashing), we collect all these samples into a set S, and sample WoR t elements from S; the total running time in this case is $O(\log n + t)$. On the other hand, if $\mathcal{R}$ has less than t distinct elements, we repeat the above by issuing another WR query with parameters $q, 2t$. By Lemma 3, a repeat is necessary with probability at most $1/2$. Therefore, overall the expected query time remains $O(\log n + t)$.

Similarly, a one-sided WoR query can be answered using a structure of Theorem 1 in $O(\log \log n + t)$ expected time.

[2]This can be done without division: generate a random integer in $[1, s]$ and check if it is smaller than or equal to $|\mathcal{C}_j|$.

3. A TIGHT I/O LOWER BOUND

Having gained some experience about IRS, we now turn our attention to external memory, where the problem is much more challenging. The first question we will answer is: would it be possible to achieve a query cost such as $O(\log_B n + t/B)$ with a small-space structure, especially given the RAM result in Theorem 2?

For this purpose, it suffices to look at a special instance of the IRS problem where all queries' search intervals are fixed to $q = (-\infty, \infty)$. Formally, our objective is to preprocess a set S of n elements such that all queries of the following form can be answered efficiently: given an integer $t \geq 1$, randomly sample t elements WR or WoR from S. We refer to this special instance as the *set sampling problem*.

We will prove a query cost lower bound under the *indivisibility assumption* that every element in S must always be stored as an atom. This assumption was first introduced to prove a sorting lower bound in EM [2] (which still remains the best today), and is a key assumption in the *indexability model* [9] which encapsulates almost all the existing lower bounds on EM data structures. Specifically, our result is:

THEOREM 3. *Let c be an arbitrary constant. For the set sampling problem, when $n \geq B^2$, every structure occupying at most n^c blocks must incur*

$$\Omega\left(\min\left\{t, \frac{t}{B}\log_{M/B}\frac{n}{B}\right\}\right)$$

amortized I/Os in expectation answering:

- *a WR query with parameter $t \in [B, n]$.*
- *a WoR query with parameter $t \in [B, n^{1-\epsilon}]$, where $\epsilon > 0$ is an arbitrarily small constant.*

Notice that the lower bound is on amortized query cost (and hence, stronger than a worst-case lower bound). In other words, a structure is allowed to balance the work among queries in an attempt to keep the cost low on average, but the lower bound says that the average cost must still be high. Furthermore, note that the bound holds rules out any structure of $O(n^c)$ space on the IRS problem with query cost $O(\log_B n + t/B)$—such a structure can be used to answer a set sampling query with $t \geq B \log_B n$ in $O(t/B)$ I/Os.

Also notice that the ranges of t are different for WR and WoR queries in Theorem 3. In fact, the two types of queries are indeed separated in terms of hardness for t close to n. This is most intuitive when $t = n$, in which case a (set sampling) WoR query can simply return the entire dataset S in $O(n/B)$ I/Os, whereas a WR query must (essentially) produce a random permutation of S, and incur $\Omega((n/B)\log_{M/B}(n/B))$ I/Os as shown in the theorem.

As a second step, we will match the lower bound with a set sampling structure:

THEOREM 4. *For the set sampling problem, there is a structure of $O(n/B)$ space that answers a WR or WoR query in $O(1 + (t/B)\log_{M/B}(n/B))$ amortized I/Os.*

It is worth mentioning that one can always answer a WR query in $O(t)$ I/Os by sampling directly from an array (i.e., just apply a RAM algorithm by ignoring blocking), and answer a WoR query in $O(t)$ expected I/Os by resorting to the reduction in Section 2.4. Thus, both cases of the query cost in Theorem 3 are tight. The rest of the section gives our proofs for the two theorems.

3.1 Proof of Theorem 3

WoR Lower Bound for $t \in [B, n^{1-\epsilon}]$. We will first prove the WoR branch of Theorem 3. At a high level, our proof works as follows. We issue m queries of the same parameter t for some large m to be chosen later, and ask them to write their retrieved sample sets sequentially in the disk. At the end, we get a *final sequence* of m sample sets[3]. Because of queries' independence, the number of possible final sequences is huge. On the other hand, if the amortized query cost has to be low, then the query algorithm can perform only a small number of I/Os in total, such that it will not be able to produce that many possible final sequences. This puts a constraint on how low the query cost can possibly be. The complication, however, is that the data structure is randomized (no deterministic structure can solve the problem). To make the above idea work, we need to argue that the structure must nonetheless still produce numerous final sequences from at least one "initial state".

We will regard each block as a set of elements (i.e., ignoring the ordering of the elements therein). Denote by Σ the set of all possible final sequences. It is easy to see that $|\Sigma| = \binom{n}{t}^m$. We define an *initial state* of a randomized data structure as the sequence of non-empty blocks in memory followed by those in the disk at the time the structure answers the first query. Let Π be the set of all possible initial states. Since an initial state has at most $M/B + n^c < n^{c+1}$ blocks, we know that $|\Pi| < \binom{n}{B}^{n^{c+1}}$.

Given a final sequence $\sigma \in \Sigma$ and an initial state $\pi \in \Pi$, define $cost(\sigma, \pi)$ as the minimum I/O cost of all possible algorithms (of an indivisible structure) to produce σ, when the memory and disk currently have the contents π. For each σ, define its best initial state—denoted as π_σ—as the π with the smallest $cost(\sigma, \pi)$.

Let H be the expected cost of the structure in processing all the m queries we issued. It suffices to consider $H \leq nm$.[4] We observe:

LEMMA 4. $\sum_{\sigma\in\Sigma} cost(\sigma, \pi_\sigma) \leq H \cdot |\Sigma|$.

PROOF. Let X be a random variable that equals the final sequence produced by the structure. Since queries are independent and each query returns a size-t WoR sample set of S, we know that $\mathbf{Pr}[X = \sigma] = 1/|\Sigma|$ for each $\sigma \in \Sigma$. Let Y be another random variable that equals the actual number of I/Os the structure performs. We know that

$$H = \mathbf{E}[Y] = \sum_{\sigma\in\Sigma} \mathbf{E}[Y \mid X = \sigma] \cdot \mathbf{Pr}[X = \sigma].$$

By definition, $cost(\sigma, \pi_\sigma) \leq \mathbf{E}[Y \mid X = \sigma]$. The lemma then follows. □

Let Σ^* be the set of all such $\sigma \in \Sigma$ that $cost(\sigma, \pi_\sigma) \leq 2H$. Lemma 4 implies that $|\Sigma^*| \geq |\Sigma|/2$. Given an initial state $\pi \in \Pi$, let its *power*—denoted as $power(\pi)$—be the number of final sequences $\sigma \in \Sigma^*$ that finds π as their best initial state, namely, $\pi = \pi_\sigma$. Let π^* be the initial state with the greatest power. Obviously:

$$power(\pi^*) \quad \geq \quad |\Sigma^*|/|\Pi|.$$

Let us define the *final state* of a structure as the combination of (i) the set of elements in memory, and (ii) the sequence of occupied blocks in the disk, both at the moment when it has answered all m queries. Starting from π^*, how many different final states can we leave the structure in after $2H$ I/Os? By a standard permutation argument [2], the answer is at most $(2(n^c + 2H)\binom{M}{B})^{2H} \leq (2(n^c + 2nm)\binom{M}{B})^{2H}$. As this number must be at least $power(\pi^*)$, we have:

$$\begin{aligned}
\left(2(n^c + 2nm)\binom{M}{B}\right)^{2H} &\geq |\Sigma^*|/|\Pi| \\
&\geq \frac{(1/2)\binom{n}{t}^m}{\binom{n}{B}^{n^{c+1}}} \\
(\text{by } t \in [B, n^{1-\epsilon}]) &\geq \frac{1}{2}\binom{n}{t}^{m-n^{c+1}}.
\end{aligned}$$

Now we fix $m = 2n^{c+1}$ so that the above gives

$$\begin{aligned}
\left(n^{c+3}\binom{M}{B}\right)^{2H} &\geq \frac{1}{2}\binom{n}{t}^{m/2} \\
\Rightarrow \frac{H}{m} &\geq \frac{(t/8)\ln(n/t)}{(c+3)\ln n + B(1 + \ln(M/B))}.
\end{aligned}$$

Note that $\ln(n/t) = \Omega(\log n)$, and that when $n \geq B^2$, $\ln(n/B) = \Theta(\log n)$. Hence, the above inequality implies that $H/m = \Omega(\min\{t, (t/B)\log_{M/B}(n/B)\})$ as claimed.

WR Lower Bound for $t \in [B, n]$. We now prove the WR branch of Theorem 3. Let us start with $t = B$. In fact, this follows immediately from the reduction in Section 2.4; namely, when $n \geq B^2$, if there is an algorithm answering a WR query with $t = B$ in H amortized expected I/Os, we can apply the algorithm to answer a WoR query with $t = B$ in $O(H)$ amortized expected I/Os. Hence, a WR query with $t = B$ must incur $\Omega(\min\{B, \log_{M/B}(n/B)\})$ amortized cost in expectation.

Now consider any $t \in [B, n]$. Observe that if an algorithm can answer a WR query with parameter $t = \tau > B$ in H amortized expected I/Os, we can use it to answer a WR query with $t = B$ in $O(HB/\tau)$ amortized expected I/Os. This is because by running a $t = \tau$ query once we get enough samples for the next $\lfloor \tau/B \rfloor$ queries of $t = B$. With this, we have completed the whole proof of Theorem 3.

3.2 Proof of Theorem 4

We prove the theorem by developing such a structure. We store all the elements of S in an arbitrary order using an array. In addition, we also take n samples WR from S, and store these samples in a separate array A, which is called the

[3]Note that this is a sequence of *sets*; the ordering of the elements in the same sample set does not matter.

[4]Otherwise, $H/m > n > t$, in which case our claim $\Omega(\min\{t, (t/B)\log_{M/B}(n/B)\})$ already holds.

sample pool. With sorting, all these samples can be obtained in $O((n/B)\log_{M/B}(n/B))$ I/Os.

When the structure is newly built, all samples are marked as *clean*. To answer a query with parameter t, we simply return the next t clean samples from the pool, and mark them *dirty*. When the pool runs out of clean samples, we rebuild it in $O((n/B)\log_{M/B}(n/B))$ I/Os. The cost can be amortized on the n samples already returned, so that each of them is charged only $O((1/B)\log_{M/B}(n/B))$ I/Os. The amortized query cost is therefore $O(1+(t/B)\log_{M/B}(n/B))$.

4. I/O-EFFICIENT IRS STRUCTURES

In this section, we present I/O-efficient structures for solving the IRS problem, focusing on the scenario where the input set P is static. We will discuss only WR queries because the extension to WoR queries is straightforward by the reduction in Section 2.4.

4.1 A One-Sided Structure

The one-sided RAM structure in Section 2.1 can be adapted to work in EM, by still using a constant fanout for the base tree, and replacing each secondary array with a set sampling structure of Theorem 4. It answers an IRS query in $O(\log_B \log_2 n + (t/B)\log_{M/B}(n/B))$ amortized expected I/Os[5]. Next, we improve the query cost to $O(\log_B \log_B n + (t/B)\log_{M/B}(n/B))$. The improvement owes to a new idea we call *sample replenishing*. This idea also lies at the core of our other EM structures.

Structure. We build a B-tree T on P with leaf capacity and branching parameter both set to B. Let Π_ℓ be the path from the root to the leftmost leaf. For each node u that either is the root or has its parent on Π_ℓ, we build a set sampling structure $\mathcal{T}(u)$ of Theorem 4 on $P(u)$ (the set of elements in the subtree of u). All the set sampling structures occupy $O(n/B)$ space in total.

Given an internal node u, we use $u[i]$ to denote the i-th child node of u, counting from the left. Consider now u as an internal node on Π_ℓ with f child nodes. For each $i \in [1, f]$, we record in u the value of $|P(u[i])|$. We also store $\log_2 f$ bags (i.e., multi-sets) of samples at u but of different sizes such that in total all the samples occupy $O(f)$ blocks. Specifically, for each $l = 1, 2, ..., \log_2 f$, we store a bag $\mathcal{S}_u(l)$ of $2^l B$ samples, each of which is an element randomly taken WR from $\bigcup_{i=1}^{2^l} P(u[i])$. All samples are marked as clean at this point. It is easy to see that the overall space of our structure still remains $O(n/B)$.

Query. As before, given a node u in T, we use $I(u)$ to represent the interval associated with u. To answer a one-sided IRS query with parameters $q = (-\infty, x]$ and t, we first identify in $O(\log_B \log_B n)$ I/Os the lowest node u on Π_ℓ such that $I(u)$ covers q. Let l be the smallest integer such that q is contained in the union of the intervals of the leftmost 2^l child nodes of u. We repeat the following step to generate a bag $\mathcal{R}$ of t samples: take the next clean sample e from $\mathcal{S}_u(l)$, and add it to $\mathcal{R}$ if e falls in q. We mark e dirty regardless of whether it has been added to $\mathcal{R}$. Due to the choice of l, we expect to repeat the step at most $2t$ times. The total query cost is therefore $O(\log_B \log_B n + t/B)$ expected so far.

[5]We adopt the convention that $O(\log_b x)$ should be understood as $O(\max\{1, \log_b x\})$.

When $\mathcal{S}_u(l)$ runs out of clean samples, we launch a *replenishing process* to re-compute a new bag $\mathcal{S}_u(l)$ of $2^l B$ samples. For this purpose, we first determine how many samples to get from each child node of u, by generating $2^l B$ random ranks from 1 to $\sum_{i=1}^{2^l} |P(u[i])|$. Then, for every $u[i]$ with $i \in [1, 2^l]$, we take the desired number of samples from $P(u[i])$ using the set sampling structure $\mathcal{T}(u[i])$. By Theorem 4, doing so to all the 2^l child nodes requires $O(2^l + (2^l B/B)\log_{M/B}(n/B)) = O(2^l \log_{M/B}(n/B))$ amortized I/Os in total. Finally, we generate a random permutation of the $2^l B$ samples in $O(2^l \log_{M/B}(n/B))$ I/Os [7], and store the permutation as the newly computed $\mathcal{S}_u(l)$. All these samples are marked as clean.

Overall, replenishing $\mathcal{S}_u(l)$ takes $O(2^l \log_{M/B}(n/B))$ amortized I/Os. However, $2^l B$ samples must have been reported from $\mathcal{S}_u(l)$ since its last replenishment. We can thus amortize the cost over those samples, so that each of them bears only $O((1/B)\log_{M/B}(n/B))$ I/Os. Therefore, the amortized query cost is bounded by $O(\log_B \log_B n + (t/B)\log_{M/B}(n/B))$ expected.

THEOREM 5. *For the IRS problem, there is a structure of $O(n/B)$ space that answers a one-sided WR or WoR query in $O(\log_B \log_B n + (t/B)\log_{M/B}(n/B))$ amortized I/Os in expectation.*

4.2 A 2-Sided Structure of Near-Linear Space

In this subsection, we will give a structure of $O((n/B)\log^\star(n/B))$ space that answers a 2-sided IRS query in $O(\log_B n + (t/B)\log_{M/B}(n/B))$ amortized expected I/Os. We achieve this by utilizing our set sampling and 1-sided structures, and the sample replenishing technique.

Structure. We will build a tree T on P of $h = 1+\log^\star(n/B)$ levels where the fanout decreases very rapidly as we descend the tree. First, create a root node, and associate it with all the points in P. In general, a node u at level $h - i$ ($0 \le i \le h$) is associated with a set $P(u)$ of at most $2B\log^{(i)}(n/B)$ points in an interval $I(u)$ satisfying:

- $P(u) = I(u) \cap P$
- the intervals of all nodes at the same level are disjoint, and their union is $\mathbb{R}$.

If u is the root, $i = 0$, $P(u) = P$, and $I(u) = \mathbb{R}$. For $i < h$, we obtain the child nodes of u as follows. Divide $I(u)$ into a set of intervals, such that each interval covers exactly $\lfloor B\log^{(i+1)}(n/B)\rfloor$ points in $P(u)$, except possibly the last one which is allowed to have between $\lfloor B\log^{(i+1)}(n/B)\rfloor$ and $2\lfloor B\log^{(i+1)}(n/B)\rfloor$ points. Create a child node of u for each interval, and associate it with the points covered.

Now, consider an internal node u in T. Let $u[1], ..., u[f]$ be the child nodes of u (the value of f depends on the level of u). We store $|P(u)|$ at u, and associate u with some secondary structures:

- A set sampling structure of Theorem 4 on $P(u)$.
- A one-sided structure of Theorem 5 on $P(u)$.
- A binary search tree $\mathcal{T}(u)$ on the intervals $I(u[1]), ..., I(u[f])$. Let v be a node in $\mathcal{T}(u)$, and suppose that $I(u[j]), ..., I(u[j'])$ are the intervals covered in the subtree of v. Define $I(v)$ as the union

of these intervals, and $P(v) = P(u[j]) \cup ... \cup P(u[j'])$ (clearly, $P(v) = I(v) \cap P$). We store at v the value of $|P(v)|$, and a bag $\mathcal{S}(v)$ of $(j' - j + 1)B$ samples, each of which is taken randomly WR from $P(v)$.

To analyze the space, let u be an internal node at level $h - i$. The number f of its child nodes is at most $\frac{2B \log^{(i)}(n/B)}{\lfloor B \log^{(i+1)}(n/B) \rfloor} = O(\frac{\log^{(i)}(n/B)}{\log^{(i+1)}(n/B)})$. $\mathcal{T}(u)$ as well as all the sample bags in its nodes occupy $O(f \log f)$ blocks. As there are at most $n/\lfloor B \log^{(i)}(n/B) \rfloor$ nodes at level $h - i$, all their secondary structures occupy

$$\begin{aligned} & \frac{n}{\lfloor B \log^{(i)}(n/B) \rfloor} \cdot O(f \log f) \\ = \; & O\left(\frac{n}{B \log^{(i)}(n/B)} \cdot \frac{\log^{(i)}(n/B)}{\log^{(i+1)}(n/B)} \log \log^{(i)}(n/B) \right) \\ = \; & O(n/B). \end{aligned}$$

Therefore, the overall space is $O((n/B) \log^\star(n/B))$.

Query. To answer a 2-sided query with parameters $q = [x, y]$ and t, we first identify the lowest node u in T such that $I(u)$ covers q. This can be done in $O(\log_B n)$ I/Os, because the fanout is exponentially lower every level down. If u is a leaf, we answer the query by reading all the $O(B)$ points in $P(u)$.

If u is an internal node, then based on the intervals of the child nodes of u, we can break the query into two 1-sided queries (which will be answered at two child nodes of u), and a 2-sided query with a range q^* that is aligned with the intervals' endpoints. We can process each query separately, after determining appropriately how many samples to take from them, respectively, and then perform a random permutation on all the collected t samples in $O((t/B) \log_{M/B}(t/B))$ I/Os. The samples are then output in the permuted order. Given Theorem 5, it suffices to explain how to handle the query with q^*.

Suppose that we want to take t^* samples from q^*. Let $k^* = |P \cap q^*|$. Using the approach explained in Section 2.2, we can identify two nodes v_1, v_2 in $\mathcal{T}(u)$ such that (i) $I(v_1)$ and $I(v_2)$ are disjoint, and their union covers q, and (ii) $|P(v_1)| + |P(v_2)| = O(k^*)$. This can be done in $O(\log_B n)$ I/Os by applying the standard idea of packing the routing information of multiple nodes of $\mathcal{T}(u)$ in a block. We then generate a bag $\mathcal{R}$ of t^* samples by repeating the following two-step procedure. First, take t^* samples WR from $P(v_1) \cup P(v_2)$ (we will discuss how to do so shortly). Second, add to $\mathcal{R}$ all the samples falling in q^*. We can fill up $\mathcal{R}$ by running the procedure only $O(1)$ times in expectation.

Next, we clarify how to obtain t^* samples from $P(v_1) \cup P(v_2)$. From $|P(v_1)|$, $|P(v_2)|$, and t^*, we can decide how many samples to take from $P(v_1)$ and $P(v_2)$, respectively, by generating random ranks in memory. Then, we simply pull the desired number of samples from $\mathcal{S}(v_1)$ and $\mathcal{S}(v_2)$, respectively. If $\mathcal{S}(v_1)$ has been exhausted (the case with $\mathcal{S}(v_2)$ is similar), we perform sample replenishing as follows. Let $u[j]$, ..., $u[j']$ be the child nodes of u whose intervals are covered by $I(v_1)$. Recall that $\mathcal{S}(v_1)$ had $(j' - j + 1)B$ samples after it was newly computed. We obtain a new $\mathcal{S}(v_1)$ of this size from $P(u[j]) \cup ... \cup P(u[j'])$ in two steps:

1. Decide how many samples to take from each of $P(u[j]), ..., P(u[j'])$. For this purpose, generate $(j' - j + 1)B$ random ranks from 1 to $|P(v_1)|$, sort them, and see how many ranks fall in each of those subtrees according to the values of $|P(u[j])|, ..., |P(u[j'])|$. This can be done in $O((j' - j + 1) \log_{M/B}(n/B))$ I/Os.
2. Query the set sampling structures of $u[j]$, ... $u[j']$ to get the desired numbers of samples, and then carry out a random permutation over all the samples. This demands $O((j' - j + 1) \log_{M/B}(n/B))$ amortized I/Os.

Each sample already output is thus amortized only $O((1/B) \log_{M/B}(n/B))$ I/Os. This completes the description of our query algorithm.

LEMMA 5. *For the IRS problem, there is a structure of $O((n/B) \log^\star(n/B))$ space that answers a two-sided WR or WoR query in $O(\log_B n + (t/B) \log_{M/B}(n/B))$ amortized I/Os in expectation.*

4.3 Two-Level IRS

This subsection tackles a separate problem which we call *two-level IRS*, whose solution is the key to obtaining an $O(n/B)$-space structure for 2-sided IRS queries.

In the two-level IRS problem, we have a set $\mathcal{I}$ of g disjoint intervals $\mathcal{I}_1, ..., \mathcal{I}_g$ whose union is $\mathbb{R}$. Each interval $\mathcal{I}_i$ ($1 \leq i \leq g$) covers a set $\mathcal{C}_i$ of points (whose concrete locations within $\mathcal{I}_i$ are irrelevant). We refer to $\mathcal{C}_i$ as a *chunk*. Define $P = \bigcup_{i=1}^{g} \mathcal{C}_i$, and $n = |P|$. A query specifies an integer $t \geq 1$ and a range $q = [x, y]$ that is always aligned with the intervals' endpoints (namely, q always equals the union of several consecutive intervals in $\mathcal{I}$). The query returns t samples each taken randomly WR from $P(q) = P \cap q$.

One can apply Lemma 5 to solve this problem, but we aim to meet a different space budget:

LEMMA 6. *For the two-level IRS problem, there is a structure of $O(n/B + g \log^\star g)$ space that answers a query in $O(\log_B n + (t/B) \log_{M/B}(n/B))$ amortized I/Os in expectation.*

The rest of the subsection proves the lemma by describing such a structure.

Structure. We create a set sampling structure of Theorem 4 on each $\mathcal{C}_i$ ($i \in [1, g]$). They occupy $O(g + n/B)$ blocks in total. We also need a separate structure at the chunk level, which is similar to the one in Section 4.2, but differs in the secondary structures, as explained next.

We build a tree T on $\mathcal{I}$ of $h = 1 + \log^\star g$ levels such that a node u at level $h - i$ ($0 \leq i \leq h$) is associated with an interval $I(u)$ with all the properties below:

- $I(u)$ is the union of between $\lfloor \log^{(i)} g \rfloor$ and $2 \log^{(i)} g$ consecutive intervals in $\mathcal{I}$.
- The nodes at the same level have disjoint intervals, whose union is $\mathbb{R}$.
- The interval of an internal node is always the union of its child nodes' intervals.

Define $P(u) = I(u) \cap P$.

Now consider an internal node u. Let $\mathcal{I}_j, \mathcal{I}_{j+1}, ..., \mathcal{I}_{j'}$ be the intervals covered by $I(u)$ in ascending order, and $\lambda = j' - j + 1$. We associate u with several secondary structures:

- The first one allows us to do set sampling on $P(u)$, that is, to extract t random samples WR from $P(u)$ in $O(1 + (t/B)\log_{M/B}(n/B))$ amortized I/Os. Our aim, however, is to use only $O(\lambda)$ blocks; thus, we cannot simply create a set sampling structure of Theorem 4 on $P(u)$. Instead, we store a pool of λB samples from $P(u)$. Upon a set sampling request, we simply return the next t samples from the pool. When the pool is exhausted, a new pool of λB samples is generated by querying the set sampling structures of chunks $\mathcal{C}_j, \mathcal{C}_{j+1}..., \mathcal{C}_{j'}$, and then performing a random permutation in $O(\lambda \log_{M/B}(n/B))$ amortized I/Os.

- The second one allows us to answer a one-sided query on $P(u)$. The space budget is still $O(\lambda)$, and thus disallows a direct application of Theorem 5. Instead, we apply once again an idea developed in Section 4.1: prepare $\log_2 \lambda$ bags of samples of size B, $2B$, $4B$, ..., λB, respectively, where the l-th bag ($1 \leq l \leq \lceil \log_2 \lambda \rceil$) of samples is taken from $\mathcal{C}_j \cup ... \cup \mathcal{C}_{j+2^l-1}$. A one-sided query can be answered in $O(\log_B \log_2 \lambda + (t/B)\log_{M/B}(n/B))$ amortized expected I/Os from an appropriate bag. When a bag is exhausted, replenish it by querying the set sampling structures of the relevant chunks and a random permutation.

- Let $u[1], ..., u[f]$ be the child nodes of u. The third secondary structure allows us to answer a two-sided query whose query interval is the union of several continuous intervals in $\{I(u[1]), ..., I(u[f])\}$. For this purpose, we directly use the same structure $\mathcal{T}(u)$ as described in Section 4.2.

T has $O(g/\log^{(i)} g)$ nodes at level $h - i$ ($0 \leq i \leq h$), each with at most $f = O(\frac{\log^{(i)} g}{\log^{(i+1)} g})$ child nodes, and thus requiring $O(f \log f)$ blocks for its secondary structures. Hence, all these nodes occupy $O(g)$ blocks in total. Overall, T requires $O(g \log^\star g)$ space.

Query. The query algorithm is completely the same as the one in Section 4.2 (remember that each node u in T has been given secondary structures with the same functionality as those in the previous subsection). We thus conclude the proof of Lemma 6.

4.4 A 2-Sided Structure of Linear Space

We are now ready to describe our $O(n/B)$-space structure for answering two-sided IRS queries. Set $s = B\log^\star(n/B)$. We divide $\mathbb{R}$ into a set $\mathcal{I}$ of $g = \Theta(n/s)$ intervals $\mathcal{I}_1, ..., \mathcal{I}_g$. Each interval covers exactly s points of the input set P, except possibly the last one which can contain between s and $2s$ points of P. In any case, define $\mathcal{C}_i = \mathcal{I}_i \cap P$ for each $i \in [1, g]$. Build a structure of Lemma 6 on the two-level IRS problem defined by $\mathcal{I}$ and $\mathcal{C}_1, ..., \mathcal{C}_g$. This structure occupies $O(n/B + g \log^\star g) = O(n/B)$ blocks.

Now we explain how to process a query with parameter $q = [x, y]$ and t. If q is covered by some interval $\mathcal{I}_i$, we answer it within chunk $\mathcal{C}_i$. First, scan $\mathcal{C}_i$ to obtain $S = \mathcal{C}_i \cap q$, and store S in an array; this requires $O(s/B)$ I/Os. Second, generate t random ranks in the range $[1, |S|]$, and sort them in ascending order. Third, obtain t samples by merging the rank list and S in $O(s/B)$ I/Os. Finally, compute a random permutation of the t samples. Therefore, the total query cost is $O(\log^\star(n/B) + \log_B n + (t/B)\log_{M/B}(t/B))$, where the second term is for finding $\mathcal{I}_i$.

If q intersects at least two intervals in $\mathcal{I}$, we break the query into two one-sided queries (each of which will be answered within a chunk) and one two-sided query with a range q^* that equals the union of several intervals in $\mathcal{I}$. We process the two one-sided queries by the "within-chunk" algorithm we have just described, and the query with range q^* by the two-level IRS structure, after deciding the sample sizes for these queries appropriately. Finally, we perform a random permutation of all the samples fetched. The query cost is $O(\log^\star(n/B) + \log_B n + (t/B)\log_{M/B}(n/B))$ amortized expected. We have thus established the last main result of this paper:

THEOREM 6. *For the IRS problem, there is a structure of $O(n/B)$ space that answers a two-sided WR or WoR query in $O(\log^\star(n/B) + \log_B n + (t/B)\log_{M/B}(n/B))$ amortized I/Os in expectation.*

ACKNOWLEDGEMENTS

This work was supported in part by projects GRF 4165/11, 4164/12, and 4168/13 from HKRGC.

5. REFERENCES

[1] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 487–498, 2000.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

[3] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal of Computing*, 32(6):1488–1508, 2003.

[4] V. Braverman, R. Ostrovsky, and C. Zaniolo. Optimal sampling from sliding windows. *Journal of Computer and System Sciences (JCSS)*, 78(1):260–272, 2012.

[5] P. Efraimidis and P. G. Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters (IPL)*, 97(5):181–185, 2006.

[6] R. Gemulla, W. Lehner, and P. J. Haas. Maintaining bernoulli samples over evolving multisets. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 93–102, 2007.

[7] J. Gustedt. Efficient sampling of random permutations. *Journal of Discrete Algorithms*, 6(1):125–139, 2008.

[8] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 171–182, 1997.

[9] J. M. Hellerstein, E. Koutsoupias, D. P. Miranker, C. H. Papadimitriou, and V. Samoladas. On a model of indexability and its bounds for range queries. *Journal of the ACM (JACM)*, 49(1):35–55, 2002.

[10] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. *ACM Transactions on Database Systems (TODS)*, 33(4), 2008.

[11] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *The VLDB Journal*, 19(1):67–90, 2010.

[12] F. Olken. *Random Sampling from Databases*. PhD thesis, University of California at Berkeley, 1993.

[13] M. H. Overmars. *The Design of Dynamic Data Structures*. Springer-Verlag, 1983.

[14] A. Pol, C. M. Jermaine, and S. Arumugam. Maintaining very large random samples using the geometric file. *The VLDB Journal*, 17(5):997–1018, 2008.

[15] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[16] Z. Wei and K. Yi. Beyond simple aggregates: indexing for summary queries. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 117–128, 2011.

Appendix: Proof of Lemma 3

Denote by R the set of samples we obtain after eliminating duplicates. Consider any $t < s$, and an arbitrary subset S' of S with $|S'| = t$. Thus, $\mathbf{Pr}[R \subseteq S'] = (t/k)^{2s}$. Hence, the probability that $R = S'$ is at most $(t/k)^{2s}$. Therefore:

$$\begin{aligned}
\mathbf{Pr}[|R| < s] &= \sum_{t=1}^{s-1} \mathbf{Pr}[|R| = t] \\
&\leq \sum_{t=1}^{s-1} \binom{k}{t} (t/k)^{2s} \\
&\leq \sum_{t=1}^{s-1} (ek/t)^t \cdot (t/k)^{2s} \\
(\text{by } e^t < e^s < e^{2s-t}) \quad &< \sum_{t=1}^{s-1} (et/k)^{2s-t} \\
&< \sum_{t=1}^{s-1} (es/k))^{2s-t} \\
&\leq \frac{es/k}{1 - es/k} \leq \frac{1/3}{2/3} = 1/2.
\end{aligned}$$

The lemma thus follows.