Independent Range Sampling on a RAM

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 invited paper summarizes our results on the "independent range sampling" problem in the RAM computation model. 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 no updates are allowed, there is a trivial $O(n)$-size structure that guarantees $O(\log n + t)$ query time. The problem is much more interesting when P is dynamic (allowing both 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.

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 those elements can be extremely time consuming. 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:

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

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 Model. We study IRS 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.

Existing Results. Next we review the literature on IRS, assuming the WR semantics—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 $q, 2t$.

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.

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, 3, 4, 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. 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*. We will describe a dynamic structure of $O(n)$ space that answers a two-sided 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). For one-sided queries, the query time can be improved to $O(\log \log n + t)$ expected, while retaining the same space and update complexities. Besides their excellent theoretical guarantees, all of our structures have the additional advantage of being fairly easy to implement.

Assuming the WR semantics, we will first describe a structure for one-sided queries (Section 2), before attending to two-sided ones (Sections 3 and 4). In Section 5, we will explain how to answer WoR queries.

2 A One-Sided Structure

Structure. We build a *weight-balanced B-tree* (WBB-tree) [2] 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 [2]), 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.

3 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.

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:

- $\boldsymbol{c1}$ $I(u_1)$ and $I(u_2)$ are disjoint, and their union covers q;
- $\boldsymbol{c2}$ $|P(u_1)| + |P(u_2)| = O(k)$.

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 2: The u_1 and u_2 we decided satisfy conditions $\boldsymbol{c1}$ and $\boldsymbol{c2}$.

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

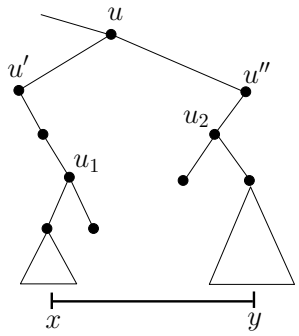

Figure 1: Answering a query at two nodes

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 for updating our one-sided structure. The overall update time is $O(\log n)$.

Lemma 3: 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.

4 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 3 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 4: 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.

5 Reduction from WoR to WR

We will need the fact below:

Lemma 5: 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 $1/2$.

Proof: 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

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

most $(t/k)^{2s}$. Therefore:

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

The lemma thus follows. □

A two-sided WoR query with parameters q, t on dataset P can be answered using a structure of Theorem 4 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 5, 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.

6 Concluding Remarks and Future Work

In this paper, we have described an IRS structure that consumes $O(n)$ space, answers a WoR/WR query in $O(\log n + t)$ expected time, and supports an insertion/deletion in $O(\log n)$ time. The query time can be improved to $O(\log \log n + t)$ if the query range is one-sided.

For two-sided queries, we can make the query time $O(\log n + t)$ hold deterministically (currently, it is expected). For that purpose, we need sophisticated big-twiddling structures (specifically, the *fusion tree* [5]) that allow one to beat comparison-based lower bounds on searching an ordered set. We do not elaborate further on this because the resulting structures are perhaps too complex to be interesting in practice. How to make the one-sided query time $O(\log \log n + t)$ hold deterministically is still an open question. IRS has also been studied in external memory (i.e., in the scenario where P does fit in memory). Interested readers may refer to [9] for details.

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.

Acknowledgments

This work was supported in part by projects GRF 4164/12, 4168/13, and 142072/14 from HKRGC.

References

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

[2] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM Journal of Computing*, 32(6):1488–1508, 2003.

[3] Vladimir Braverman, Rafail Ostrovsky, and Carlo Zaniolo. Optimal sampling from sliding windows. *Journal of Computer and System Sciences (JCSS)*, 78(1):260–272, 2012.

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

[5] Michael L. Fredman and Dan E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 1–7, 1990.

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

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

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

[9] Xiaocheng Hu, Miao Qiao, and Yufei Tao. Independent range sampling. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 246–255, 2014.

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

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

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

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

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

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

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