Cache-Oblivious Iterated Predecessor Queries via Range Coalescing

Erik D. Demaine, Vineet Gopal, and William Hasenplaugh

Massachusetts Institute of Technology
Cambridge, MA 02139, USA
{edemaine, vineetg, whasenpl}@mit.edu
http://toc.csail.mit.edu/

Abstract. In this paper we develop an optimal cache-oblivious data structure that solves the iterated predecessor problem. Given k static sorted length-n lists $L_1, L_2, \ldots, L_k$ and a query value q, the ***iterated predecessor*** problem is to find the largest element in each list which is less than q. Our solution to this problem, called "range coalescing", requires $O(\log_{B+1} n + k/B)$ memory transfers for a query on a cache of block size B, which is information-theoretically optimal. The range-coalescing data structure consumes $O(kn)$ space, and preprocessing requires only $O(kn/B)$ memory transfers with high probability, given a tall cache of size $M = \Omega(B^2)$.

1 Introduction

The ***predecessor*** problem is to find the largest item in a given sorted list L that is less than a query value $q \in \mathbb{R}$. The ***iterated predecessor*** problem is to find the predecessor for a query q in each of a set of k static lists $L_1, L_2, \ldots, L_k$, each of size n. A naive solution involves individual binary searches over all k lists, which would require $O(k \lg n)$ time in the worst-case. However, Chazelle and Guibas [6] showed that the lists can be preprocessed to support iterated predecessor queries in $O(\lg n + k)$ time, with linear preprocessing time and linear space via their technique ***fractional cascading***.

In this project, we will demonstrate that the iterated predecessor problem can also be solved using a technique called ***range coalescing*** in $O(\lg n + k)$ time. Range coalescing is ***cache-oblivious*** [7], using only $O(\log_{B+1} n + k/B)$ memory transfers per query in the worst case.[1] Furthermore, range coalescing requires only linear space and the preprocessing requires $O(kn)$ time and $O(kn/B)$ memory transfers.

The essence of range coalescing, as the name suggests, is to coalesce ranges of the query space into n "bins", each of which could generate $O(k)$ different results depending on the specific value of q within that range. Figure 1 illustrates how the smallest element in each bin is the "splitter" for the bin, so named because

[1] Throughout this paper we will use the notation $\lg n$ to mean $\log_2 n$ and $\ln n$ to mean the natural logarithm.

they collectively "split" all of the elements into bins of contiguous value ranges. In addition, the predecessor of the splitter from each list is included in order to service predecessor requests that are smaller in value than the smallest element in the bin from each list. Figure 2 gives an example of how the data in a bin is stored. The elements from each list are stored in sorted subsequences of varying lengths, but of total length $O(k)$. Each bin stores $O(k)$ different values and thus the total data structure is linear space.

A query q on a bin D walks through the $O(k)$ elements in D in a single pass. Within D are k subsequences of each list in sorted order as depicted in Figure 2, so we merely take the largest element from the ith subsequence which is less than q as our predecessor answer from the ith list. A more detailed description of the query process can be found in Section 4.

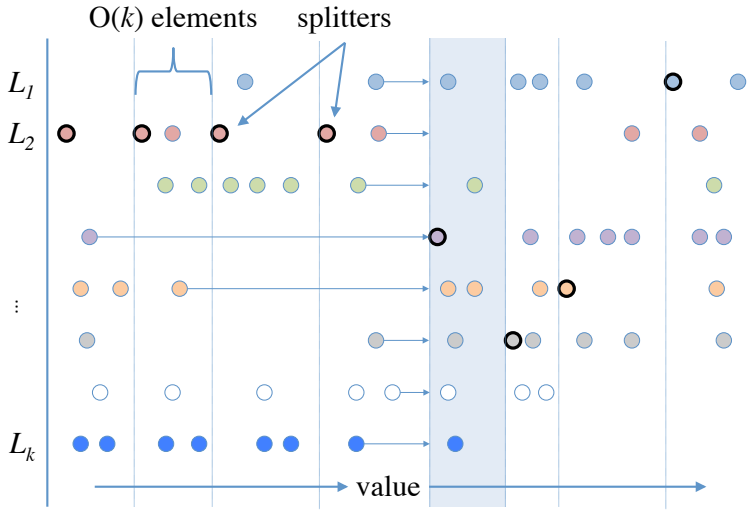

Fig. 1: Range coalescing data structure for the iterated predecessor problem with the value of the element on the x-axis and each row representing a list in $\{L_1, L_2, \ldots, L_k\}$. Each vertical line delineates the $O(k)$ elements in each bin. The elements with a heavy black border are the splitters for each bin — the smallest item in the bin is the splitter. An example bin is highlighted by the blue vertical bar. The elements to the left of the bar with rightward arrows are the predecessors of the splitter from the blue bin in each list.

Throughout this paper we will let M be the size of the cache and B be the size of the cache blocks on a hypothetical external-memory machine. We will consider solutions to the iterated predecessor problem, in which we are given k n-length lists $L_1, L_2, \ldots, L_k$ and we preprocess them to improve the query time.

In Section 2 we discuss some simple known results from the study of cache-oblivious algorithms and data structures which are useful in subsequent analysis. Section 3 gives an overview of solutions to the iterated predecessor problem which are not cache-oblivious, but nonetheless serve as a reasonable baseline for

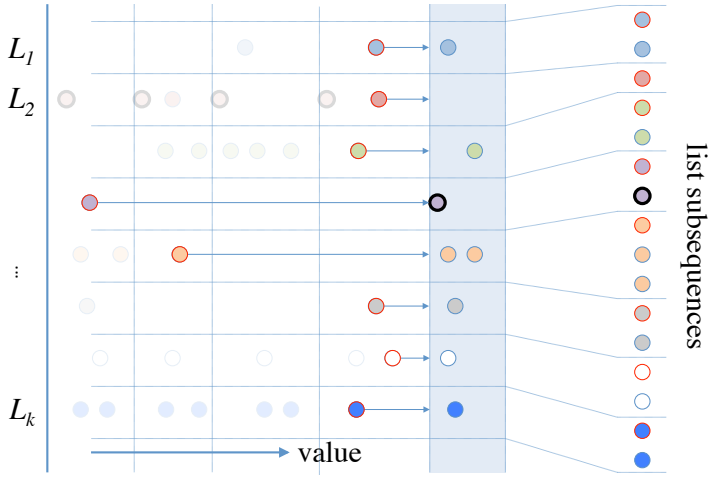

Fig. 2: An example of how a bin is constructed in a range coalescing data structure. The elements that fall in the range of the blue bar and the predecessor of the splitter (with heavy black border) from each list is represented in the bin. In particular, the subsequences of those elements from each list are stored together and concatenated so that the subsequence from L_1 comes first and so on until the subsequence from L_k.

our work. We present range coalescing in Section 4 and show that they answer queries cache-obliviously using $O(\log_{B+1} n + k/B)$ memory transfers. Section 5 demonstrates that the preprocessing for a range coalescing data structure requires only $O(kn/B)$ with high probability. Section 6 describes our implementations of each solution described herein and an experimental methodology for testing the performance of each. Finally, Section 7 describes some limitations of range coalescing, providing opportunities for future work.

2 Cache-oblivious Tools

This section describes some cache-oblivious primitive operations that are known in the literature and useful to build up solutions to the iterated predecessor problem in this paper.

2.1 Array Scanning

Accessing a random element in an length n array requires $O(1)$ memory transfers. However, if we access the entire array in order, we can achieve $O(n/B)$ memory transfers, where B is the size of a block in cache. Each memory transfer brings B elements into cache, so we must make at most $O(n/B)$ transfers.

2.2 vEB Layout

Traditional binary search on an array requires $O(\lg(n/B))$ memory transfers. Every access to the array is a random access, and could be located in a different cache block, except for the last $O(\lg B)$ elements which are located on the same block.

The vEB layout as depicted in Figure 3 tries to optimize memory transfers by rearranging the array. It works by recursively dividing the tree into triangles, and storing each triangle contiguously in memory. This means that children and parent nodes are likely to be stored together in memory, reducing the number of memory transfers required.

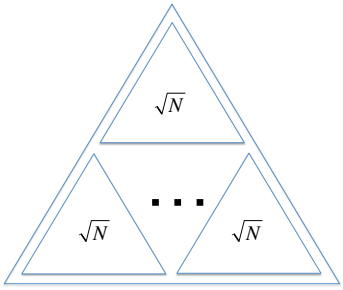

Fig. 3: vEB trees are recursively divided into triangles of size $\sqrt{N}$. Each of these triangles is stores contiguously in memory.

Lemma 1. *A query on an binary tree in the vEB layout requires* $O(\log_{B+1} n)$ *memory transfers.*

Proof. Triangles of size S are recursively divided into smaller triangles of size $\sqrt{S}$. Lets examine the largest triangle that has at most B elements. This triangle must have at least $\sqrt{B+1}$ elements, so its height must be at least $\frac{1}{2}\lg(B+1)$. This entire triangle can be loaded into one cache block. There are at most $\lg n / \frac{1}{2}\lg(B+1) = 2\log_{B+1} n$ of these triangles along a root to leaf path, so we only need to make $O(\log_{B+1} n)$ memory transfers to find an element. □

The vEB layout has been the basis for several known cache-oblivious algorithms, including B-trees [2], funnel sort [7], and priority queues [1, 5]. It is also used in our solution for range coalescing.

3 Known Solutions

We examine several simple solutions to the iterated predecessor problem. These solutions provide a good background for understanding range coalescing and serve as our implementation baselines in Section 6.

3.1 Sequential Binary Search

The simplest solution to the iterative predecessor problem is to do a binary search on each of the k unmodified lists $L_1, L_2, \ldots, L_k$, and write down the output from each. Since each binary search requires $O(\lg(n/B))$ memory transfers, this solution requires a total of $O(k \lg(n/B))$ memory transfers. The total space usage is optimal, $O(kn)$.

3.2 Sequential vEB Binary Search

Using the vEB layout described previously, we can do binary searches using only $O(\log_{B+1} n)$. If we use a vEB layout for each of the k lists, we can perform the searches in $O(k \log_{B+1} n)$. The total space usage is optimal, $O(kn)$.

3.3 Fractional Cascading

Fractional cascading [6] is the incumbent solution for the in-memory iterated predecessor problem. An external memory-oriented extension of fractional cascading described below achieves a runtime of $O(\log_{B+1} n + k)$.

The main idea behind fractional cascading is to use the query result from each list to perform a search on the next list in constant time. One needs only to do a binary search on the first list to prime the pipeline. To do this, we store pointers in each list to its predecessor and successor in the next list. This gives us a constant-sized range to search through in the next list.

If we did this naively, this would be of no benefit — the predecessor and successor of the ith list could span the entirety of the $i + 1$st list. However, by altering the lists slightly, we can achieve constant time per remaining search. Starting with the last list, we insert every other element into the previous list. We do this for each list. This ensures that the range between predecessor and successor is at most a constant value.

We store the initial list in the vEB layout to minimize memory transfers for the initial binary searches. Using this method, we can perform a query using $O(\log_{B+1} n + k)$ memory transfers. The total space usage is optimal, $O(kn)$.

3.4 Quadratic Storage

A brute-force fast solution involves storing one sorted kn-length list of all elements from all lists using the vEB layout. Accompanying each element in the list is a k-length sublist with a copy of its k predecessors, one from each list in $\{L_1, L_2, \ldots, L_k\}$. We can iterate over this contiguous sublist using $O(k/B)$ memory transfers, so the total number of memory transfers is $O(\log_{B+1}(kn) + k/B)$. However, the total space usage is $O(nk^2)$, since we must store a k-length sublist for each element.

4 Range Coalescing

In this section we describe how an iterated predecessor query can be satisfied cache-obliviously using $O(\log_{B+1} n + k/B)$ memory transfers using a ***range coalescing*** data structure. We describe how a range coalescing data structure is built cache-obliviously from a set of sorted lists $L_1, L_2, \ldots, L_k$ each of size n using only $O(kn/B)$ memory transfers with high probability in Section 5.

Let H be a range coalescing data structure built from a set of n-length sorted lists $L_1, L_2, \ldots, L_k$. H is composed of n bins, each of size $O(k)$, which partition the space of possible query values using a sorted list of n splitters S, as depicted in Figure 1. Figure 2 illustrates how a bin concatenates k sequences of elements, each of which is a subsequence of each constituent list from $\{L_1, L_2, \ldots, L_k\}$. The first element of the ith subsequence in the jth bin is the predecessor of the splitter S_j in L_i and is strictly smaller than S_j by construction, a fact that we will exploit to implicitly denote the beginning of each subsequence.

Lemma 2. *A range coalescing data structure H built from a set of n-length sorted lists $L_1, L_2, \ldots, L_k$ consumes $O(kn)$ space.*

Proof. The elements from all n-length lists $L_1, L_2, \ldots, L_k$ are partitioned into n bins. In addition, each bin has exactly one element for each of the k lists which is smaller in value than the splitter for the bin. Thus, each of the n bins has $O(k)$ elements and the data structure has $O(kn)$ space. □

4.1 Iterated predecessor queries

This section describes the process by which a range coalescing data structure answers iterated predecessor queries and demonstrates that the process incurs $O(\log_{B+1} n + k/B)$ memory transfers with high probability. Figure 4 gives pseudocode for the procedure QUERY, which takes a range coalescing data structure H and a query q and returns an ordered list of results which correspond to the predecessors of q for each constituent list in $\{L_1, L_2, \ldots, L_k\}$.

While it may be that the function QUERY is correct by inspection, we leave nothing to chance and prove it here.

Lemma 3. *Given a range coalescing data structure H and a query value q, the function* QUERY(H,q) *returns the correct answer.*

Proof. Consider the jth bin, with corresponding splitter S_j, which is used to satisfy all queries $q \in [S_j, S_{j+1})$. By construction, the jth bin contains all elements $\{l \in \cup_{i=1}^{k} L_i \text{ s.t. } l \in [S_j, S_{j+1})\}$ in addition to the predecessor of S_j from each list in $\{L_1, L_2, \ldots, L_k\}$. Thus, the jth bin contains the k correct answers — the predecessors of q in each constituent list in $\{L_1, L_2, \ldots, L_k\}$. We also see that each subsequence has exactly one element that is less than the splitter S_j, which allows us to know which subsequence we are in during the course of the scan — each element falling below the splitter denotes the beginning of a new subsequence. Furthermore, since the subsequences are stored in sorted order,

```
QUERY(H, q)
1  ⟨D, s⟩ = vEB(H.S, q)
2  j = 1
3  for i = 1 to D.size − 1
4      if D_i < q
5          Z_j = D_i
6      if D_{i+1} < s
7          j = j + 1
8  return Z
```

Fig. 4: Pseudocode of the QUERY method for a range coalescing data structure H. H contains a sorted array S of splitters organized using a van Emde Boas layout [3]. The function vEB returns a bin D, organized as an array, and a splitter s, which is the predecessor of q in S. The bin D is walked in a linear fashion, overwriting potential predecessors in the output array Z and incrementing the output position whenever the subsequence of the next list begins. The jth subsequence begins with the one and only element from L_j that is smaller than the splitter s and each bin is appended with $-\infty$ to handle the boundary condition when D_{i+1} is compared with the splitter s.

we know that the predecessor result for a particular list L_i corresponds to the largest element less than q in L_i's subsequence. □

Now we bound the number of memory transfers incurred by QUERY by walking through the pseudocode in Figure 4.

Theorem 1. *An iterated predecessor query* QUERY(H, q) *on a range coalescing data structure H built from a set of n-length sorted lists $L_1, L_2, \ldots, L_k$ incurs $O(\log_{B+1} n + k/B)$ memory transfers on a processor with cache blocks of size B.*

Proof. We use the cache-oblivious search tree structure described by Bender, Demaine and Farach-Colton [3] on line 1 of Figure 4 to find the bin corresponding to the predecessor in the sorted n-length splitter list S using $O(\log_{B+1} n)$ memory transfers. After we find the splitter and the corresponding bin D, we merely scan through D once and write out the answers in a continuous stream to the array *output*. Thus, we incur a read stream and a write stream, each of which is $O(k)$ elements and $O(k/B)$ memory transfers. □

5 Preprocessing

This section describes how a range coalescing data structure is built cache-obliviously from a set of k n-length sorted lists $L_1, L_2, \ldots, L_k$ and bounds the number of memory transfers incurred by the process. We do this in four steps. First, we give a suboptimal deterministic strategy for finding the "splitters" — the values that partition the query space such that each partition has $O(k)$ elements from the constituent lists in $\{L_1, L_2, \ldots, L_k\}$. Second, we demonstrate that the elements from each list can be assembled in the bin corresponding to

each splitter using $O(kn/B)$ memory transfers. Finally, we give two randomized algorithm for finding the splitters when $k < \ln^2 n$ and $k \geq \ln^2 n$, respectively, each of which incurs $O(kn/B)$ memory transfers.

5.1 Preprocessing suboptimally

In this section, we show how to find an n-length sorted splitter array S, such that $O(k)$ elements from $\mathcal{L} = \cup_{i=1}^{k} L_i$ fall in each range $[S_j, S_{j+1}) \; \forall \, 1 \leq j \leq n$. If we assume that all elements in $\mathcal{L}$ are unique, we can merely merge all the elements and take every kth element in the merged list as the splitters.[2] We can use a cache-oblivious k-merger [7] to merge the elements using $O((kn/B)\log_{M/B}(k/B) + k)$ memory transfers if $k \leq \sqrt[3]{n}$ and $O((kn/B)\log_{M/B}(kn/B))$ memory transfers otherwise.

5.2 Bin construction

This section demonstrates how we can build the $O(k)$-sized bin corresponding to each splitter in the array S using $O(kn/B)$ memory transfers in the worst case. If we were to merely build each of the n bins in sequence, each of which could incur as many as $2k$ memory transfers since k may be larger than M, we could incur as many as $O(kn)$ memory transfers overall. This is unacceptable. Instead, we will build the bins using a Z-order traversal [8] of the 2D space spanned by the cross-product of bin number and list number, notated as the bin number $\times$ list number iteration space.

Theorem 2. *Given a sorted list of $O(n)$ splitters S and k sorted n-length lists $L_1, L_2, \ldots, L_k$, the bins for a range coalescing data structure can be constructed deterministically and cache-obliviously using $O(kn/B)$ memory transfers on a processor with a cache of size $M = \Omega(B^2)$ and cache blocks of size B.*

Proof. Consider a 2^r by 2^r naturally aligned region in the bin number $\times$ list number iteration space, like the one depicted in Figure 5.[3] The cache need only keep the head of each of the 2^r constituent lists $L_{i2^r+1}, L_{i2^r+2}, \ldots, L_{(i+1)2^r}$ for some $i > 0$ and the head of the 2^r bins $D_{j2^r+1}, D_{j2^r+2}, \ldots, D_{(j+1)2^r}$ for some $j > 0$. The "head" of a list is the current location in the list as the list is streamed linearly to transfer the elements to various bins. For the head of each list, there can be as many as two non-full memory transfers (i.e., not all of the elements on the cache block were written or read). Consider the largest r for which these $2 \cdot 2^r$ list heads fit in cache, so that $a2^r B = M$ for some constant a. In total,

[2] We can extend the value of each element with the list number in order to make them unique, since the elements from any particular list L_i are unique. Note that if each list contained a value l and the value l from the L_i was chosen as the jth splitter S_j, we do not compromise the correctness of the query, since the next smaller value than S_j from each list in $\{L_1, L_2, \ldots, L_k\}$ is contained in the jth bin.

[3] A ***naturally aligned*** region of size c by c is one which begins with some index $i \equiv 1 \pmod{c}$ in one dimension and $j \equiv 1 \pmod{c}$ in the other.

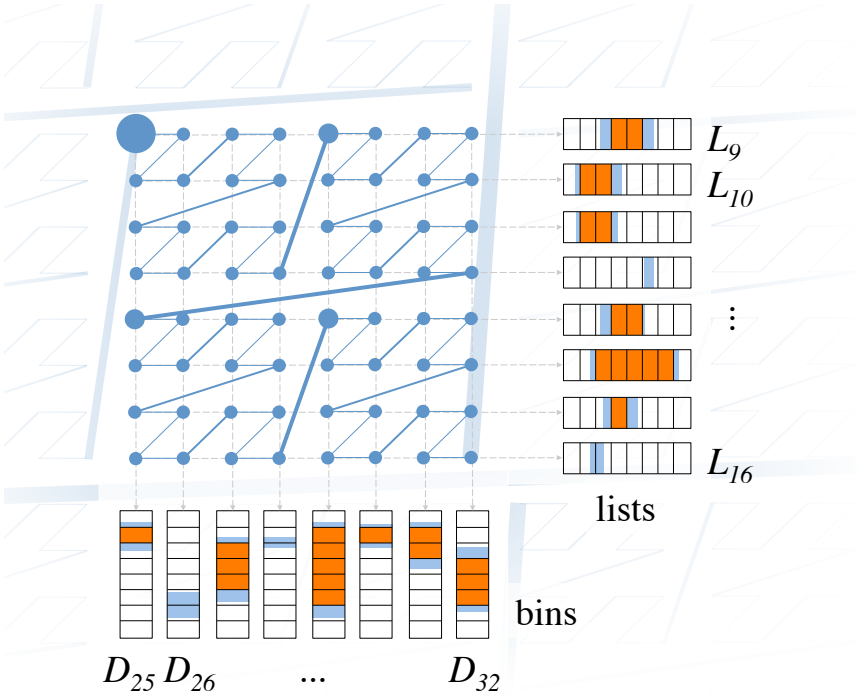

Fig. 5: Example 2^r by 2^r (for $r = 3$) region of a Z-order traversal of the bin number $\times$ list number iteration space. During the course of the execution of this example region there are 8 lists and 8 bins active. The blue regions represent cache blocks which are partially read (lists $L_9, L_{10}, \ldots, L_{16}$) and partially written (bins $D_{25}, D_{26}, \ldots, D_{32}$). The orange blocks are those which are fully read or written, respectively.

there will be $kn/2^{2r}$ cache flushes for a total of $(M/B) \cdot kn(aB/M)^2 = O(kn/B)$ cache blocks, assuming $M = \Omega(B^2)$. There may also be additional full memory transfers in the course of processing each 2^r by 2^r region, though each element may appear in at most one full memory transfer, thus there are at most kn/B full memory transfers. □

5.3 Finding splitters for small k

In this section, we show how to find the splitters which partition all of the elements into bins, each of which stores the answers to any query which falls in the range of values between the associated splitter and the splitter for the next bin of larger value. When k is less than $\ln^2 n$, we can randomly select elements to be splitters with probability $1/k$ using $O(kn/B)$ memory transfers by streaming the lists and writing out the samples to separate lists. Next, we subdivide bins that are too large, potentially generating extra splitters. We will show that this

process generates $O(kn/B)$ memory transfers with high probability.[4] First, we establish two straightforward yet useful lemmas.

Lemma 4. *Consider a coin with heads probability $1/k$. In kn flips we will see between $n/2$ and $2n$ heads with probability at least $1-(kn)^{-c}$ for some $c>1$.*

Proof. The proof follows from an application of Hoeffding's inequality, for sufficiently large n and the assumption that $k < \ln^2 n$. □

Lemma 5. *The largest number of elements with value between two splitters selected randomly with probability $1/k$ is $(1+\varepsilon)k\ln(kn)$ with probability at least $1-(kn)^{-\varepsilon}$ for any $\varepsilon > 0$.*

Proof. We can think of a bin as being created by successive coin flips with probability of heads equal to $1/k$: every time tails comes up, the bin grows by one. Thus, the probability that a bin is of a particular size R is at most $(1-1/k)^R/k$. Summing from $R = R'$ to ∞, we can bound the probability that a particular bin has size at least R', $\sum_{R=R'}^{\infty}(1-1/k)^R/k = (1-1/k)^{R'}$. Letting $R' = (1+\varepsilon)k\ln(kn)$ and taking the union bound across at most kn different bins, the proof follows. □

Consider the process of splitting a bin which exceeds $2(1+\varepsilon)k$ elements. We use $\lfloor R/2(1+\varepsilon)k \rfloor$ applications of a cache-oblivious selection algorithm [7] to subdivide large bins of size R into bins of size at most $2(1+\varepsilon)k$ elements using $O(\lfloor R/2(1+\varepsilon)k \rfloor R/B)$ memory transfers.[5]

Theorem 3. *The total number of memory transfers S required to subdivide all m bins to be less than $2(1+\varepsilon)k$ elements each is $O(kn/B)$, assuming that the largest bin is at most $(1+\varepsilon)k\ln(kn)$ elements and $n/2 \le m \le 2n$.*

Proof. Let x_i be the number of memory transfers incurred by the ith bin and thus $S = \sum_{i=1}^{m} X_i$. Then, we have

$$\begin{aligned} E[x_i] &\le a\sum_{R=0}^{\infty}\frac{1}{k}\left(1-\frac{1}{k}\right)^R\left\lfloor\frac{R}{2(1+\varepsilon)k}\right\rfloor\frac{R}{B} \\ &\le a\sum_{R=0}^{\infty}\left(1-\frac{1}{k}\right)^R\frac{R^2}{2(1+\varepsilon)k^2B} \\ &\le a\frac{k}{B} \end{aligned}$$

for some constant $a > 0$. Also, we see that

$$\begin{aligned} x_i &\le \left\lfloor\frac{(1+\varepsilon)k\ln(kn)}{2(1+\varepsilon)k}\right\rfloor\frac{(1+\varepsilon)k\ln(kn)}{B} \\ &\le \frac{k}{2B}(1+\varepsilon)\ln^2(kn) \end{aligned}$$

[4] In this context, ***with high probability*** means with probability greater than $1-N^{-c}$ where N is the total number of elements in the problem and $c > 1$ is some constant.

[5] For convenience, we assume $R > B$. There is no need to make the bins smaller than B elements, since processing a bin incurs at least one memory reference.

for all i since the largest bin is assumed to have at most $(1+\varepsilon)k\ln(kn)$ elements. Let $t = (k/2B)(1+\varepsilon)\ln^2(kn)$ and note that each random variable in $\{x_1, x_2, \ldots, x_m\}$ has support in the range $[0, t]$. A Hoeffding bound on S gives us $\Pr\Big\{S - E[S] \geq t\sqrt{\varepsilon m \ln(kn)}\Big\}$

$$\begin{aligned}
&\leq \exp\left(-2\frac{\varepsilon m \ln(kn) t^2}{mt^2}\right) \\
&\leq \exp(-2\varepsilon \ln(kn)) \\
&\leq (kn)^{-\varepsilon}.
\end{aligned}$$

Then, for sufficiently large kn, $S = O(kn/B)$ with high probability. □

We need to verify that the process of subdivision does not unduly increase the number of bins.

Lemma 6. *After subdivision, we will have $O(n)$ bins.*

Proof. Initially, there are $O(n)$ splitters with high probability by Lemma 4. The process of subdividing bins generates bins with size at least k, thus we can create at most n extra bins through subdivision. □

5.4 Finding splitters for large k

When k is at least $\ln^2 n$, we use an oversampling technique as used in sample sort [4] to find a set of splitters and bound the size of all bins. In particular, we start by randomly sampling elements as candidate splitters with probability $1/\ln k$, which can be accomplished by streaming each list and writing out the samples to a separate candidate list with $O(kn/B)$ memory transfers. Then, we merge these candidates using and a cache-oblivious k-merger [7] using $O((kn/B \ln k)\log_{M/B}(k/B) + k) = O(kn/B)$ memory transfers, assuming $M = \Omega\big(B^2\big)$ and $n = \Omega(B)$. Finally, we take every n evenly space elements from this sorted list as our set of splitters.

Theorem 4. *Given an oversampling rate of $k/\ln k$, the largest resulting bin has at least $2(1+\varepsilon)k$ elements with probability less than $(kn)^{-\varepsilon}$.*

Proof. Let R be the size of the largest bin. By Theorem B.3 of [4], for sufficiently large kn, we have that

$$\begin{aligned}
\Pr\{R > 2(1+\varepsilon)k\} &\le kn \exp\left(-(1+\varepsilon)\left(\frac{1+2\varepsilon}{2(1+\varepsilon)}\right)^2 \frac{k}{\ln k}\right) \\
&\le kn \exp\left(-(1+\varepsilon)\frac{k}{4\ln k}\right) \\
&\le kn \exp\left(-(1+\varepsilon)\frac{\ln^2 \frac{kn}{\ln^2 n}}{4\ln\ln^2 n}\right) \\
&\le kn \exp\left(-(1+\varepsilon)\frac{\ln^2(kn) - o\big(\ln^2(kn)\big)}{8\ln\ln n}\right) \\
&\le kn \exp(-(1+\varepsilon)\ln(kn)) \\
&\le (kn)^{-\varepsilon}.
\end{aligned}$$

□

6 Implementation and Experimentation

We implemented each of the simple solutions described in Section 3, and compared their performance to range coalescing on a variety of data sets.

Each solution was implemented in C++, and compiled and tested on an Intel i7 processor with 3MB of L3 cache. We implemented the full merge range coalescing solution described in Section 5, instead of the randomized solution. Before each test, k lists each of length n were generated using a uniform distribution of integers from 0 to 1 million. These lists were passed in as input to each of the solutions. The initialization times and average query times of each solution were recorded.

Range coalescing performed significantly better in practice than other linear-storage solutions. It performed better on both small and large datasets. Average query times are shown in Figure 6 and Figure 7. As the datasets got larger, the effects of range coalescing became more evident. For $n = 50$ and $k = 1000$, range coalescing did 5 times better than a simple binary search, whereas for $n = 5000$ and $k = 1000$, range coalescing performed 18 times better.

However, range coalescing requires much more time for preprocessing. It takes about 20 times longer to initialize than the vEB search, and 3 times longer than fractional cascading. We believe these results could be improved upon - we did not implement the linear time randomized preprocessing method described in Section 5, leaving it instead to future work.

The average query time for the quadratic storage solution is better than all linear storage solutions. However, it requires $O(nk^2)$ time to initialize. For $k = 1000, n = 100$, this is 42 times longer than the preprocessing for the range coalescing solution.

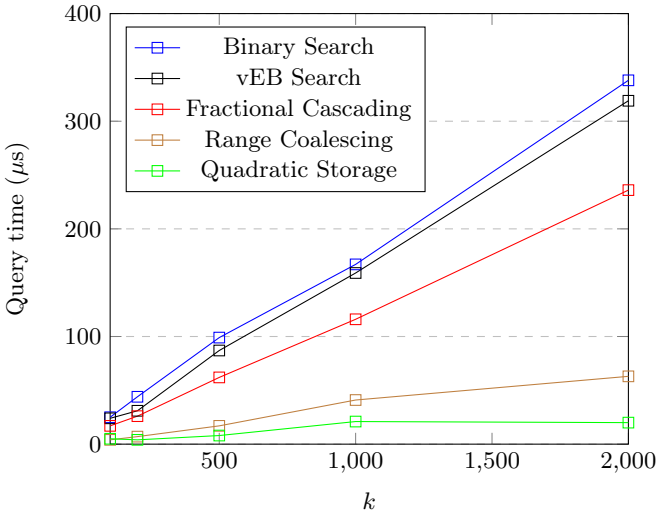

Fig. 6: Query times vs k (for fixed n=1000).

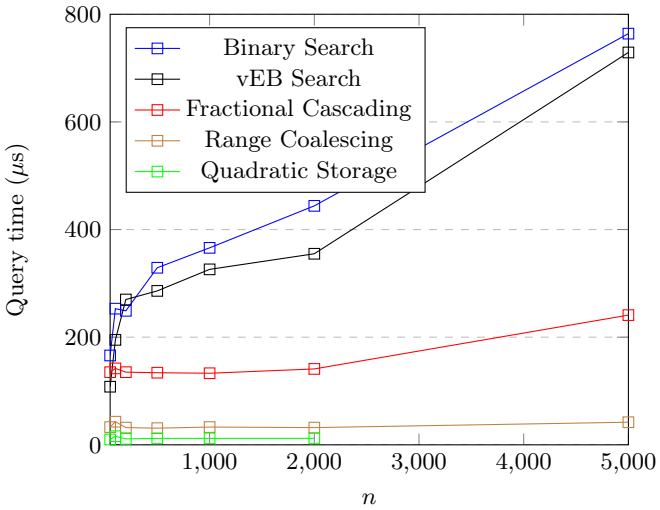

Fig. 7: Query times vs n (for fixed k=1000).

7 Future Work

We have presented an optimal cache oblivious solution for the static iterated predecessor query. There are several areas in which this can be extended. Range coalescing does not currently support dynamic operations like insert or delete. For instance, it is not obvious how one would avoid the adversarial behavior of repeatedly adding and deleting an element. An element can appear in many bins if the corresponding list does not have other elements in those bins. Repeatedly

adding and deleting such an element could cost $\Omega(n)$ work per operation given a naive extension to dynamic range coalescing.

Range coalescing specifically solves the iterated predecessor problem on k lists, but this does not generalize easily to a graph of lists. Fractional cascading achieves a running time of $O(\lg n + k)$ on a graph query, where k is the length of the traversed path in the graph. Applying range coalescing directly to this problem results in $O(\log_{B+1} n + K)$, where K is the total size of the graph. The concepts of range coalescing could hopefully be developed to be used as a black box for such problems.

Acknowledgments

This work was begun during the open-problem sessions of the MIT class 6.851: Advanced Data Structures taught by E. Demaine in Spring 2014. We thank the other participants for making a creative and productive environment.

References

1. Arge, L., Bender, M. A., Demaine, E. D., Holland-Minkley, B., and Munro, J. I. Cache-oblivious priority queue and graph algorithm applications. In *STOC* (2002).
2. Bayer, R., and McCreight, E. M. Organization and maintenance of large ordered indexes. *Acta Informatica* (1972).
3. Bender, M. A., Demaine, E., and Farach-Colton, M. Cache-oblivious B-trees. In *FOCS* (2000), pp. 399–409.
4. Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., and Zagha, M. A comparison of sorting algorithms for the connection machine CM-2. In *SPAA* (1991).
5. Brodal, G. S., and Fagerberg, R. Funnel heap - a cache-oblivious priority queue. In *ISAAC* (2002).
6. Chazelle, B., and Guibas, L. Fractional cascading: 1. a data structuring technique. *Algorithmica* (1986).
7. Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. Cache-oblivious algorithms. In *FOCS* (1999).
8. Morton, G. A computer oriented geodetic data base; and a new technique in file sequencing. Tech. rep., IBM, 1966.