Range Mode Queries in Arrays

Casper Green, 20106348

Jakob Landbo, 20105640

Master's Thesis, Computer Science

January 2016

Advisor: Kasper Green Larsen

Supervisor: Gerth Stølting Brodal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

A mode of an array is the value that occurs at least as frequent as any other value in the array. The range mode problem is, given an array A of size n, construct a data structure that efficiently can answer range mode queries on A. A range mode query consists of a pair of indices (i, j) for which a mode of the sub-array $A[i..j]$ must be returned.

Chan et al.[3] describes four methods to solve the range mode problem. We consider two of the methods and describe algorithms to construct them. We then implement and examine them as well as compare them to their theoretical complexity. Our results show that the theoretical query times fit in practice, and the two methods supplement each other very well, which suggests that a combination of the two could be very efficient as described by Chan et al.[3] in the fourth method.

Furthermore, we look on data structures that approximates the range mode queries. We implement and experiments with a data structure by Bose et al.[2] as well as a data structure by Greve et al.[6]. We describe an algorithm to construct the data structures and give a worst-case construction time bound. Our experiments on the data structures show that the approximate data structures have a faster query time than the exact data structures, but the construction was much slower.

Contents

Chapter 1

Introduction

The range mode problem is given by an input array A and two indices i, j in the array. We want to be able to find the range mode of the values in the subarray $A[i..j]$. The mode of an array is the value that appears at least as frequently as any other value in the array. A range mode is the mode of a subarray rather than the entire array. A range mode query provides the range mode of any subarray of an input array, given two indices in said array.

The mode, along with median and mean, are the most important statistics in data analysis [2, 3]. We focus on data structures that can answer the mode of a range efficiently. This can either be the exact mode or an approximation of the mode within a certain tolerance factor from the frequency of the mode. The problem is basically a trade-off between speed, storage space and for approximating data structures also the guarantee of the data structure. This trade-off was first studied by Kriznac et al.[9] for mode and median range queries, where they proposed a data structure to answer range mode queries in $\mathrm{O}\left(n^{\epsilon}\log n\right)$ time using $\mathrm{O}\left(n^{2-2\epsilon}\right)$ space, where $0 < \epsilon \leq \frac{1}{2}$ is the query time-space tradeoff. They also describe a constant query time using $\mathrm{O}\left(n^2 \log\log n / \log n\right)$ space. This is later studied and improved on by many, which is further described in Chapter 2.

The range mode query problem is simple to solve naively in linear time, but as the data increases a linear time solution might be too slow for practical usage cases depending on the expected query range size. Likewise we can naively compute the mode of every possible range, in which case we are able to respond to queries in constant query time, but this requires $\mathrm{O}\left(n^2\right)$ space as we need to store a mode for all sub-arrays in the array.

In this thesis we consider static data structures that work on stored non-changing data sets. We explain the theory behind some data structures, and describe how we implement them in practice. We further expand on the theory of some of the data structure to include a worst-case bound on the construction time. Some of the data structures that we considered did not mention construction time.

We contribute with a construction time bound on the 3-approximate data structures described by Greve et al.[6] We describe a $\mathrm{O}\left(n^{3/2}\right)$ time construction algorithm to initialize the data structure. We also give a worst case construction time on first and third method in the paper by Chan et al.[3]. The construction time wasn't the main focus in the articles, so it was an obvious choice to research this area further and see the practical implications of the construction algorithms.

For a range mode query $M_{i,j}$ on input size n with Δ unique elements, we have the following analysis,

Data Structure	Construction	Query Time	Space
Naive	$O(1)$	$O(j-i)$	$O(1)$
A	$O\left(n^{3/2}\right)$	$O(\sqrt{n})$	$O(n)$
B	$O(n)$	$O(\Delta)$	$O(n)$
A + B	$O\left(n^{3/2}\sqrt{w}\right)$	$O\left(\sqrt{\frac{n}{w}}\right)$	$O(n)$
1+ϵ	$O(n^2)$	$O(\log\log_{1+\epsilon} n)$	$O(n\log_{1+\epsilon} n)$
3-Approx	$O\left(n^{3/2}\right)$	$O(1)$	$O(n\log\log n)$

1.1 Structure of thesis

In this thesis we consider five different data structures each of which is described in a chapter of its own. For each of these chapters we describe the theory behind the data structure as well as our implementations and how they vary. We analyse worst-case query and construction time of each data structure.

In the following section we summarize what is to be found in the different chapters in the thesis as well as marking who wrote what chapters. Chapters marked with $^{[C]}$ is mainly written by Casper Green, chapters marked with a $^{[J]}$ is mainly written by Jakob Landbo. Chapter $1^{[C,J]}$ introduces the thesis and the underlying problem that is being researched in the thesis. Chapter $2^{[C]}$ references previous work as well as work done on related problems. In Chapter $3^{[C,J]}$ we outline and explain the essential terminology and concept used in this thesis. Chapter $4^{[J]}$ describes the theory of first data structure in the paper by Chan et al.[3] as well as our practical implementation. The theory and implementation of third data structure from the same paper is described in Chapter $5^{[J]}$. Chapter $7^{[C]}$ describes theory as well as our implementation of the constant query-time 3-approximation data structure from Greve et al. [6] In Chapter $8^{[C,J]}$ we describe a data structure that can answer $(1+\epsilon)$-approximate range mode queries in $O(\log\log_{1+\epsilon} n)$ time using $O(n\log_{1+\epsilon} n)$ space proposed by Bose et al.[2]. In Chapter $9^{[C,J]}$ we describe two basic, intuitive solutions to the range mode problem.

In Chapter $10^{[C,J]}$ we describe how we construct and do the experiments, what data distributions we use, what hardware we tested on, etc. In Chapter $11^{[C]}$ we discuss and choose the ϵ to test our approximate data structure on. The experiments of the data structures giving an exact answer to a query is discussed in Chapter $12^{[J]}$, while the experiments of the data structures approximating the answer to a query is discussed in Chapter $13^{[C]}$. We compare the implementations of the two data structures in Chapter $14^{[C,J]}$.

Our conclusion of the experiments and thesis is to be found in Chapter $15^{[C,J]}$, and future work is discussed briefly in Chapter $16^{[C]}$.

1.2 Implementation

We have chosen the programming language C++ for our implementation of these data structures and the source code can be downloaded at `http://users-cs.au.dk/landbo/thesis/source.zip`

Chapter 2

Related Work

2.1 Range Mode Query

Krizanc et al.[9] considered the Range Mode Query problem on lists in 2005. They proposed a data structure that answers range mode queries in $\mathrm{O}\left(n^{\epsilon}\log n\right)$ and uses $\mathrm{O}\left(n^{2-2\epsilon}\right)$, where $0 < \epsilon \leq \frac{1}{2}$ is a time-space trade-off. In the case of $\epsilon = 1/2$ the data structure have a query time of $\mathrm{O}\left(\sqrt{n}\log n\right)$, which they observe can be improved to $\mathrm{O}\left(\sqrt{n}\log\log n\right)$ by using van Emde Boas tree to do predecessor search. In the same paper they describe a constant time data structure that uses $\mathrm{O}\left(n^2\log\log n/\log n\right)$ space, which improves on the very basic solution of precomputing modes for every range that require $\mathrm{O}\left(n^2\right)$ space.

Petersen and Grabowski[12] improve the second bound by Krizanc et al. the second bound to a constant-query time using $\mathrm{O}\left(n^2\log\log n/\log^2 n\right)$ space. Petersen[11] improves upon the first bound to a $\mathrm{O}\left(n^{\epsilon}\right)$ query time that uses $\mathrm{O}\left(n^{2-2\epsilon}\right)$ space for $0 < \epsilon \leq \frac{1}{2}$.

Chan et al.[3] improves the first bound to support queries in $\mathrm{O}\left(\sqrt{n\log n}\right)$ worse-case time as well as describes a data structure based on the data structure by Krizanc et al.[9] that answers range mode queries in $\mathrm{O}\left(\sqrt{n/w}\right)$ time and uses $\mathrm{O}\left(n\right)$ space, where $w = \Omega(\log n)$ is the word size in the standard RAM computation model.

2.2 Approximate Range Mode Query

Bose et al.[2] considered the approximate range mode query problem and describes a data structure that supports queries in $\mathrm{O}\left(\log\log_{\frac{1}{\alpha}} n\right)$ using $\mathrm{O}\left(\frac{n}{1-\alpha}\right)$ space, which will find a value which frequency is at least α of the frequency of the mode, where $0 < \alpha < 1$ is the approximation parameter. Furthermore, they propose constant query time data structures for $\alpha = 1/2, 1/3$ and $1/4$ using $\mathrm{O}\left(n\log n\right)$, $\mathrm{O}\left(n\log\log n\right)$ and $\mathrm{O}\left(n\right)$ space, respectable.

Greve et. al.[6] describes a data structure that has a constant query time using $\mathrm{O}\left(n\right)$ space for $a = 1/3$. They further describes a data structure that answers queries in $\mathrm{O}\left(\log(\alpha/(1-\alpha))\right)$ time using $\mathrm{O}\left(n\alpha/(1-\alpha)\right)$ space for $\frac{1}{2} \leq \alpha < 1$.

2.3 Related Problems

The data structures referenced only works on static data that doesn't change. A related problem to this that can work on data streams instead of static non-changing data is *frequency elements* over *sliding windows* [1]. Often times data comes in form of a continuous stream of data, financial transaction logs and network logs to name a few. This preference of new recent data is what is referred to as the *sliding window* [4], where queries are answered on the n most recent data points.

Lee and Ting[10] describes an efficient scheme to identify ϵ-approximate frequency elements over a sliding window of size n, i.e. the n most recent elements. The scheme supports a $\mathrm{O}(1/\epsilon)$ update and query space, and uses $\mathrm{O}(1/\epsilon)$ space. This bound is later improved upon by Hung et al.[8] stating the query and space time is essentially optimal, but they improve the $\mathrm{O}(1/\epsilon)$ update time to a constant update time.

Another related problem is the range α-majority query problem, which is, given a query range $[i, j]$, return all elements in $x \in A[i..j]$, where $F_x > \alpha(j - i + 1)$, i.e. the frequency of the element should be greater than $\alpha(j-i+1)$. This problem is considered by Durocher et al. [5]. They describe a data structure that can answer α-majority range queries in $\mathrm{O}(1/\alpha)$ using $\mathrm{O}(n \log(1/(\alpha + 1)))$, where $0 < a < 1$.

Chapter 3

General concepts

3.1 Logarithm

Throughout the thesis we will shorten $\log_2$ as log. If the base of log at some point is different than 2, we will explicitly state it.

3.2 Frequency

Frequency is defined as the highest number of occurrences of any value in an array. For a list A we define $F_{i,j}$ to be the frequency of a sub-list of A starting from the i'th entry to the j'th entry.

3.3 Mode and Range Mode

The mode is defined as the value with the highest frequency in a list A. If multiple values have the same frequency, all of the values with that frequency is a valid mode. Formally, x is a mode of $A = a_1, a_2 ..., a_n$, if

$$\forall y \in A : F_y(a_1, a_2, ..., a_n) \leq F_x(a_1, a_2, ..., a_n)$$

A Range Mode is the mode within a sub-array of A. We use $M_{i,j}$ as the notion of the range mode of A in the interval starting from the entry i and ending at entry j. Formally, x is a range mod of $A[i, j]$ if

$$\forall y \in A : F_y(a_i, a_{i+1}, ..., a_{j-1}, a_j) \leq F_x(a_i, a_{i+1}, ..., a_{j-1}, a_j)$$

3.4 Range Mode Query

Given two indices, i and j, a range mode query answers what value is the mode within the range of i and j on a list A. When building a range mode query data structure, we usually pre-compute values on the list A to be able to efficiently answer range mode queries on the list. We only consider data structures that can answer such queries on static data, i.e. the data structure is not supposed to efficiently update when the input data is changed.

3.5 c-approximate Range Mode Query

A c-approximate range mode query is defined to return a value that occurs at least $1/c$ times that of the mode. Formally, given a query $M_{i,j}$, where $1 \leq i \leq j \leq n$, the result of the query should be a value with at least $1/c$ frequency of the actual mode, i.e. $M_{i,j} \geq \frac{F_{i,j}}{c}$.

3.6 Rank-Space-Reduction

A rank-space-reduction is a method to transform every value in the input array to a new value range between 0 and Δ where Δ is the number of unique values in the array. It is important to note that the mapping is a one-to-one mapping between the original value to the reduced value between 0 and Δ. So if a value a is mapped to the number r then every element in the array that are equal to a is also mapped to r and no other value is mapped to r.

If we keep a map from the reduced value to the original value we can always obtain the original value by performing a look up operation on the kept map with the reduced value as input. Then we will receive the original value as the result.

The algorithm for creation a rank-space-reduction is pretty straight forward. We have two hash maps A and B. Hash map A keeps all the entries from original values to the reduced value and hashmap B keeps the entries from reduced to original. You then iterate over all the elements in the input array and every time you encounter a new unique value you add an entry to A and the reverse entry to B. After you have iterated over all the elements you can apply B to the input array and get a rank-space-reduced array. You can always return the reduced value to the original value by using A.

One advantage of rank-space-reduced values is that you can pack the value more efficiently. If you do not use rank-space-reduction every value must use at least log bits of the highest value. But when we use rank-space-reduced data every value use at least $\log \Delta$ bits which can be significantly lower.

Another advantages is that when you need to count frequencies of element that are rank-space-reduced you can use an array instead of a hashmap. You allocate an array C of size Δ and initialize the array with zeroes. When you want set the counter of a value v you update the array at index v: $C[x] = 5$. Since the is an continues array it is more efficient than a hashmap.

3.7 Fan-out

Fan-out in the thesis always correlates to a tree-structure. Here fan-out means the number of children each node in the tree has. For instance, a binary tree has a fan-out of 2.

Chapter 4

Data structure A

The first data structure we are going to describe is a linear space data structure developed in 2012 by Chan et al.[3] The data structure consists of a few auxiliary arrays, and these arrays are used to perform range mode query in $\mathrm{O}\left(\sqrt{n}\right)$ time while only requiring $\mathrm{O}\left(n\right)$ space. The data structure divides the input data into s blocks where $s = \lceil\sqrt{n}\rceil$.

4.1 Construction

We assume that the input data is rank-space-reduced so every element have a value $0, ..., \Delta - 1$. We denote the rank-space-reduced input data as table A. The first table we construct is denoted Q. It is actually a collection of tables, one for each unique value in A. The table Q contains for every value $v \in A$, all the indices in A where the value v occurs, i.e. $Q_v = \{i|A[i] = v\}$. The ith entry in Q_v is the index in A, where the v occurred i times. In particular, $Q_v[i] = \{k|F_v(0..k) = i + 1\}$.

The next table is B. This array keeps track of the rank of each element. The rank of an element is defined by the elements position in the array compared to other elements of the same value. The rank of an element a at entry i is the number of element preceding elements with the same values as a.

The last data structure we need is a pre-computed table of the mode and frequency. Obviously we can't compute the mode and frequency for every range as that would use $\mathrm{O}\left(n^2\right)$ space. Instead we divide the data array into s corresponding blocks. For each combination of blocks we compute the mode and the frequency of the mode, and store them in two tables: a mode table denoted S and a frequency table denoted S'. Given a range from block i to block j we can return the mode and it's frequency in constant time.

These are the tables needed to perform range mode queries in $\mathrm{O}\left(\sqrt{n}\right)$ time.

4.2 Query algorithm

To explain how we can perform range queries on this data structure, we need to explain a certain property that will be used in the query.

Using table B and table Q we can check if an element a occurs at least y times in a range from i to j in constant time, if we know the position of the first a element within the range.

We have that Q_a stores a list of indices in A where a is. The rank of value a is the number of preceding a values in A. The rank of a is $B[x]$, and is the position in the Q_a table which corresponds to a: $x = Q_a[B[x]]$. So the index of the next a element after position x is $Q_a[B[x]+1]$ and the index of the yth element after x is : $Q_a[B[x]+y]$. Now we can easily check if the index of the yth a value is in the range by comparing it to the end index j: $j > Q_a[B[i]+y]$

When performing a range mode query we start by dividing the range i to j into three sections: prefix, span and suffix. The span is the blocks completely contained within the range, the prefix is the part before the span and suffix the part after the span. Either of the prefix, suffix and span can be empty. The range mode must be either the mode of the span, or a value in the prefix or suffix. We start by looking up the mode and corresponding frequency of the span in the precomputed mode and frequency tables. This mode is our candidate mode, and we need to check if the mode is in the prefix or suffix to find the mode of the entire range.

Now, check each element in the prefix and suffix and start with the lowest index. For each element in the prefix and suffix, check if the element has already been checked, so we don't check the same value twice. This is done similar to how we check if an range contains at least y elements in query algorithm.

We want to check if the value a at index x is the first a value in the range from i to j. We can use the fact that the rank of a is the position in index x in Q_a: $x = Q_a[B[x]]$. The index of the value a just before index x is $Q_a[B[x]-1]$. If the index of the previous value is before the start of the range, i, we know the element a at index x is the first a value in the range: $Q_a[B[x]-1] < i$.

If the value has not been checked yet, we check if the range contains at least as many values as the candidate mode using the B and Q tables as described. If we scan the entire range, the query time would be $\mathrm{O}(j-i)$ instead of $\mathrm{O}(\sqrt{n})$. Instead we use the fact that we know there are at least as many as the candidate. If we denote the current value as a, and its index in A as x we know there are at least as many values as the candidate frequency f. We can then use the fact that $Q_a[B[x]+f]$ the index of the fth occurrence of a. Now we just check the next element in Q_a ($Q_a[B[x]+f+1]$) and if that index is before the end of the range j then we check the next entry until there are no more entries in Q_a or the index is after j. Now we know the frequency of a in the range i to j and we set our candidate mode to a and the frequency to $freq$.

After all the elements in the prefix and suffix have been checked, the candidate mode and frequency is the real mode and frequency of the range $A[i...j]$.

4.3 Space consumption

The data structure consists of five tables: A, Q, B, mode table and a frequency table. Table A contains all the elements, thus requiring $\mathrm{O}(n)$ space. Each value in table Q corresponds to a index in the data array, and no index is repeated

Algorithm 1 Range Mode Query

```
1: procedure RANGE MODE QUERY($A$, $B$, $Q$, $S$, $S$', $i$, $j$)
2:     $span_i = i/\sqrt{n}$
3:     $span_j = j/\sqrt{n}$
4:     $mode = S[span_i][span_j]$
5:     $freq = S'[span_i][span_j]$
6:     for index $x$ in prefix and suffix do
7:         if $Q_{A[x]}[B[x] - 1] < i$ then
8:             if $Q_{A[x]}[B[x] + freq] < j$ then
9:                 while $Q_{A[x]}[B[x] + freq + 1] < j$ do
10:                    $freq = freq + 1$
11:                $mode = A[x]$
12:    return $mode$
```

or left out so the space consumption of the Q table is exactly n. Each value in the rank table also corresponds to exactly one element in the data array so the space consumption is n. The mode and frequency tables do not have the same property as the Q and rank table. The mode and frequency table store a value for each combination of blocks. We have s blocks and the number of combinations is asymptotically $\mathrm{O}(s^2)$. Since the number of block s is equal to $\sqrt{n}$ the space consumption of S and S' is $\mathrm{O}\left(\sqrt{n}^2\right) = \mathrm{O}(n)$.

Since all the tables use $\mathrm{O}(n)$ space, the total space consumption of the data structure is $\mathrm{O}(n)$.

4.4 Query time

We find the prefix, suffix, and span in constant time, and we can also find the candidate mode and frequency in the corresponding tables in constant time. To answer a query, there are three more steps:

1. Determine if the value has been checked before.
2. Determine if there are more values in the range than the candidate frequency.
3. Count the number of values in the range.

We can determine if the value has been already been checked in constant time since it only required a constant amount of look ups in the Q and B tables. To determine if there are more than y occurrences of x in a range is also constant.

The tricky part is counting the number of elements in the range. We have to check every element in the prefix and suffix. That will take $\mathrm{O}(\sqrt{n})$ time, because the prefix and suffix does in worst-case contain $\sqrt{n}$ elements. Then for some elements we have to count the number of occurrences. There are two important things to note. The first one is that we keep ignoring the length of the span, because we do the candidate check in constant time, and we only count the elements in the prefix or suffix. The second things is that every time we count an extra element, we don't need to count it later, which means that we only count every element in the prefix and suffix once in worst-case.

The last step is done in $\mathrm{O}(\sqrt{n})$ time and the checking is done in $\mathrm{O}(\sqrt{n})$ time, which means the total query time is $\mathrm{O}(\sqrt{n})$.

4.5 Construction time

In [3] they describe the tables needed to perform the query in $\mathrm{O}(\sqrt{n})$. They further analyse the space consumption of the data structure. However, the construction of the data structure is never mentioned. It might be that they thought that the construction was trivial so they left it out. But even though the algorithm for constructing the data structure might be quite simple the time complexity of the construction still means a lot.

We describe an algorithm to construct the data structure as well as give a worst-case construction time bound.

The first thing we need to do is apply a rank-space-reduction. As described in section 3.6 this take $\mathrm{O}(n)$ time. Now we need to build the four remaining tables B, Q, S and S'.

The B table contains the rank of each value. We keep a counter for each unique value, and does a linear scan through the input data A. For every index i in A, we insert the value of the counter for the value $A[i]$ into $B[i]$. Then we increase the counter for value $A[i]$ by one, so next time that value is encountered, the rank will be increased by 1. The construction of B can be done in a single linear scan of A so the construction time of B is $\mathrm{O}(n)$.

The Q tables contain the indices of every value, ordered by their value. So Q_a contains the indices of all the values a in A in sorted order beginning with the lowest index. To construct Q we do a linear scan of A from lowest index to the highest. For every index i in A we simply append index i to the end of the associated Q table $Q_{A[i]}$. This is all that is needed to construct the Q tables and since it can be done in one linear scan of A the construction time of Q is $\mathrm{O}(n)$.

Now we only need to construct the mode and frequency tables S and S'. For every block b we compute the mode and frequency from block b until the last block. We keep a set of counters for each value and use these counters to keep track of the frequencies. We also keep track of which frequency is the highest. For every value in the range from block b to the last element of A we update the frequencies, and every time we enter a new block we add the current mode and frequency to the S and S' tables. Since we do at most $\mathrm{O}(n)$ operations for each block, the construction time for S and S' is $\mathrm{O}(sn)$. Since $s = \sqrt{n}$ we can simplify the construction time as $\mathrm{O}\left(n^{3/2}\right)$.

Now we have constructed all the tables needed for data structure A, and the total construction time is $\mathrm{O}\left(n + n + n^{3/2}\right) = \mathrm{O}\left(n^{3/2}\right)$.

4.6 Bit-packing

By using a succinct bit vector we can alter this data structure to support $\mathrm{O}\left(\sqrt{n/w}\right)$ query time where w is the word size. But because of the bit packing the overall mode of the data must have a frequency lower than or equal to $\sqrt{nw}$.

A succinct bit vector is a data structure which packs a string of bits and support 2 operations: *select* and *rank*. The operation $select_1(a)$ gives the position of the ath set bit in the bit vector. The operation $rank_1(a)$ gives the number of set bits before the position a, and $rank_0(a)$. Substituting 1 with 0 in these operations will work on non-set bits instead of set bits. Both *select* and *rank* run in $\mathrm{O}(1)$, and the space requirement for the succinct bit vector is $\mathrm{O}(n/w)$ space where n is the number of bits and w is the word size.

We replace the mode and frequency tables with a list of succinct bit vectors. We can look at table S' as a matrix, where each row i represent the frequencies from block i to block c, and where c is the column number. We can transform each row into a bit vector. We know that the frequency of the mode is always increasing for every row in S', so it is possible to encode the frequency as a list of zeroes. Each zero represents an increase in frequency by 1. We still need to separate the change of frequency for each column and we do that by a 1. This means we have a maximum of n/s ones, because s is the block size. The number of zeroes is bound by the frequency of the overall mode. That is why we need the constraint on the overall mode m, otherwise the bit packing would result in the same query time and only add more operations.

By compressing the frequency table by a factor w we can increase the number of blocks s from $\sqrt{n}$ to $\sqrt{nw}$ and thereby reduce the query time to $\mathrm{O}\left(\sqrt{n/w}\right)$.

If we add the constraint: $m \leq \sqrt{nw}$, we know that each bit vector in S' contains at maximum $\sqrt{nw}$ zeroes, because of the constraint and the number of ones is also at maximum $\sqrt{nw}$ because the block size is set to $\sqrt{nw}$.

Now that the frequency table S' is compressed, we still need to compress the mode table S. The mode can be computed from the compressed frequency table by using the *rank* and *select* operations, so we don't need the S table.

We now have the bit vector frequency table S', but we still need to compute the frequency and mode from the table.

Algorithm 2 Mode and Frequency of span

```
1: procedure MODE AND FREQUENCY OF SPAN($b_i$, $b_j$)
2:     $pos_{b_j} = select_1(b_j - b_i + 1)$
3:     $freq = rank_0(pos_{b_j})$
4:     $pos_{last} = select_0(freq)$
5:     $b_{last} = rank_1(pos_last) + b_i$
6:     for index $i$ in block $b_{last}$ do
7:         if $Q_{A[x]}[B[x] - freq] > b_it$ then
8:             $mode = A[i]$
9:             return $mode$ and $freq$
```

Algorithm 2 shows how to compute the mode and frequency of the span from block B_i to block B_j. We start by finding the frequency in line 1 and 2. At line 1 we find the position of the 1 that separates the last block of the span and the next one. At line 2 we count the number of zeros up to the position of the block and there by get the frequency of the mode. Finding the mode is a bit tricky. In line 3 and 4 we find the last block were the frequency increase. By using a $select_0$ operation on the frequency we get the position of the last zero on line 3. Now we have the position of the last block in the bit vector but we still don't know the actual block number. But we can use a $rank_1$ to find how many ones there are which represent the block number. When we know with block the mode occurred in and we use the Q and B tables to find out if which element in the block has the frequency in side the range. This is similar to the query algorithm where we check if a range contains at least y elements. This is time is done the opposite way but it's still in constant time. The block contains t element so finding the mode and frequency takes $\mathrm{O}(t)$ and since $t = n/s$ the operation takes $\mathrm{O}(n/s)$ time and since $s = \sqrt{nw}$ the finding of the frequency and the mode of the span will take $\mathrm{O}\left(\sqrt{\frac{n}{w}}\right)$.

The range mode query time is bounded by the block size so by decreasing the block size to $\sqrt{\frac{n}{w}}$ the query time is now $\mathrm{O}\left(\sqrt{\frac{n}{w}}\right)$

4.7 Implementation

We have implemented data structure A without bit packing. As noted in the beginning of this chapter we assumed the input data was rank-space-reduced, but in practice we apply the rank-space-reduction when constructing the data structure.

Chapter 5

Data structure B

The second data structure is also developed by Chan et al.[3] It performs range mode queries in $\mathrm{O}(\Delta)$ time and requires linear space. This data structure also contains a number of auxiliary tables. We assume that the input data is rank-space-reduced, and the input array is denoted A.

5.1 Construction

This data structure consists of Δ tables, i.e. each unique value has a table. We will denoted these tables as B, where B_a is the table for the value a. The data set A is divided into blocks of size Δ. For each block i and each unique value v, we compute the frequency of v from position 1 to position $i * \Delta$ and store the frequency in $B_v[i]$. This is done simultaneously for all unique values in a single linear scan, thus populating table B. The ith entry in table B_b is the frequency of v's in $A[1 : i * \Delta]$.

5.2 Query Algorithm

Algorithm 3 Range Mode Query

```
1: procedure RANGE MODE QUERY($A$, $B$, $i$, $j$)
2:     $block_i = (i-1)/\Delta$
3:     $block_j = j/\Delta$
4:     modes = $B[block_j]$
5:     modes = modes - $B[block_i]$
6:     for $a$ in $A[block_i..i-1]$ do
7:         $modes_a = modes_a - 1$
8:     for $a$ in $A[block_j..j-1]$ do
9:         $modes_a = modes_a + 1$
10:    $mode = MAX(modes)$
11:    return $mode$
```

To find the mode of the range $A[i..j]$, we want to find the frequency of all elements in $A[1..i-1]$ and subtract with the frequencies from $A[1..j]$.

We will start by finding the frequency of all values in the range $A[1..i-1]$. This range can be divided into a span and a suffix. The span represents the part of the range witch is covered by our B table. For every value v we can look up the frequency of the span in $B_v[\frac{j}{\Delta}]$. Since the suffix in the worst case contains Δ values, we can do a linear scan of the suffix and add the occurrence of every value in the suffix to the frequencies we got from the span. We now have the frequency of every value in the range $A[1..i-1]$. We can do the same thing for the range $A[1..j]$ and subtract the frequencies from the $A[1...i-1]$ range, which gives us a list of frequencies for every value in the range $A[i..j]$. At this point e can simply return the mode by finding the value with the highest frequency.

5.3 Space consumption

Besides the data array A, we store the frequency table. The array is divided into $\frac{n}{\Delta}$ blocks. For each block we store the frequency of every unique value. We have Δ unique values, so the size of the table is $\frac{n}{\Delta} * \Delta = n$. This is the only information we store, thus the data structure uses linear space.

5.4 Query time

Look up in table B is done in constant time. Since the suffix of each range $A[1...i-1]$ or $A[1...j]$ does not contain a full block, we know that the maximum amount of values is $\Delta - 1$. Performing the frequency count of range $1-i$ and $1-j$ take $\mathrm{O}(2\Delta) = \mathrm{O}(\Delta)$ time. When we have the frequencies for the ranges, we do a subtraction which takes Δ time, and finally we find the maximum frequency, which takes Δ time. Combining the steps, we have a worst-case query time of $\mathrm{O}(\Delta + \Delta + \Delta) = \mathrm{O}(\Delta)$.

5.5 Construction time

This data structure can be constructed in linear time. We do a single linear scan of A to fill out B. We keep a set of counters for each unique value, and then begin the linear scan. For every value we increment the corresponding frequency counter by 1. After Δ values we add the first entry to the B table for every unique value. When this process reached the end of A, table B is complete. As we do a single linear scan, the construction time of the data structure is $\mathrm{O}(n)$.

5.6 Implementation

As noted in the beginning of this chapter we assumed the input data was rank-space-reduced, but in practice we apply the when construction data structure B. We think this is a better way to perform the experiments, as the implementation then can work on any set of data with the rank-space-reduction as a mapping function, which closer depicts a real world scenario, where the data set most likely is not perfectly aligned.

Chapter 6

Data structure A + B

In [3] they combine data structure A and B into a single data structure to achieve a query time of $\mathrm{O}\left(\sqrt{\frac{n}{w}}\right)$ without the constraint of data structure. This is done because when using data structure A with bit packing we have the constraint of the overall mode m most be lower or equal to $\sqrt{nw}$.

6.1 Construction

We start by scanning the array and find the frequency of all elements. All elements with a frequency lower than or equal to s, where $s = \sqrt{nw}$, is put into a new array A, and the remaining elements are put into array B. We then create four arrays of the same size as the input array: A_i, A_j, B_i, B_j. The ith entry in the A_i array is the corresponding starting index in the array A. A_j is the ending index for corresponding query to j in array A. B_i and B_j are the same as A_i and A_j just for array B instead of A. Now we apply data structure A to array A and data structure B to array B.

6.2 Query

The query algorithm is rather simple. Given a range i to j, we use A_i and A_j to find the corresponding range in the array A, and B_i and B_j to find the range in array B. We can now perform a range query in A using data structure A by using index $A_i[i]$ and $A_j[j]$. We do the same thing for the B using data structure B. Given the mode and frequency from A and B we simply return the mode with the highest frequency.

6.3 Query time

The idea is to use data structure A for all values where the frequency is lower than or equal to $\sqrt{nw}$, and data structure B for the rest of the values. As every value added to data structure B has at least a frequency of $\sqrt{nw}$, there can be at most $\sqrt{\frac{n}{w}}$ unique values assigned to data structure B, which mean that the query time of B is at most $\mathrm{O}\left(\sqrt{\frac{n}{w}}\right)$. Since both data structures have the same query time, the total query time is $\mathrm{O}\left(\sqrt{\frac{n}{w}}\right)$.

6.4 Space consumption

As stated in Chapter 4 and Chapter 5 the space consumption of both data structure A and B is linear. Besides the two data structures, this data structure stores four additional arrays: A_i, A_j, B_i and B_j. All of these arrays map an index to the two data structures, and since there are a linear amount of indexes, these four array have linear size. Thus the space consumption of data structure A + B is $\mathrm{O}(n)$.

6.5 Construction time

The creation of A_i, A_j, B_i and B_j can all be done in a single scan on the input data set. As stated in Section 4 the construction time of data structure A is $\mathrm{O}(ns)$ and since we increased s to $\sqrt{nw}$ we get a slightly higher construction $\mathrm{O}\left(n^{3/2}\sqrt{w}\right)$. In Chapter 5 we stated that the construction time of data structure B was $\mathrm{O}(n)$. When we add this up we get the total construction time of data structure A+B, $\mathrm{O}\left(n + n^{3/2}\sqrt{w} + n\right) = \mathrm{O}\left(n^{3/2}\sqrt{w}\right)$.

Chapter 7

Datastructure: 3-Approximate Range Mode Queries

In this chapter we will go through a data structure to answer approximate range frequency queries in constant time, presented in 2010 by Greve et. al.[6]. Later we describe how to augment the data structure to answer range mode queries. This data structure answers 3-approximate range mode queries, which essentially means that it returns a value that occurs at least 1/3 times as frequent as the actual mode of the range. We will show that the data structure is consuming $\mathrm{O}(n \log \log n)$ space and answers queries in constant time, $\mathrm{O}(1)$. In addition to the original paper we contribute by describing a construction algorithm running in $\mathrm{O}\left(n^{3/2}\right)$ time.

In the end of the chapter we discuss how we implement the data structure in C++, and describe the iterations we went through while implementing the construction of the data structure. Our final solution is not the most optimal in theory, however the implementation that was good in theory used a lot of memory, which took a lot of time for the operating system to allocate. Therefore we ended with a data structure that uses less memory in the construction phase and faster overall due to less memory allocations and memory copying.

The original paper by Greve et al.[6] doesn't focus on the construction time of the data structure, so it was an obvious field to research for us. This is also the reason we went through the different iterations to speed up the implementation. In the experiments our main focus is on the construction time versus query time, especially how many queries is needed to outweight the construction time compared to the other data structures described and implemented in this thesis. As 3-approx is an approximate data structure, we further compare the correctness and deviation of the data structure to that by Bose et al.[2] as described in Chapter 8.

7.1 Theory

Observe that given three arrays A_1, A_2, A_3, let A be the concatenation of A_1, A_2 and A_3, a third of the frequency of A will be less than or equal to the maximum frequency A_i for all $i = 1..3$, and an array A_i will at most have the

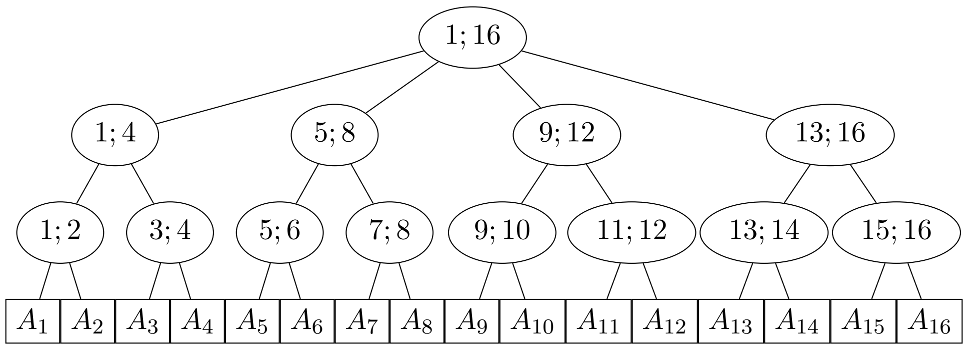

Figure 7.1: Example of 3-approx tree on a data set, $n = 16$. The square nodes are the input array, round nodes are the data structure tree nodes indicating what range they span.

same frequency as A for any i.

Formally, given an array $A[i, j]$ and three disjoint sub-arrays $A[i, x]$, $A[x + 1, y]$ and $A[y + 1, j]$, we define $F_{a,b}$ to be the frequency of $A[a, b]$, in which case we have,

$$\frac{1}{3}F_{i,j} \leq \max\{F_{i,x}, F_{x+1,y}, F_{y+1,j}\} \leq F_{i,j}.$$

It is trivial to verify the correctness of this statement.

7.1.1 Building the data structure

Given an array of data values A, we build the data structure, which uses $\mathrm{O}(n \log\log n)$ space and answers queries in $\mathrm{O}(1)$ time. We start off with a tree T of polynomial fan-out with a total of n leaves, where leaf i represents $A[i]$, for $i = 0, ..., n - 1$, i.e. all the values in the array is leaf nodes in the same order as in the array, such that the left-most leaf is also the first value in the array, and the right-most leaf is the last value in the array. This is illustrated in Figure 7.1 with an example of a tree on an input A of size 16.

The round nodes indicate the interval in the input data array, the node is responsible for, or spans. A node $(i; j)$ answers range queries for the sub-array $A[i..j]$. Each of these nodes hold a set of prefix, suffix and child pair frequencies. In the bottom we see the entire input array, A, which is the raw data, and thus the data structure is built on top of it. Each non-leaf node has the frequencies of the children's prefix and suffix as well as the frequencies of the pair of the children. Because of each node stores this data, we are can look up the frequencies when we have the least common ancestor node when querying. We look in the children pair frequencies for ranges that spans all the children, prefix and suffix are for queries that don't span an entire child.

Formally, for a node v, let T_v denote the sub-tree of T rooted at v. The notation $|T_v|$ denotes the number of leaves in the tree T_v, i.e. how many values in the tree represents. The fan-out[1] of a node v is denoted $f_v = \lceil\sqrt{|T_v|}\rceil$. For

[1]For definition of fan-out, see Chapter 3

this choice of fan-out we have that the height of the tree T is $\Theta(\log \log n)$, which gives us some great properties that will become clear in the analysis of the run times.

For every node $v \in T$, we have R_v to denote the consecutive range of values that v spans over, i.e. the consecutive leaves of the sub-tree T_v. If T_v spans over the values in $A[a_v, b_v]$, we have that $R_v = A[a_v, b_v]$, where a_v denotes the first value that v spans over, and b_v denotes the last value v spans over. The children of v, $c_1, ..., c_{f_v}$ then partitions R_v into f_v disjoint sub-ranges, $R_{c_1}, ..., R_{c_{f_v}}$ each of size $\mathrm{O}\left(\sqrt{|T_v|}\right)$. For every pair of children (c_r, c_s) where $r < s - 1$ we store $F_{a_{c_{r+1}}, b_{c_{s-1}}}$. At last we store for every child range R_{c_i} all prefix and suffix ranges of R_{c_i}. I.e. we store $F_{a_{c_i}, k}$ for all prefixes and $F_{k, b_{c_i}}$ for all suffixes.

Besides storing T we also store a lowest common ancestor (LCA), which given two indices i and j finds the least common ancestor of the leaves corresponding to the indices in T. This query can then be done in constant time.

7.1.2 Querying

To answer a query from i to j, we first get the LCA of A_i and A_j. The LCA node is the node in T where A_i and A_j lie in different child sub-trees, T_{c_x} and T_{c_y} with ranges $R_{c_x} = A[a_{c_x}, b_{c_x}]$ and $R_{c_y} = A[a_{c_y}, b_{c_y}]$, respectively. We look up the frequency $F_{a_{c_{x+1}}, b_{c_{y-1}}}$ which we stored for the pair of children c_x and c_y. We also look up the prefix and suffix frequencies. From the observation described at first, the max of these frequencies are at least 1/3 of the total frequencies, thus we return the maximum of the frequencies and we have the 3-approximate range mode query.

7.1.3 Analysis

The analysis contains the space and query analysis of the data structure. Similar analysis can be found in the article by Greve et al.[6] Furthermore we give an upper-bound on the construction time of the data structure.

Space analysis We start by looking at the space consumption on the prefix and suffix of each node. How much space is required for node v depends on $|T_v|$, which in turn depends on what level in the tree v is on. Consider node v being on level i, the space required for prefix and suffix is $2n^{1/2^i} = \mathrm{O}\left(n^{1/2^i}\right)$, where the root is on level 0. Level i then uses $\mathrm{O}\left(n^{1/2^i} n / (n^{1/2^i})\right) = \mathrm{O}(n)$ space for level i. Additionally, each node v stores the frequency of the children pairs. A node v on level i have $\sqrt{n^{1/2^i}}$ children, which means the number of pairs is $\mathrm{O}(n)$. In total for each level we have $\mathrm{O}(n) + \mathrm{O}(n) = \mathrm{O}(n)$. The height of the tree is $\log \log n$, thus the total space consumption for the tree is $\mathrm{O}(n \log \log n)$.

Besides our tree structure we also have the LCA, which requires $\mathrm{O}(n)$ space as shown by Dov and Tarjan in 1984[7].

Query time Looking up the lowest common ancestor is $\mathrm{O}(1)$ when using the LCA data strcuture by Dov and Tarjan[7]. When we have the LCA, it requires a maximum of three look ups in the LCA's child, prefix and suffix frequencies, which is done in constant time by using a simple array lookup and arithmetics. Now all that is left is to compare the three potential modes. This results in a $\mathrm{O}(1)$ constant query time for any query.

Construction Time The construction consists of calculating the prefix, suffix and children frequencies. We first look on the prefix and suffix frequencies, which both have the same time complexity. For a node v on level i there is $2f_v$ prefixes and suffixes to calculate. If the prefix and suffixes are calculated consecutively, we effectively scan over all the values the node spans, i.e. $|T_v|$, adds them to a hash map and calculates the frequencies. The number of values T_v spans is given by,

$$|T_v| = n/n^{1/2^i},$$

where level $i = 0$ is the root value. As the entire level i has a total of $n^{1/2^i}$ nodes, the total cost of all the prefixes and suffixes for level i is then

$$2|T_v|n^{1/2^i} = \mathrm{O}(n)$$

We have $\log\log n$ levels, which makes the total time to calculate prefixes and suffixes $\mathrm{O}(n \log\log n)$.

Now we need to account for the calculation of the children pair frequencies. Each node needs to calculate pairs of all its children. A node on level i will at most use $\mathrm{O}\left(\frac{n}{n^{1/2^i}}\right)$ time for each sequence of pairs starting with c_k, and there are at most $n^{1/2^{i+1}}$ of such sequences, thus a total of $\mathrm{O}\left(\frac{n}{n^{1/2^i}} n^{1/2^{i+1}}\right)$ per node on level i. As there are $n^{1/2^i}$ nodes at level i, the total of level i is $\mathrm{O}\left(n^{1/2^{i+1}} n\right)$. For the root level, that is $\mathrm{O}\left(n^{3/2}\right)$, for level 1 $\mathrm{O}\left(n^{5/4}\right)$, for level 2 $\mathrm{O}\left(n^{9/4}\right)$, and so on. That results in the child-pair calculations being done in a total of $\mathrm{O}\left(n^{3/2}\right)$.

The total construction time is thus,

$$\mathrm{O}\left(n \log\log n + n^{3/2}\right) = \mathrm{O}\left(n^{3/2}\right)$$

7.1.4 From frequency to mode

The data structure described returns an approximation of the frequency of the mode rather than the mode of the range. We can easily augment the data structure to either return the mode or both by storing the mode along with the frequency. This will not break the complexity as we will only store a constant integer more than previously.

In particular, while building the data structure, we store a tuple of the mode and frequency, thereby the space consumption will be doubled, which is still within the bound of $\mathrm{O}(n \log\log n)$. When querying we still have a constant number of look ups and comparisons, and thus still $\mathrm{O}(1)$.

7.2 Implementation

We had three iterations while implementing the 3-approximate data structure. We will mainly focus on construction here, as all the iterations of our implementation have the same space and query complexity.

First implementation

The implementation is relatively simple for this data structure. Instead of having a constant time LCA look-up, we have a $\mathrm{O}(\log\log n)$ LCA look-up. We trade off the $\mathrm{O}(n)$ space complexity with an additional $\mathrm{O}(\log\log n)$ tree traversal LCA look up, as this look up requires very few worse-case look-ups in the tree structure. We find the lowest common ancestor by a simple traversal, easily described by the pseudo code in algorithm 4. On line 1 we get the number of values the node spans. Line 2 and 3 give the indices of the children spanning the two leaves, respectively. If the children is the same, we search recursively in that child, otherwise we have the LCA, because we start the recursion in the root node.

Algorithm 4 Lowest Common Ancestor

```
1: procedure FindLCA($node$, $leafidx_1$, $leafidx_2$)
2:     $elements \leftarrow node$.num_elements()/$root$.num_children()
3:     $childidx_1 \leftarrow leafidx_1 / elements$
4:     $childidx_2 \leftarrow leafidx_2 / elements$
5:     if $childidx_1 = childidx_2$ then
6:         $offset \leftarrow childidx_1 * elements$
7:         return FindLCA($node$, $leafidx_1 - offset$, $leafidx_2 - offset$)
8:     else
9:         return $node$
```

A node calculates all the prefix and suffix frequencies for all its children. The prefix and suffix is calculated in a very naive manner. For each prefix and suffix, we iterate over all the values that is spanned, add all values to a hash map and calculate the frequency. To calculate children pair frequencies, we simply add all values spanned by the children pair, adds to hash map and calculates the frequencies.

This is very simple and straight-forward. However, the complexity is not constant as the theory suggests. There can potentially be as many recursive calls as the height of the tree, which is $\log\log n$. This will not hurt our query-time that much, but it does mean that our implementation has a query time of $\mathrm{O}(n \log\log n)$.

When calculating the prefix, suffix and child pair frequencies, we do it as described by Algorithm 5. First we calculate the prefix and suffix frequencies for each child by iterating all the different sizes of prefixes and suffixes as seen on line 3. For each of these iterations we initialize two hashmaps to keep track of the number of occurrences of the different values. We then populate the hashmaps with the values in line 7 and 8. On line 9 and 10 we calculate the frequency and mode in the hash maps and adds them to the node's prefix and suffix lists. Pretty much the same is done for the children pairs. We get the pair of child indices on line 11, iterating all the children in between and adding all their values to as hashmap, then calculates the frequency and mode for the hash map and adds the child frequency to the node's list.

Algorithm 5 First Frequency Calculation

```
 1: procedure CalcFreq($node$)
 2:     for each child $c$ in $node$.Children() do
 3:         for $i = 0..\text{c.Values().Size()} - 1$ do
 4:             $prefix \leftarrow$ hashmap
 5:             $suffix \leftarrow$ hashmap
 6:             for $j = 0..i$ do
 7:                 $suffix[c.\text{Values()}[j]]$+=1
 8:                 $prefix[c.\text{Values()}[\text{Size()} - j - 1]$+=1
 9:             $node$.AddSuffixFreq($c$, Max($suffix$))
10:             $node$.AddPrefixFreq($c$, Max($prefix$))
11:     for each pair of indices $c_1, c_2$ in $node$.Children()) do
12:         $child_H \leftarrow$ hashmap
13:         for $i = c_1, ..., c_2$ do
14:             $child_H$.AddAllValues($node$.Children()[$i$]
15:         $node$.AddChildFreq(Max($child_H$))
```

Improving construction time

When looking through the pseudocode in Algorithm 5 it becomes clear that we can optimize it a lot. As we continually add values to the hash maps in the prefix and suffix calculation we add the same values multiple times. In fact, for a node with m values, prefix and suffix takes $\text{O}\left(m^2\right)$. Instead we realize that when we add a value to the hash map we don't need to remove it again for that node, effectively making the process a linear scan. Similarly, when we calculate the frequency and mode we only need to consider the value we just added instead of re-calculating the frequency and mode for the entire hash map for each added value.

Similarly we realize that we can reuse the hash maps we generated for prefix and suffix to calculate the children pairs, because the prefix occurrences hash maps of the node's children spans all the nodes we need to consider in the children pairs. Therefore we save the prefix hash maps of the children's prefix and instead merge hash maps when adding iterating the children instead of adding all the values. The difference in runtime in this case depends on the number of unique values in the children.

The pseudocode improving the construction time is seen in Algorithm 6.

For calculating $freq(c_1, c_2)$ we add the suffix hash map of c_2 to the hash map of c_1. Now, when we calculate $freq(c_1, c_3)$ we add c_3 hash map to the suffix map of c_1 which at this point hold the values of both c_1 and c_2. This way

Algorithm 6 Second Frequency Calculation

```
1: procedure CalcFreq($node$)
2:     $prefix_C \leftarrow$ hashmap
3:     for each child $c$ in $node$.Children() do
4:         $prefix_H \leftarrow$ hashmap
5:         $suffix_H \leftarrow$ hashmap
6:         for $i = 0$..c.Values().Size() $- 1$ do
7:             $suffix_H[c.\text{Values}()[i]] {+}{=} 1$
8:             $c$.AddSuffixFreq(Max($suffix_H$))
9:             $prefix_H[c.\text{Values}()[\text{Size}() - i - 1]] {+}{=} 1$
10:            $c$.AddPrefixFreq(Max($prefix_H$))
11:        $prefix_C[c] \leftarrow prefix_H$
12:    for each pair of indices $c_1, c_2$ in $node$.Children()) do
13:        for $i = c_1 + 1, ..., c_2$ do
14:            $prefix_C[c_1]$.Merge($prefix_C[i]$)
15:        $node$.AddChildFreq(Max($child_C[c_1]$))
```

when calculating a child pair $freq(c_i, c_j)$ we only add the unique values that c_j spans over, which is at most $\frac{n}{n^{1/2^i}}$ where c_j is on level i in the tree (root starting at level 0). This way of calculating the child pair frequencies, it is important that we iterate in a specific order as we merge one hash map into another. We want to iterate the smallest indices first: $(1,1)$, $(1,2)$, ..., $(1,n)$, $(2,2)$, $(2,3)$, ..., $(2,n)$, and so on.

Improving construction memory use

By some memory profiling we saw that this solution required a lot more memory than we anticipated. There is two causes for this. Firstly, we now, albeit temporary, store a hash map for each child containing all the memory it spans, and, secondly, we saw the hash map implementation of unordered_map in c++ to be quite memory heavy. The implementation allocates much more memory than needed in an attempt to avoid re-allocating when more keys are added to it. However, we knew the exact limit for values to be inserted in our hash map, so this was an unnecessary feature we would be better without.

While the first point makes our construction time a bit faster in theory, it needed a lot more memory allocations, which is quite costly in practice. We therefore decided we were better off without that micro-optimization. As to the second point, we ended up using the c++ array abstraction, vector, together with a rank-space-reduction approach. This way we had more control of the memory allocations. Additionally, we now only allocate the memory to be used for these frequency calculations once instead of for every single node.

The pseudocode in Algorithm 7 shows how our solution at the end looks like. We apply a rank-space-reduction on our input data, allocates our vector for calculations of size Δ, which is the unique values of our input data, and then do all the calculations as previous with the small modification described above. We describe how our rank-space-reduction works in Chapter 3.

Additionally, when the occurrences vector is populated we now find the frequency of the prefix and suffix in constant time instead of the linear scan we did before. The way we are doing this is by storing the previous frequency of both suffix and prefix, and then comparing the old frequency with the new frequency when we add it to the frequency map. As we only add one value at a time, we are guaranteed that either the previous frequency or the current is the maximum.

Algorithm 7 Final Frequency Calculation

```
procedure CalcFreq($input$)
    RankSpaceReduce($input$)
    $nodes \leftarrow GenerateTree(input)$
    $calc_H \leftarrow$ vector($\Delta$)
    for each $node$ in $nodes$ do
        for each child $c$ in $node$.Children() do
            $suffix_{\text{prev}} \leftarrow \infty$
            $prefix_{\text{prev}} \leftarrow \infty$
            $calc_H$.Clear()
            for $i = 0..$c.Values().Size() $- 1$ do
                $suffix_{\text{cur}} \leftarrow (calc_H[c.\text{Values}()[i]] {+}{=} 1)$
                $suffix_{\text{prev}} \leftarrow \text{Max}(suffix_{\text{prev}}, suffix_{\text{cur}})$
                $c$.AddSuffixFreq($suffix_{\text{prev}}$)
            $calc_H$.Clear()
            for $i = 0..$c.Values().Size() $- 1$ do
                $prefix_{\text{cur}} \leftarrow (calc_H[c.\text{Values}()[\text{Size}() - i - 1]] {+}{=} 1)$
                $prefix_{\text{prev}} \leftarrow \text{Max}(prefix_{\text{prev}}, prefix_{\text{cur}})$
                $c$.AddprefixFreq($prefix_{\text{prev}}$)
        for each pair of indices $c_1, c_2$ in $node$.Children()) do
            $calc_H$.Clear()
            for $i = c_1, ..., c_2$ do
                $calc_H[c_1]$.AddAllValues($node$.Children()[$i$])
            $node$.AddChildFreq(Max($calc_H[c_1]$))
```

Overall improvement

From our first implementation we have now obtained an improved construction speed; before, the calculation of frequencies for each node took $O(m^2)$, where m is the number of values the node spans, to a calculation time of $O(m)$. The same space in memory is required as before, however we only allocate the memory once, which is a significant improvement to performance. The amount of space required to do these calculations is $O(\Delta)$, where Δ is the number of unique values in the input data.

The data structure implementation of construction time is specifically improved upon to make it viable for discussion, when we discuss the experiments.

We focus on the construction time and range mode query speed as well as correctness of the approximate data structures, thus the time of the construction algorithm is important in this context.

Chapter 8

Datastructure: $(1+\epsilon)$-approximate

In the previous data structure we described a way to answer 3-approximate range mode queries in constant time, i.e. we are guaranteed that a query answers with a mode that is at least 1/3 as frequent as the actual mode. If $\frac{1}{3}$ is not a tolerable error margin, one would need to use a data structure giving a stronger bound. In this case we consider a data structure described by Bose et al.[2] that can answer queries in $\log\log_{1/\alpha} n$ time using $n\log_{1/\alpha} n$ space, where $0 < \alpha < 1$ is the approximation factor of the data structure, giving a value which frequency is at least $1/\alpha$ of the frequency of the mode. In the theory part we will use the terminology from the article by Bose et al.[2], i.e. the data structure is a $(1/\alpha)$-approximate data structure. However, apart from this section we refer to the data structure as $(1+\epsilon)$-approximate. This is a simple transformation, which is easily seen,

$$\begin{aligned} \frac{1}{\alpha} &= (1+\epsilon) \\ \Rightarrow \alpha &= \frac{1}{1+\epsilon} \end{aligned}$$

That means the step size described in the theory becomes the following in the implementation,

$$\begin{aligned} \frac{1}{\alpha^k} &= \frac{1}{\frac{1}{1+\epsilon}^k} \\ &= (1+\epsilon)^k \end{aligned}$$

8.1 Theory

8.1.1 Approximating Ranges From One Point

Given an approximation factor α and an input array $A = a_1, a_2, ..., a_n$, if we only consider the ranges starting from the left-most entry, a_1, we can later use the same method for every entry in the data input to get our data structure.

We want to be able to answer queries going from a_1 to a_i, where $1 \leq i \leq n$ is the other end of the range query with an approximating of $1/\alpha$ of the frequency of the mode. We define a lookup table B that stores the approximated modes, $b_1, b_2, ...$ The first entry b_1 will always be the same value as the first entry in the data array, a_1. The next value, b_2, is the value that first value that occurs $\lceil \frac{1}{\alpha} \rceil$ times in A, i.e. $F_{b_2}(a_1, ..., b_2) = \lceil \frac{1}{\alpha} \rceil$ and there is no frequency of a value in the same range that exceeds the frequency of b_2 in that range,

$$\forall \{x \in A | x \neq b_2\} : F_x(a_1, ..., b_2) < F_{b_2}(a_1, ..., b_2).$$

To generalize this idea, b_k is the first value that has occurred $\lceil \frac{1}{\alpha^{k-1}} \rceil$ times in the sub-array as the right side of the query range increases. We have that b_k is the approximation of the mode of the sub-array $A[1, j]$, where $b_k \leq j \leq b_k + 1$. The reason this works is because we have the actual mode in b_k and b_{k+1}, so for all the entries between these two approximation points will not exceed our approximation factor because of the way we have chosen our approximation modes. For the general case we have that the following invariant for our approximation modes,

$$\forall \{x \in A | x \neq b_k\} : F_x(a_1, ..., b_k) < F_{b_k}(a_1, ..., b_k).$$

We store m approximation modes, and the last approximation mode, b_m, will naturally occur at least $\lceil \frac{1}{\alpha^{m-1}} \rceil$ times in the array A. It is clear that the number of approximation modes m we need to store for these invariants to be true is at most $\log_{\frac{1}{\alpha}} n + 1 = \mathrm{O}\left(\log_{\frac{1}{\alpha}} n\right)$.

When answering a range mode query, we look in our B table with approximation modes. For query $(1, i)$ we do a binary search in the table B to find the approximation mode that is less than i. Let b_k be that mode. Because of how we constructed the look up table B, we know that frequency cant be off by more than bound by the approximation, thus we have our approximation mode answer.

Complexity As the approximation table B needs to store $\log_{\frac{1}{\alpha}} n$ entries, that is how much space the data structure is consuming. With regard to query time, we need to a binary search in the table B, because the modes clearly are sorted by frequencies. The query time for the data structure is thus $\log \log_{\frac{1}{\alpha}} n$.

8.1.2 Approximating All Ranges

In the data structure described above, we assume to have a fixed start entry. Effectively, we can only answer queries starting from the first entry in the array.

It is quite simple to construct a data structure that can approximate any arbitrary range. If we simply store n of the look up tables B described above, one for each entry in the array, and for a range mode query $Q(i, j)$ we query table $B'[i]$ as described above by doing a binary search in the array for the value below j.

Complexity The space complexity now has a factor n more than the single look up table, which means the space complexity is now $n \log_{\frac{1}{\alpha}} n$.

Querying is the same as before. The only difference is a single constant time look up for the right approximation table. The query time is thus still $\log \log_{\frac{1}{\alpha}} n$.

8.2 Implementation

We applied a rank space reduction to the data before creating the data structure as it would be much faster to count the frequencies of every element and since we have to do it $\mathrm{O}\left(n^2\right)$ times we believe that the trade of was worth it. This means that besides the tables need to perform the queries we also store the rank-space-reduced input data and a map which can be used to return the mode to it's original value. For every entry in the input data we have to computed as list of indexes. Given an index we find the predecessor of that index in the array. the value of the predecessor have to be in worst-case an $1 + \epsilon$ approximation of the mode.

The way we build a table T_i for every index i we scan the range $A[i..n]$. While we do the linear scan we keep track of the frequency of every value. This book keeping is done in an array since we have done a rank-space-reduction and the value are in the range $0..\Delta - 1$.

While the frequency is below $1/\epsilon$ we added every element the increases the frequency of the mode to the list T_i. When the frequency of the mode have reached $1/\epsilon$ we multiple the current frequency with $1 + \epsilon$ to get frequency need for the next element. By doing so we ensure that when the frequency of the mode increases by a factor $1 + \epsilon$ we add a new index to T_i. When continues to increase the frequency needed back a factor $1 + \epsilon$ every time we add an index to T_i until we have scanned the entire array.

The query algorithm is quite simple. Given a range from index i to index j we find the table corresponding to index i T_i and the do a binary search for the index just below j. We then return the element at the index as the $1 + \epsilon$ approximate mode.

Chapter 9

Additional Implementations

Apart from the more advanced data structures, there are two very simple approaches to the range mode query problem, which we explain in detail here. We only test with the Naive approach as the greedy approach is impractical for most scenarios as we explain.

9.1 Naive approach

A straight-forward approach to the mode query problem is to not do any pre-computing and only do work when answering a query. For a query from a to b on your input data v of size n, we simple iterator over the entries $v[a], v[a+1], ..., v[b]$. If the label on the entry doesn't exist in our hashmap, we add it with the value 1, otherwise we increment the value by one with the label as key. At this point we have a hashmap with all the unique values and occurrences of the values from a to b. We then iterate the hashmap and returns the values containing the highest frequency.

9.1.1 Construction time

We don't need to do any pre-processing on the input data. Our query works on the raw input data, thus we are not required to build anything.

9.1.2 Query time

For a query ranging from a to b, we linearly scan over the entries from a to b in the data array and insert all of those values in our hash map. This takes $\mathrm{O}(b-a)$ time, as it is indeed a linear scan with hash map having amortized constant insertion time. Now we iterate over all the entries in the hash map and find the value with the most occurrences. The number of entries depends on the number of unique values in the range a to b, which clearly is lower than $b-a$, thus giving us a total query time of $\mathrm{O}(b-a)$.

9.1.3 Space consumption

All we need is the input data. Apart from that we are not needed to store any information, as we work on the data as we get the queries.

9.2 Naive Approach (Rank space)

As we have noticed the implementation of unordered_map is really slow, we also consider the naive approach by rank space reducing the data in construction. This has the benefit that we know how many unique values our data has, and thus can use an array of size Δ to calculate the frequencies of each element, which should be very fast in practice compared to the hash map implementation in C++ STL.

9.2.1 Construction time

When constructing this data structure on a data set of n values, we add iterate all n values and insert all unique values in a hash map, which is Δ values. Insertions in a hash map is expected $\mathrm{O}(1)$. Then we map every value in the data set to a new data set based on this hash map. Lookups in the hash map is constant, and we have n values to map. This gives us a construction time of $\mathrm{O}(\Delta + n) = \mathrm{O}(n)$.

9.2.2 Query time

The query time complexity of naive with rank space reduction is the same as the naive without rank space reduction. We have a pre-allocated array with n values, where we can update the frequency count and look up frequency counts in constant time. When calculating the frequency count we also keep track of the current mode and the frequency of the current mode. When updating the frequency of a value, we check if the frequency of the value we are updating is larger than the frequency of the current mode. If that is the case, we update the current mode, otherwise we iterate to next element in the range.

As we iterate $b - a$ elements, and each iteration does a constant amount of work, i.e. update frequency counter and check if frequency counter is greater than the current mode, the complexity of the query algorithm is $\mathrm{O}(b - a)$.

9.2.3 Space consumption

In theory if we don't need to store the data itself, we still need to have the rank space reduced data, which means we need to transform all the data to the rank space reduced data, which is n values of data. Additionally we keep the reverse transformation mapping, so we can return the actual mode rather than the mode in terms of rank space reduction, which is a mapping of size Δ, i.e. the number of unique elements. The total space consumption is thus,

$$\mathrm{O}(n + \Delta) = \mathrm{O}(n)$$

9.3 Greedy approach

The range mode query problem can be solved in a very fast worst-case time, but this requires every possible mode to be pre-computed, which will use n^2 space. If that is done, we can then answer queries by doing a single look up in an array.

This data structure is not very interesting because of the memory use. Although the constant query time is great, the cost of memory is way too high to be feasible in the far majority of cases. Memory in general is way more expensive than CPU time, so the trade-off is not looking promising in a practical scenario. We don't use it in the experiments, but include an analysis of it for completeness.

9.3.1 Construction time

We have to compute all modes of all ranges in the input data. This requires us to look on every combination of pairs of indices. For a data set of size n, we have $\frac{1}{2}(n-1)n$, which results in an asymptotic upper bound of $\mathrm{O}\left(n^2\right)$.

9.3.2 Query time

When querying, we need to do a single look up on a specific index in our hash map. As look ups in hash maps is done in constant time and as the index can be calculated with simple arithmetic operations, the query time is constant.

9.3.3 Space consumption

The very bad thing about this data structure is that every single combination of modes are being calculated and stored, which is very memory intensive. For a data set of size n, we need a mode for each combination of indices, thus the total space consumption for this data structure is $\mathrm{O}\left(n^2\right)$.

Chapter 10

Experiments

We have two different kinds of data structures. One answers a query with an accurate answer, and one type answers queries with an estimated answer. Naturally, the theoretical query times of the estimating data structures is way better than the accurate ones.

We consider different scenarios to try finding out, which of the data structures have a place in practical applications, and if so, in which scenarios should which data structures be used. The data structures require some pre-processing in order to build the structure to efficiently answer queries, and as this pre-processing takes time, an important aspect of choosing the optimal data structure depends on the construction time compared to query time and number of expected queries. Likewise, some data structures might be better in a setting where big range sizes are expected versus small range sizes.

The ultimate goal of the experiments is to account for advantages and disadvantages of the different data structures, and eventually propose which data structures to use in various use cases depending on the data and setting.

10.1 Test Data

The parameters we choose are different kind of data distributions, different kind of query sizes as well as estimated number of queries.

10.1.1 Data distribution

We experiment with different data distributions. We use uniform random data to get close to different estimated unique elements in our data. Some data structures have a theoretical run time in terms of unique elements in input data, thus testing those theoretical bounds seems like a good indicator of how the performance scales. We are using three different range limits for our data:

1. Δ_{constant} , having a constant number of unique values, in particular 100.
2. $\Delta_{\sqrt{n}}$, scaling the unique number of values with square root.
3. Δ_{linear} , scaling the unique number of values linearly with n, in particular with a factor 1/10.

10.1.2 Number and length of queries

As most of the data structures have a building phase, it is important to look on how expensive this phase is. Additionally, we need to compare the construction with the queries. If a very large number of queries are expected to be run on the data, it is probably feasible to build the data structure compared to a linear-scan for each query. However, this is something we are looking into, finding the sweet spot for number of queries needed for it to be feasible to construct the data structures rather than running the linear naive approach.

There is an additional aspect of this. To consider the number of queries, we also need to consider the query range size. How small ranges and large ranges compare on the different data structures is also important to determine when the construction time is out-weighted by the query time.

The best approach would be to generate all possible ranges for every data size but that would make the running time of the experiments but that would make the experiments take too long to run. So instead we generate a number random ranges queries.

We use a linear number of range queries per data size. This is done to make sure the random generated ranges does provide a reasonable result when the data size is increasing. In our experiments we made $\frac{1}{80}n$ range queries per data size where n is the data size in bytes.

Besides the number of ranges we also need to decide how to construct the actual ranges. We randomly choose two indexes in the range $1...n$ and creates a ranges query from the lowest index to the highest. This gives an expected range query length of $\frac{1}{3}n$.

10.1.3 Data size

We start off with a data set of 8 kilobytes going in a linear interval to a total of 2 megabytes. This gives us 10 different data size samples, which seems reasonable to give a concise answer to use case.

10.2 Hardware

Processor	Intel Core i7 930 @ 2.80GHz
L1 cache size	256 kB
L2 cache size	1024 kB
L3 cache size	8192 kB
Memory size	6 gB

10.3 Software

CPU architecture	64 bit
Operating system	Linux 4.2
Compiler	g++ (GCC) 5.3.0
Compiler flags	-std=c++11 -O2 -fomit-frame-pointer

Chapter 11

The choice of ϵ

We have two data structures that give an approximate answer to range mode queries. One of which is 3-approximate, the other one being $(1+\epsilon)$-approximate. ϵ is tradeoff parameter between space/query time and guarantee. To determine which ϵ to use in our experiments, we run some tests to document and reason our choice of ϵ. Tweaking on ϵ will affect run time, space consumption and guarantee. A high ϵ will run the query faster, and a lower ϵ will use a bit more space. Additionally, when we have a low ϵ, the range mode query will be guaranteed to return a value which frequency is to closer to the frequency of the mode.

The test includes both various choices of ϵ as well as the 3-approximate data structure, in an attempt to properly compare the two in later use cases. We want the tests to show us the following:

- How often is the mode returned the actual mode of the range?
- What is the deviation from the actual mode in terms of frequency?

We run the data on the same data sizes as above on uniformly random data. First, we look on the construction and query time with different values of ϵ. This will help us see how the data structure scales based on changes in ϵ, and help us determine if there is a trade off we have to consider. Clearly, we want the best possible guarantee, but if it hurts performance too much, it might not be feasible. This is the trade off we are balancing here.

11.1 Construction

The construction time in theory doesn't depend on ϵ. However, with a smaller ϵ we store more data points which in practice requires more memory allocations. Our test on construction time can be seen in Figure 11.1.

As we can see in our test of construction time for various choices of ϵ, Figure 11.1, the construction time is a bit worse when we have a smaller ϵ. The difference is more visible with more data. With an data size of 512 KB we see a difference of 7% between $\epsilon = .35$ and $\epsilon = .05$. This is as we expected for our implementation. If the implementation did the allocation in one sweep

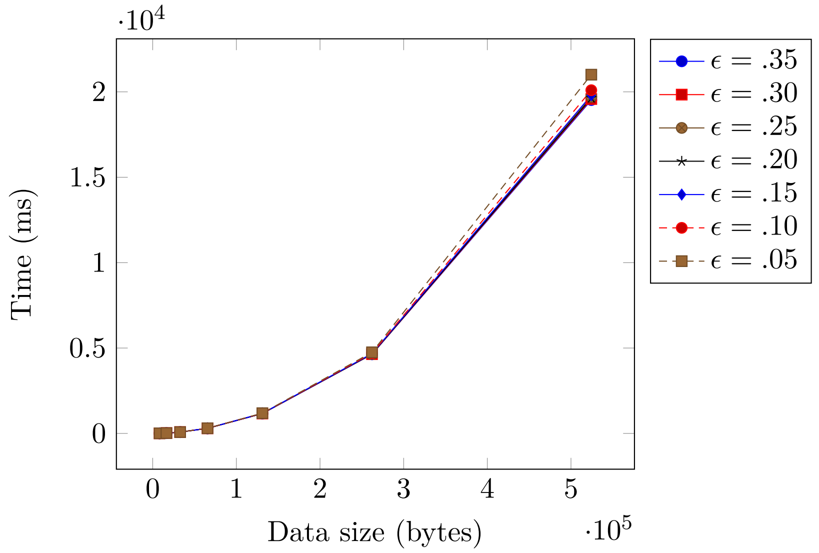

Figure 11.1: Construction time for different choices of ϵ

instead of multiple allocations, this difference could likely be much smaller. As we know how many elements we are going to store, this would be a possibility, but has the liability that the operation system needs to find a big consecutive chunk of memory for us.

The data structure construction is a one-time thing, and with the tweak to the implementation described above, the construction time seems insignificant for the choice of ϵ in this case.

11.2 Query Time

When discussing query time, ϵ seems more influencing. The hypothesis for using this data structure is that there will be a lot of queries, so many queries that make an exact range mode data structure infeasible. The choice of ϵ is very important for performance critical applications like these. The theory states that ϵ has a significant impact on the running speed of the query algorithm. A smaller ϵ requires more modes saved in the data structure, thus the query algorithm might need to look through more values to answer a query.

As seen in Figure 11.2, our implementation fits the theory very well here. For a larger ϵ, the time to answer a query on average is much faster for any data size. For a data size of 512 KB the data structure with $\epsilon = .05$ is 43.8% slower than $\epsilon = .35$, which obviously is significant if performance is critical for the application. With a high epsilon you will be able to carry out a lot more queries a second than with lower epsilon. This, of course, comes at the cost of a worse approximation of the actual mode.

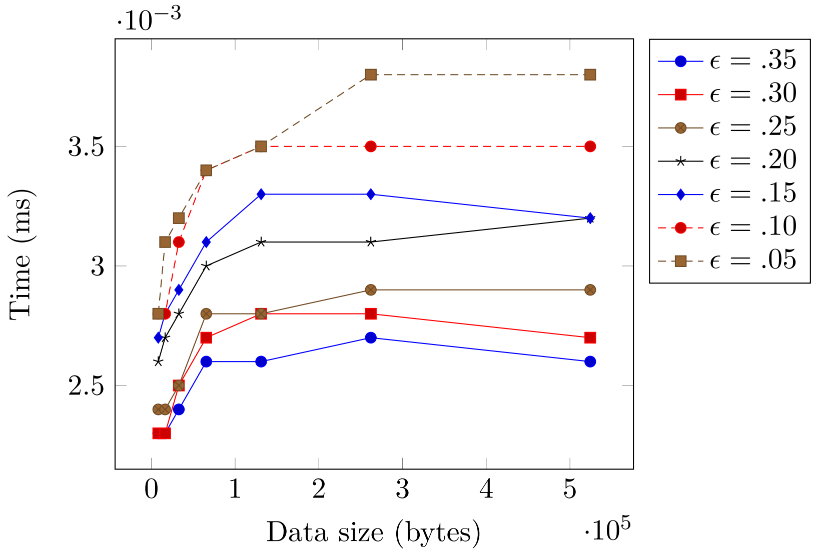

Figure 11.2: Avg. query time for various choices of ϵ

11.3 Correctness

The factor to consider is how the approximation scales depending on ϵ. We have a theoretical guarantee based on ϵ, but the data structure might perform way better than the lower-bound guarentee. We examine two aspects, how often does the data structure return the correct mode, and how much is the frequency of the approximation off compared to the frequency of the actual mode.

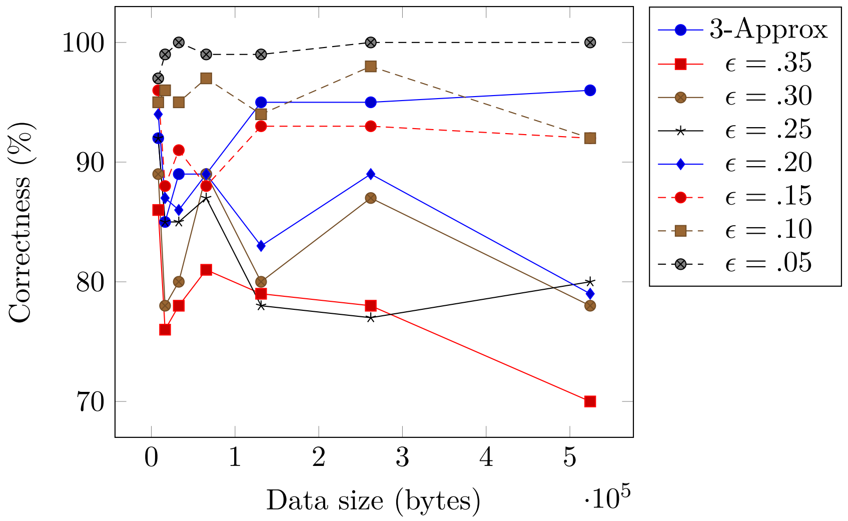

Figure 11.3: Percentage of how often the actual mode is returned for various ϵ

In Figure 11.3 we see the percentage that the data structure returns the correct mode. We can see that the ϵ has a huge impact on the correctness. A ϵ of 0.35 returns the correct mode around 75% of the time, whereas an ϵ of 0.05 is very precise and returns the correct mode about 98% of the time. We added the

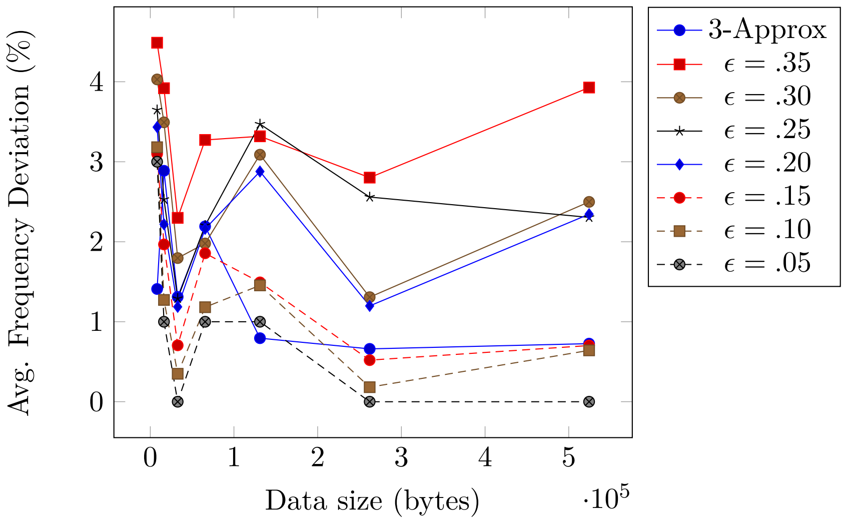

Figure 11.4: Avg. deviation of returned frequency from frequency of mode in percentage

3-approximate data structure to have a reference point to another approximate data structure. We can see that 3-approx steadily returns the correct mode above 90% of the time, which is in the high-end of the spectrum in terms of returning the correct mode compared to the ϵ-values we have tested with.

Next interesting aspect is, what the average deviation from the frequency of the correct mode is for these approximate data structures. The results of that experiment can be seen in Figure 11.4. The figure shows the average deviation in percentage of the frequency of the returned mode compared to the frequency of the actual mode of the range query. We see that the average deviation for $\epsilon = 0.35$ is about $3 - 4\%$ of the frequency of the actual mode. With $\epsilon 0.05$ the average deviation is very close to 0%. This seems like a somewhat huge gap, and can be a critical point when choosing the desired ϵ. In comparison, the other approximation data structure, 3-Approx, has an average deviation is just below 1%.

11.4 Combining the factors

We have looked on three different factors, construction time, query time, and correctness. We argued for ϵ not having a huge impact on the construction time. Query time and correctness have a significant impact, however, to properly be able to compare the other approximate data structure with the $1 + \epsilon$-approximate data structure, we have decided to mainly look on the correctness aspect. As seen in Figure 11.3 and Figure 11.4, 3-approx is very close to $\epsilon = 0.15$ in regard to correctness and percent-wise deviation of frequency.

We will thus use $\epsilon = 0.15$ in our experiments going forward to have a proper reference point when comparing the two approximate data structures.

Chapter 12

Exact data structures: Query and Construction

If we need the actual mode of a range, we will need to use either naive, data structure A or data structure B. In theory the linear-scan naive data-structure does all the work in the query algorithm and thus doesn't require more space than the input data. Beside the naive data structure we have a naive data structure with rank-space-reduction, which does a small amount of work in the construction. Contrary to the naive approach we have the data structures A and B, where we need to pre-compute data to make the querying faster. Naturally, these two data structures are to be used in different scenarios. We look on two parameters. Firstly, it is important to find out how much the pre-compute time affects the scenario overall; if a lot of queries are needed, the pre-compute time is minimal compared to the query time. Secondly, it is important how well the two query algorithms compare to each other; if the expected query range is low, a linear scan might not be bad. We try to analyse the data structures to define where the sweet spot of the trade offs is, and propose when one data structure is beneficial compared to the other.

12.1 Query time

12.1.1 Naive

The query time of the naive approach is relatively slow compared to the other data structures. In theory the Naive method should scale linear with the range of the query. The queries used for the experiments is linear compared to the size so the naive method should scale linearly compared to the data size in our experiments. If we look at Figure 12.1 we see that naive is in fact scaling linear in practice, but it is apparent that the data distribution affects the query time quite a lot.

Theoretically, the naive method should scale similar no matter what kind of data we are using but in practice the naive method is significantly slower when using Δ_{linear} data compared to $\Delta_{\sqrt{n}}$ and Δ_{constant} data. Thus, when the number of unique elements increase, the naive query time increases as well.

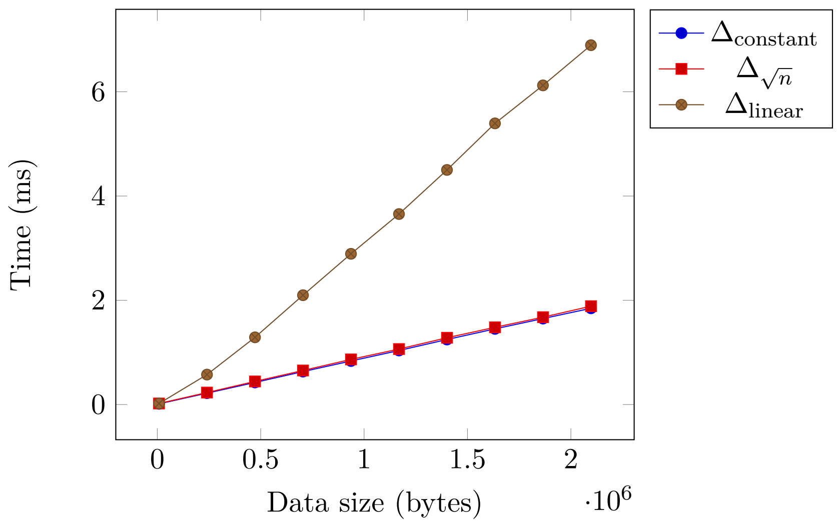

Figure 12.1: Query time of Naive

The naive implementation uses a hash map to store the count of each unique element in the range. Every time we encounter a new element we add a new entry to the has map. Otherwise we simply increment the entry associate with the value. The more unique values we have the more insertions we perform on the hash map, and the more space the hash map uses. Based on the data in Figure 12.1 it is safe to assume that the insertion cost is higher than the update cost, which result in a higher constant, but still gives us a linear scaling.

The hash map we use is the unordered_map from the C++ Standard template library[1]. It dynamically allocates memory, and because we don't know how many unique values there are in the range, we cannot allocate all the space we need at once. During the query the hash map might need to allocate more memory, and when unordered_map needs more memory, it allocates a new large piece of memory and copy all the elements to the new place. If this happens a lot the query will be slow, because you copy the same elements multiple times. This is also a factor that increases the query time of Δ_{linear} data compared to the $\Delta_{\sqrt{n}}$ and Δ_{constant} , because the space usage of unordered_map increases, when the number of unique elements increases.

When looking at Figure 12.1 we see that query time on Δ_{constant} and $\Delta_{\sqrt{n}}$ data distribution are very close and in fact cannot be visually distinguished in the graph. If we instead look at the actual numbers in appendix section A.1 of the experiments, we can see that $\Delta_{\sqrt{n}}$ is a tiny bit slower than Δ_{constant} .

Rank space reduction is a method to avoid the excessive use of memory that the C++ hash map implementation has. Rank space reduction is described in Section 3.6. If rank space reduction is used, we can easily know the number of unique values, and we can replace hash map with an array, allocate it all at the start, and we don't have to perform any insertions. The query time is then

[1]`http://en.cppreference.com/w/cpp/container/unordered_map`

bound to the number of unique elements and the length of the range,

$$\mathrm{O}\left(\Delta + (j - i)\right)$$

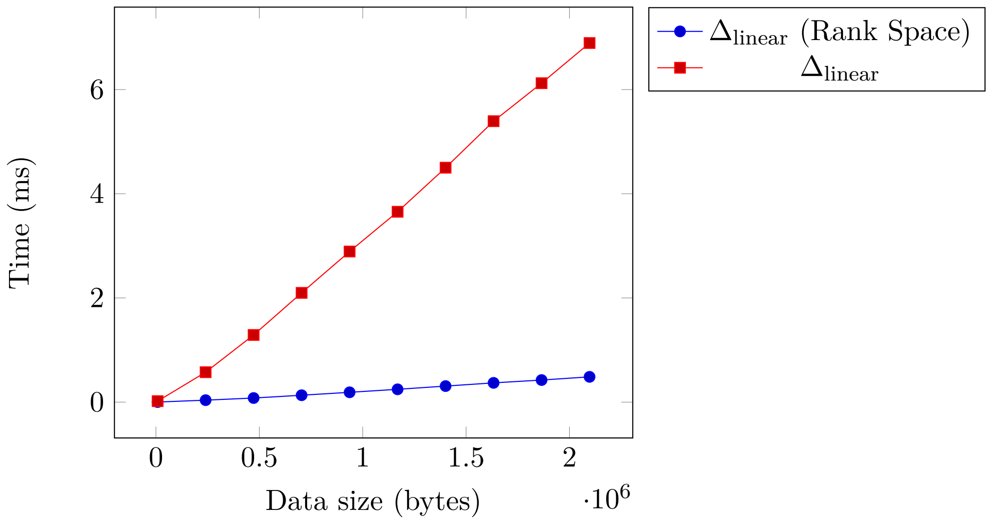

Figure 12.2: Query time of Naive and Naive with Rank-Space-Reduction

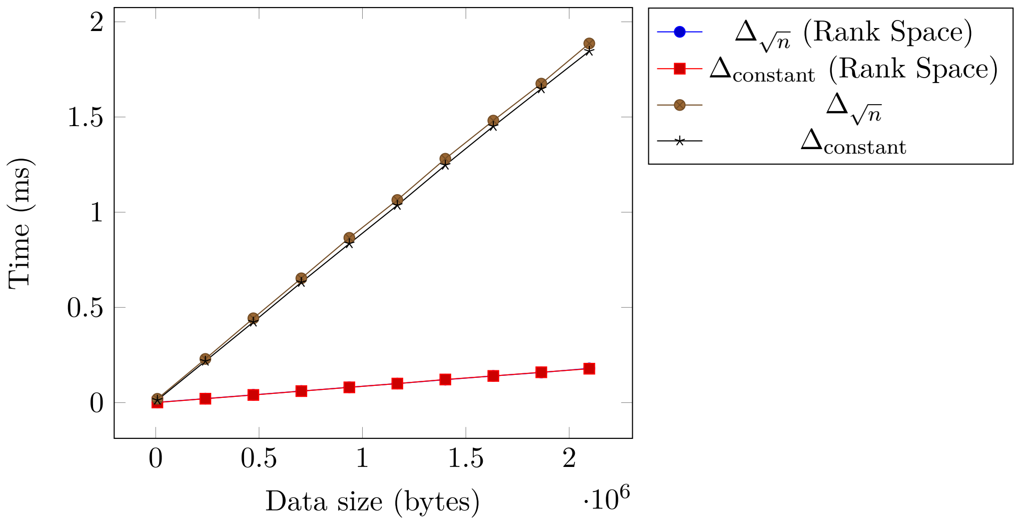

Figure 12.3: Query time of Naive and Rank-Space-Reduction for Δ_{constant} and $\Delta_{\sqrt{n}}$

As we can see in Figure 12.2 and Figure 12.3 the naive method with rank space reduction outperforms the hash map by a significant factor. It is also worth noticing that using the rank space reduction did not change the scaling of the query time. It is still linear which is nice since we only changed a single implementation detail, while the algorithm still is the same, thus having the same complexity. The way naive with rank-space-reduction is implemented the query time is $\mathrm{O}\left((j - i) + \Delta\right)$, because we use an array of size Δ to count the frequencies, and then do a scan of the array to find the largest frequency. As the number of unique elements is lower than the average range length in the experiment, the run time is $\mathrm{O}(j-i)$. However, at this point it would be better to use data structure B, which have a query time of $\mathrm{O}(\Delta)$ without the range factor, so data structure B would always outperform naive with rank space reduction of the naive algorithm.

The naive algorithm without rank space reduction does still have a use case. If you have a range query, where the length of the range is below $\sqrt{n}$ and Δ, the naive implementation is in theory faster than both data structure A and B.

12.1.2 Data structure A

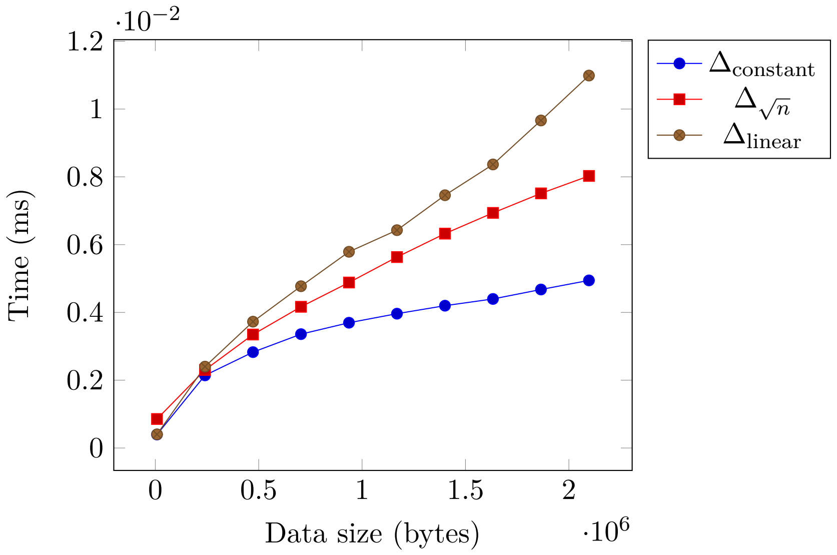

Figure 12.4: Query time of data structure A

The theoretical query time of data structure A is $\mathrm{O}(\sqrt{n})$. Figure 12.4 shows the query time of data structure A on the three different data distributions. The graph seems to resemble a $\sqrt{n}$ like curve, but with different constant factors. So the practical query time fits the theory, but according to the theory the type of data distribution shouldn't matter. However, we can see that the more unique values in the data, the slower query time we are are getting. This is not obvious from the theory, because the query time only depends on the input size n, and not the number of unique elements Δ. The results on Figure 12.4 clearly shows, that when the number of unique values increases, so does the query time. We need to figure out what happens when the number of unique element increases. As the test data is uniformly distributed the expected frequency of the mode of any range is $(j-i)/\Delta$. When the number of unique elements increases the expected frequency gets lower.

Remember that in the query algorithm of data structure A, we split the query range into three parts: prefix, span and suffix. As described in Chapter 4 for every unique element i in the prefix and suffix we perform at operating

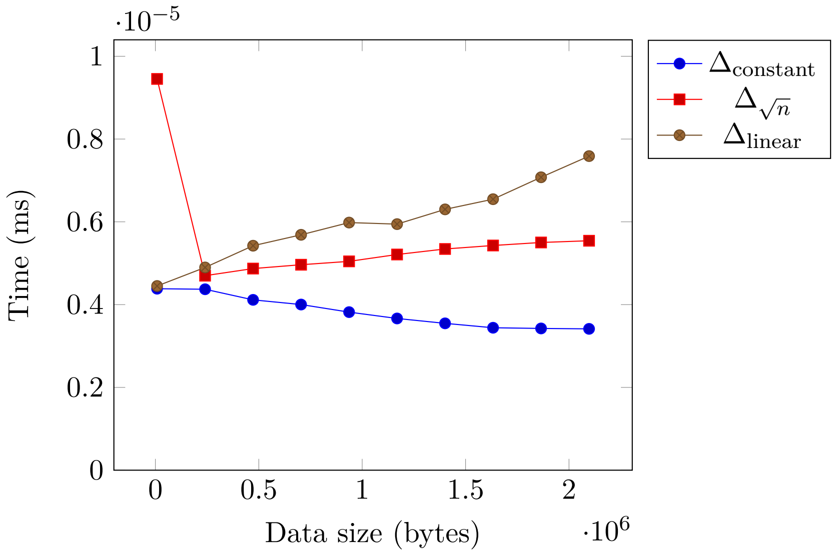

Figure 12.5: Query time of data structure A divided by $\sqrt{n}$

to see, if the range contains at least as many i elements as the frequency of the span. If that is the case we count the number of i elements in the range using the Q_i table. The important thing is that every time we count an extra i element in the range there is one less unique element, because that must be an extra i in the prefix or suffix.

This means that if there are a high amount of unique values, there will be more checks of the frequency and less counting of elements and visa verse. So, why is this slower than the counting operation? They are both constant operations, so theoretically the query time is the same, but just because they both are constant time the constant can be different in practice. When the number of unique elements increases, we perform more lookups into Q_i where i is the unique value for every element in the prefix or suffix, the i element changes so we have to fetch a new table from memory.

This is where the theory and practice differ. In the theoretical model it doesn't matter where data is located in the memory. Everything can accessed in constant time, in practise, however, it matters where data is located in memory. Data that are closely located will be faster to read, because the processor fetches a block of memory into the cache. Fetching data from the cache is much faster than fetching from memory, so if you only need a single value from the memory block, you will use more time, than if you use multiple values from the same block.

Therefore when we have a lot of unique element we need to fetch a lot of different tables from the memory and perform a single lookup. Since we only need a single value from each table, it will lead to the processor spending a lot of time waiting for memory. On the other hand, if we have a few unique elements, we have to count the element in a single table. Since we need multiple values from the same table this will be faster because we have to read fewer blocks from memory.

12.1.3 Data structure B

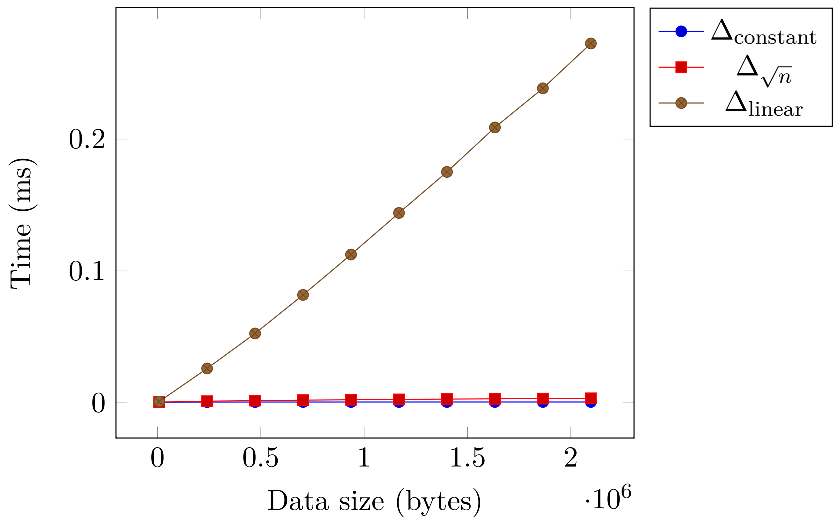

Figure 12.6: Query time of data structure B

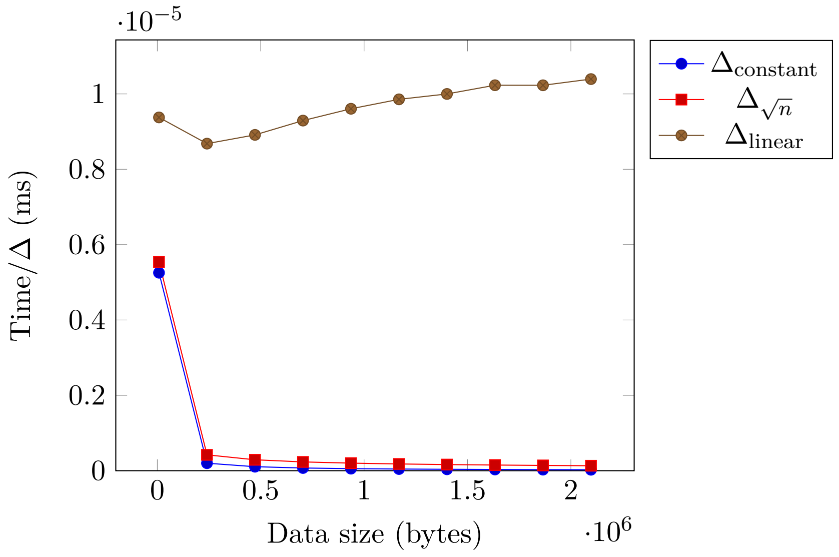

Figure 12.7: Query time of data structure B divided by Δ

Data structure B perform range mode queries in time $O(\Delta)$, which means that the query time scales linearly with the number of unique values. The different data distribution are based upon the number of unique element, so the query time of Data structure B on Δ_{linear} should be $O(n)$, the query time of $\Delta_{\sqrt{n}}$ should be $O(\sqrt{n})$, and Δ_{constant} should be $O(1)$. On Figure 12.6 we see the results of the query time experiments for data structure B.

From Figure 12.6 we can see that Δ_{linear} results in a linear query time, which fits the theory perfectly. But we cannot differentiate $\Delta_{\sqrt{n}}$ and Δ_{constant} , because they seem to be right on top of each other and cannot be visually distinguish. Since the query time should be linear compared to the number of unique values we can divide the query times the result should be a constant.

In Figure 12.7 we see the query time divide by the number of unique values. If we ignore the first data point of Δ_{constant} and $\Delta_{\sqrt{n}}$ the results is constant query time per unique elements as expected. The Δ_{linear} data distribution seems to grow a little bit but still pretty much constant.

12.1.4 A vs B

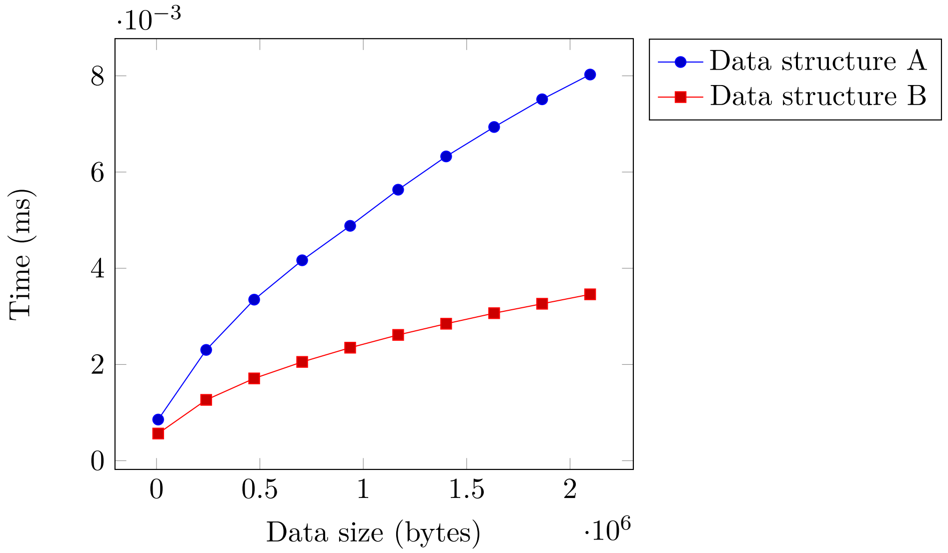

Figure 12.8: Query time of A and B on $\Delta_{\sqrt{n}}$ data distribution

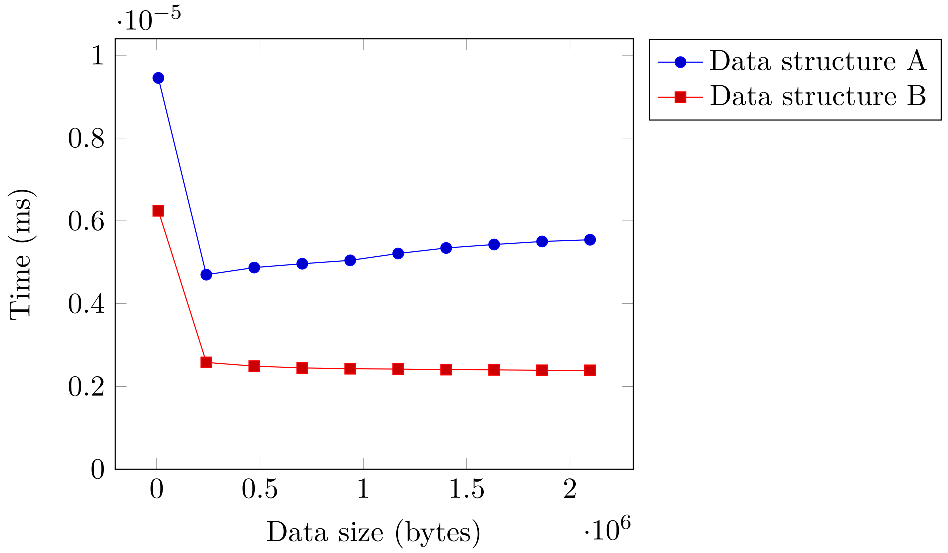

Figure 12.9: Query time of A and B on $\Delta_{\sqrt{n}}$ divided by $\sqrt{n}$

It is quite interesting to compare data structure A's and data structure B's query time, because we can discuss the combination of the two data structure from Chapter 6. The data structure A+B split the data input into A and B dependent on the total frequency of a given value. If value i has a frequency higher than $\sqrt{nw}$ we use data structure B and if the frequency is lower, we use data structure A. It is interesting to compare the query performance for the $\Delta_{\sqrt{n}}$ data distribution.

If we look at both data structure A and B for $\Delta_{\sqrt{n}}$, it is expected to have the same complexity. The complexity of data structure A does not depend on the number of unique elements and is always $\mathrm{O}\left(\sqrt{n}\right)$. Data structure B's query time is $\mathrm{O}\left(\Delta\right)$ and with $\Delta_{\sqrt{n}}$ we have $\Delta = \sqrt{n}$, which gives us the query time $\mathrm{O}\left(\sqrt{n}\right)$ for data structure A and B.

If we look at Figure 12.8 we can see that the query time of both A and B follow a $\sqrt{x}$ like curve. In Figure 12.9 we have divided the query time by $\sqrt{n}$, which should give us a constant line. If we discard the first point they are almost constant. The query time of data structure A is a bit more than a factor two of the query time for data structure B. The query algorithm for data structure B is fairly simple compared to data structure A, so it seems logical that the constant factor is a higher for data structure A than data structure B.

12.2 Construction time

12.2.1 Naive

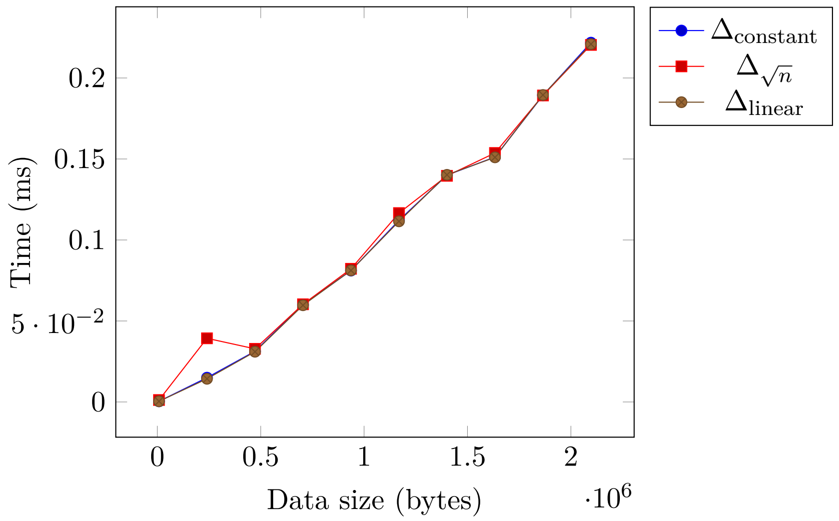

Figure 12.10: Construction time of Naive

In theory the naive implementation has nothing to do when initialising and thus have constant time construction. However, in practice we implemented it differently as we copy the input data to a new memory location. The main advantage of doing this is that if the code responsible for the memory storing the input data changes said data, the data structure will no longer work on the data as was, but instead work on the updated data. The disadvantage is that we have to copy the data, which means the construction time of the naive implementation is $O(n)$. Making a copy of the data might not be desired in practice, but it depends on the situation. We can see the construction time in Figure 12.10.

As argued, the only thing the construction of naive does is copying the input data, thus the amount of unique values is insignificant. All of the three data distributions run in the same time as they all copy the same amount of data, but as we can see, the graph is not perfectly linear. This might be due to certain sizes being aligned better in memory than other sizes, making the copying is faster. Overall the construction time scales linearly with the data size as expected.

The rank-space-reduced naive performs a rank-space-reduction when constructing the data structure. Performing a rank-space-reduction have linear complexity, since every value needs to be transformed to the rank-spaced-reduced variant. The construction time should still be linear, but it should take significant longer time since doing a rank-space-reduction requires more complex operations than a simple copy.

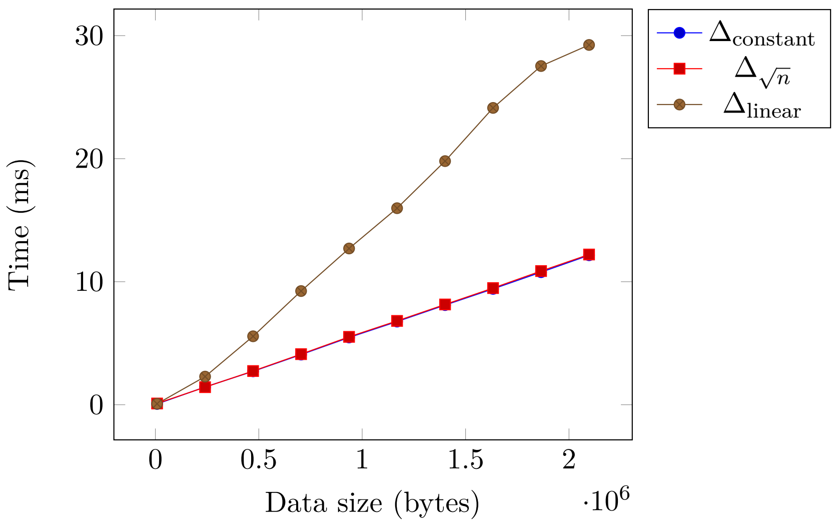

Figure 12.11: Construction time of Naive with Rank-Space-Reduction

In Figure 12.11 we see the construction time of naive with rank-space-reduction. As expected the construction time is linear, and the rank-space-reduction adds an significant factor to the construction time. This makes a lot of sense since copying data in memory is very efficient, but the rank space reduction actually have to performs some modifications and comparisons. When performing a rank-space-reduction we insert every unique element into a hash map. So the number of insertions into the hash map is the same as the number of unique elements in the data. As we can see on Figure 12.11 when the number of unique elements scales linear the construction time increases significantly. As with the query of naive we can conclude that the cost of an insert operation on the hash map is quite high and it dominates the construction time.

The last data point for Δ_{linear} on Figure 12.11 is a bit lower in an otherwise near-perfect linear line. The reason for this could be that the data size exactly is 2 megabytes. 2 megabytes is 2^20 so it is a power of 2 and numbers of powers of 2 tends to be better aligned in memory, thus the access time would be smaller. Also, the insertions into the hash map might avoid allocating too many memory pages, if the data fits better.

12.2.2 Data Structure A

In Chapter 4 we demonstrated that the construction time of data structure A is $\mathrm{O}\left(n^{3/2}\right)$. In Figure 12.12 we see the construction time of Data Structure A in practice on the three different data distributions. Even though the construction time is a bit different between the data distribution the graphs have the same development, and only differ by a constant factor. It is difficult to conclude that the construction time is $\mathrm{O}\left(n^{3/2}\right)$ from Figure 12.12 alone, but in Figure 12.13 we have divided the query time with $n^{3/2}$ where n is the size of the input data. This should give a constant graph for the three distributions, if the construction time is $\mathrm{O}\left(n^{3/2}\right)$. The data in Figure 12.13 shows that $\Delta_{\sqrt{n}}$ runs in a $n^{3/2}$ curve with the exception of size 8kB. The data distribution Δ_{constant} runs faster than $n^{3/2}$.

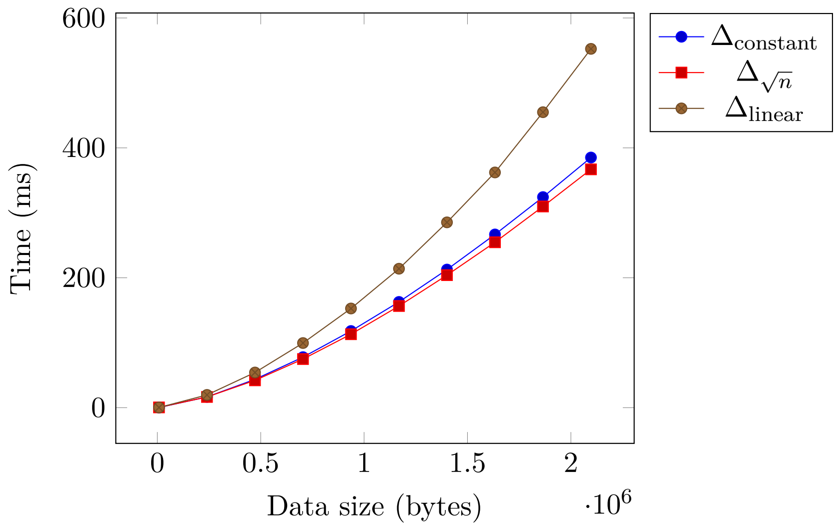

Figure 12.12: Construction time of data structure A

12.2.3 Data structure B

The construction time of data structure B is expected to be linear compared to the size. All the data distribution should give the same results. When we look at Figure 12.14 we can see that the type of data distribution does affect the construction time quite a lot. Even though the data distribution have different construction time, all three of them is still growing linearly.

Data structure B performs a Rank-Space-Reduction on the input data before actually constructing the data structure. If we compare Figure 12.14 with

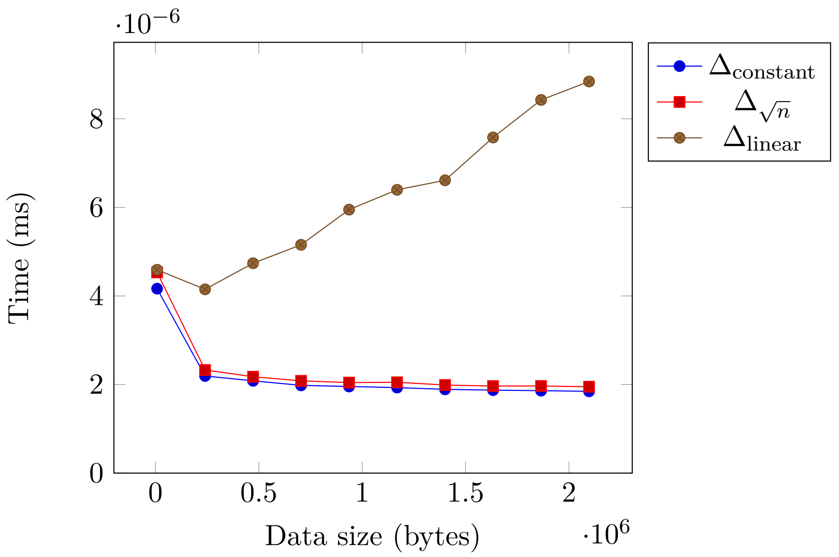

Figure 12.13: Construction time of data structure A divided by $n^{3/2}$

Figure 12.11 we see that they are very similar. The only difference between the naive with rank-space-reduction and data structure B is that B builds an extra table. If we look at Δ_{linear} with the 2 megabyte data size data structure B takes 31 milliseconds, and the naive with the rank-space-reduction takes 29 milliseconds. That is a difference of 2 milliseconds and since the only construction naive with rank-space-reduction does is the rank-space-reduction the rank-space-reduction takes approximately 90% of the construction time for data structure B.

The rank-space-reduction is also why Δ_{linear} have a much higher construction time than $\Delta_{\sqrt{n}}$ and Δ_{constant} . The construction time of data structure B for Δ_{linear} at data size of 2 megabytes even is bit lower than expected just like in figure 12.11.

12.3 Construction time vs Query time

The cost of faster query times is the construction time. In this section we compare this tradeoff between query times and construction time, and reason about when it is better to use a data structure, than the naive approach. Specifically, we will see how many naive queries we can perform, before the construction time and query time of a specific data structure is faster.

This can be formulated as the equation,

$$\frac{\textit{construction time}}{\textit{naive query time} - \textit{query time}}$$

Figure 12.15 shows the number of naive queries needed for it to be more efficient to use a rank-space-reduced naive approach. The number of queries required is constant compared to the input size. This is because the construction time of the rank-space-reduction is linear, and the query time for naive is also

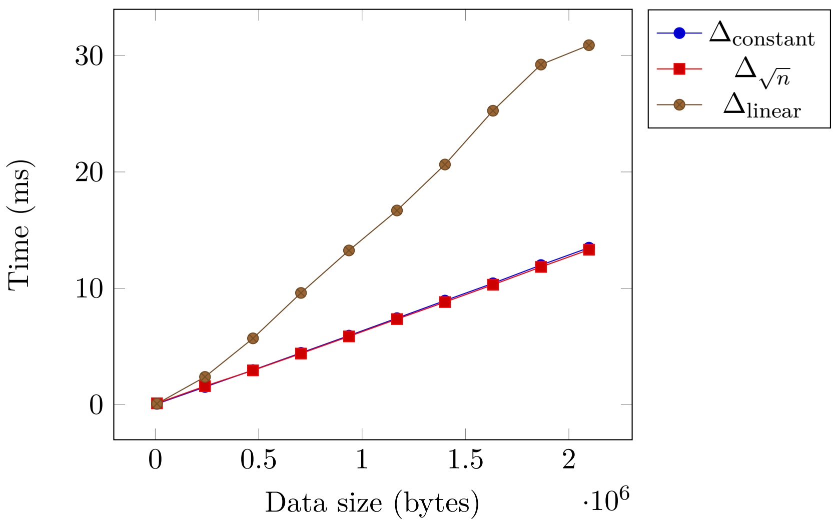

Figure 12.14: Construction time of data structure B

linear. When you divide to linear functions you get a constant. For $\Delta_{\sqrt{n}}$ and Δ_{constant} if you perform more the 7 queries, it would be more efficient to perform a rank-space-reduction first. For Δ_{linear} you only need to perform 5 queries.

The number of unique elements affect the number of queries needed. This is because the query time of naive increase significantly with number of unique element increase and thus the number of queries required will be smaller.

In Figure 12.15 we also see the number of naive queries required for data structure B to become more efficient. As we can see the queries required for data structure B and naive with rank-space-reduction are very similar. As stated when analysing the construction time of data structure B, approximately 90% of the construction time is spend during the rank-space-reduction. That is why the number of queries needed is about the same as for the naive with rank-space-reduction.

The construction time of data structure A is $\mathrm{O}\left(n^{3/2}\right)$. This means that when we divide the construction time with the linear query time of naive we get an $\mathrm{O}(\sqrt{n})$ graph because $n^{3/2}/n = n^{1/2} = \sqrt{n}$. So in contrast to data structure A the data size actually matter when determining have many queries are needed.

As stated in Chapter 10 the average range length of the queries is 1/3 of the size. This means that Figure 12.15 and 12.16 show the number of naive queries needed, if the average query size is $1/3n$. But what if your queries do not have an expected average size of $1/3n$? Since the query time of naive queries are linear, we just multiply the query numbers with 3 and divide by the expected average query size.

An example: The expected average range size is $1/4n$, the data size is 1 megabyte, and have $\sqrt{n}$ number of unique elements. How many naive quires do you have to perform before A or B would be feasible instead? For data structure A you need to perform 7 queries but we have to divide by 3 because the graphs data had a range length $1/3n$ and multiply by 4 because the expected range

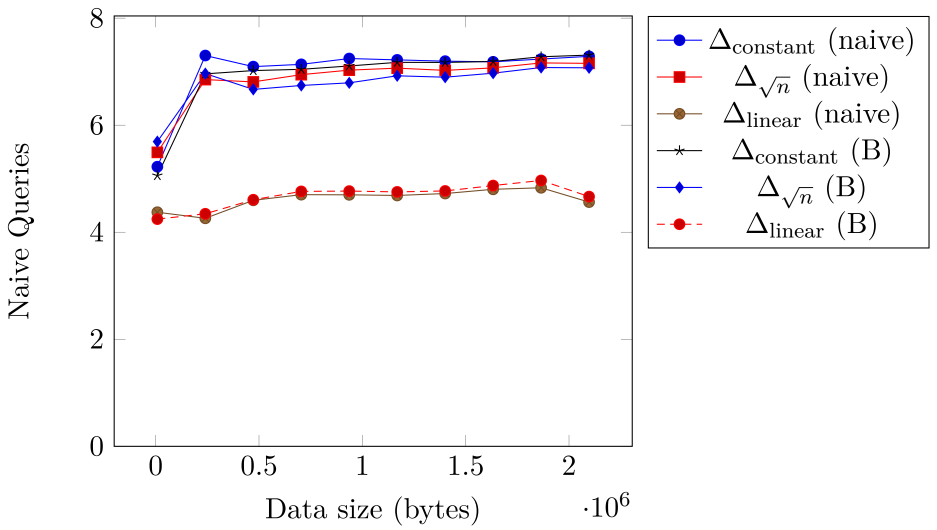

Figure 12.15: Data structure B & Naive with Rank-space-reduction: Naive queries per construction

length in the example is $1/4n$.

Data structure A:

$$130 * \frac{4}{3} = 174 queries$$

Data structure B:

$$7 * \frac{4}{3} = 10 queries$$

12.4 Summary

When choosing which exact data structure to we need to consider 3 things. How many queries will be performed? What is the expected length of the range queries? How many unique values are in the data set? In our experiments we see that if number of unique elements is below $2\sqrt{n}$ data structure B has the lowest query time. If the number of unique elements is higher than $2\sqrt{n}$ it's better to use data structure A.

Compared to data structure A and data structure B the naive query algorithm does have a construction time. So if you perform a few range mode queries it might be better to not construct a data structure and just use the naive algorithm. However our experiments show that the number of ranges have to be really low before the data structures begin to be faster. Because data structure B has a linear construction our results are that if you perform more than 7 range mode queries it is more efficient to construct data structure B and perform the queries as long as the expected range length greater than or equal $1/3n$. If the range length is lower the $1/3n$ the number will of course be higher.

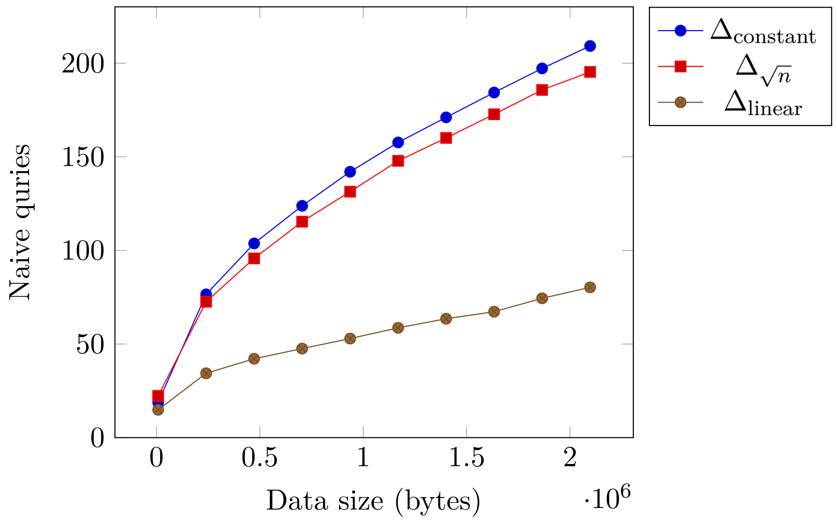

Figure 12.16: Data structure A: Naive queries per construction

Data structure B is linear construction time. So if the expected range query length is linear the number of queries needed before data structure B is more efficient is constant number and does not depend on the size of the data. Data structure A construction time is $\mathrm{O}\left(n^{3/2}\right)$ so if the expected range query length is linear the number of quires needed is not a constant number but is dependent on the data size.

If the query length is lower than the number of unique elements and $\sqrt{n}$ then the naive algorithm is faster the both data structure A and data structure B since data structures A query time is $\mathrm{O}\left(\sqrt{n}\right)$ and data structure B's query time is $\mathrm{O}\left(\Delta\right)$.

Based on our experiments we can see that a combination of data structure A and B makes a lot sense. In data structure A+B you use data structure A for all element with a frequency lower than $\sqrt{n}$ and data structure B for all elements with a frequency higher than $\sqrt{n}$. Since the minimum frequency of any element in data structure B is $\sqrt{n}$ the maximum amount of of element is is $\sqrt{n}$. Based on our result you can change the split factor to all frequencies below $\frac{1}{2}\sqrt{n}$.

Besides the split you can add a condition to use the naive approach if the query range length is lower than the number of unique element and $\sqrt{n}$. Like that you can combine all three exact data structures and get a single data structure witch can handle any combination of range length, number of unique element and data size in a reasonable time.

Chapter 13

Approximate Data structures: Query and Construction

On the other hand, if an approximate answer within a tolerance factor is sufficient, we have two data structures to consider, both of which in theory should perform significantly better than the accurate ones: 3-Approx described in Chapter 7 and $1+\epsilon$-approx described in Chapter 8. Both of these data structures require some pre-computing, and both have a very fast theoretical query algorithm. We look into how the query algorithms compare to each other as well as look at how the pre-compute times compare. We finally give a heuristic for when to use one over the other. While these algorithms can be used for the same scenario, $1+\epsilon$-approx can adjust the level of estimation while 3-approx cannot, so there might be scenarios in which the only valid option is $1+\epsilon$-approx or one of the exact data structures. We will also look into these cases and compare the $1+\epsilon$-approx with the precise data-structures on the same parameters as we compared them with each other: expected number of queries and expected query range size.

The approximate data structures are $1+\epsilon$ and 3-Approx. Both of these are compared and discussed in this section for $\epsilon = 0.15$. The choice of this ϵ is accounted for in Chapter 11.

The expected query time of $1+\epsilon$ is $\mathrm{O}\left(\log\log_{1+\epsilon} n\right)$ and the expected query time of 3-Approx is $\mathrm{O}\left(n \log\log n\right)$. We expect 3-Approx to be faster than $1+\epsilon$, since the theoretical query time is better.

13.1 Query time

13.1.1 3-Approximate

The 3-approximate data structure has a constant theoretical query time. As discussed in Chapter 7, the implementation uses a $\mathrm{O}\left(\log\log n\right)$ LCA look up, which makes the query time of the implementation a total of $\mathrm{O}\left(\log\log n\right)$. The graph of our query time experiments for 3-approx can be seen in Figure 13.1. We can see the query time is less than linear in respect to input size.

There are some points that are a bit hard to account for, mainly in $\Delta_{\sqrt{n}}$

data distribution. We had a hard time figuring out why there is a big difference between the different data distributions as the query time should be independent from number of unique values. There is a limited number of look ups and the number of unique values does not play a part in how the querying works.

There are a few factors that have a somewhat big impact on the querying time. One of these factors is the ranges chosen. If a range spans an entire LCA node, there will only be a single look up (apart from the actual LCA look up). However, if the queried range doesn't span an entire LCA node, we will have to look up three different frequencies and modes and compare them to find the 3-approximate mode of the range. This means there is up to a factor of 3 difference in the amount of look ups needed depending on the range.

Another factor that impacts the querying time is the LCA look up. As we don't have a constant-time LCA, we first look at the root and see if the range is contained in a single child. If it is, we recursively look for the LCA in that child. As the tree has a height of $\log \log n$, the height for an input size of 2 MB is 5.

The query times are blazingly fast, which makes a lot of sense implementation-wise, which is also why any work done by the operating system during execution of the tests have a bigger percentage-wise impact on the test results, but as we run the experiments three times with a lot of ranges, and the data from each experiments seem to match each other, we don't think this is too much of a problem.

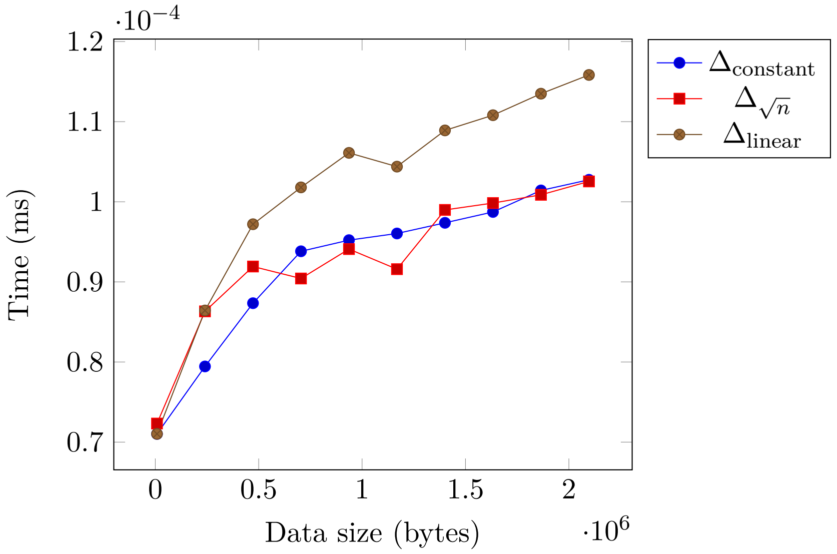

Figure 13.1: Query time of 3-approximate

From a perspective of theory and implementation we have a hard time understanding why the Δ_{linear} data has a clear tendency of being worse than the other two data distribution, but there are some things to consider. There is the hardware specific aspect of branch prediction. There is a lot of branches in the query algorithm including comparisons of modes, this could be one of the reasons for this trend. To further expand on this point, the final step in the implementation, where the mode is returned, has three different branches. First it checks if the mode is in the child pair span, otherwise it will check for either prefix or suffix and return the mode with the largest frequency. It is clear that the child pair span is more likely to contain the highest frequency compared to the frequency of the prefix and suffix. As the unique number of values in a data set increases, the probability that the child span contains the frequency decreases. The probability that the mode is in the child pair span is inversely correlated with the unique number of values. Therefore, when the data distribution is Δ_{linear} , the probability is higher for the prefix or suffix to be higher frequency than with Δ_{constant} , thus we might encounter more branch misprediction, which could be the cause of the 10% increase in query time for that data distribution. This, however, is fully dependent on the CPU, and we didn't pursue this thought. As to why $\Delta_{\sqrt{n}}$ isn't following the same tendency, the unique number of values might not be big enough to see an actual difference compared to Δ_{constant} . The data distribution $\Delta_{\sqrt{n}}$ has roughly 7 times more unique values than Δ_{constant} for a data set of 2 MB, whereas Δ_{linear} has roughly 70 times more unique values than $\Delta_{\sqrt{n}}$ for a data set of 2 MB. This is quite a big leap between the different data distributions we have chosen, which might be the reason the tendency is not clear from the graph.

13.1.2 $(1+\epsilon)$-Approximate

In the theory section of $(1+\epsilon)$-approximate data structure, we account for the query time of the data structure to be $\mathrm{O}\left(\log\log_{1+\epsilon} n\right)$. In Chapter 11 we experimented with different values of ϵ and we choose $\epsilon = 0.15$, which results in a theoretical query time of $\mathrm{O}\left(\log\log_{1.15}(n)\right)$.

Looking at Figure 13.2 we can see very fast query times on the first data point. After that we see a spike in performance which stabilizes itself. The growth seems to be much more than $\log\log n$ for $\Delta_{\sqrt{n}}$ and Δ_{constant} , but Δ_{linear} seems to follow the theory.

It is quite interesting that Δ_{constant} is the one performing worst, however quite easy to explain and follows from our implementation. The more uniformly unique values there are in our data, the longer it will take to find the next frequency that is a factor $1+\epsilon$ greater than the previous frequency, which is what gives us the $(1+\epsilon)$ approximation guarantee. This means that when there are more unique values, there are fewer elements to iterate in average when querying. Therefore, Δ_{linear} performs way better and $\Delta_{\sqrt{n}}$ and Δ_{constant} is about the same with Δ_{constant} being the worse.

The reason that the first data point performs this well could be that the entire data structure can be located in the cache and the total number of access to memory is minimal, however for the larger data sizes the operating system needs to access memory more.

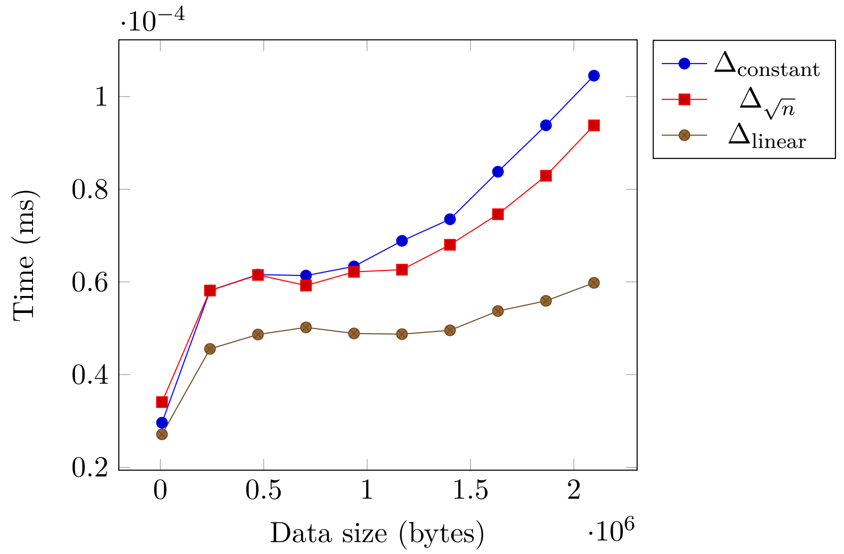

Figure 13.2: Query time of $(1+\epsilon)$-approximate with $\epsilon = 0.15$

13.2 Construction time

13.2.1 3-Approximate

The theoretical running time for the construction algorithm is $\mathrm{O}\left(n^{3/2}\right)$. As we can see in Figure 13.3, the implementation seems to match this bound. It is quite clear that the construction time is worse when the number of unique values is large. This tendency makes a lot of sense as we have talked about throughout the discussion. A major reason for this is the rank-space-reduction, which creates a huge gap in performance between $\Delta_{\sqrt{n}}$ and Δ_{linear} but barely a noticeable difference between Δ_{constant} and $\Delta_{\sqrt{n}}$.

The implementation allocates an array of size Δ after the rank-space reduction, where Δ is the number of unique values. This array is used whenever a frequency is to be calculated, thus used a lot of times throughout the construction, and is the main factor affecting the performance. Since the array is of static size Δ based on input data, we iterate all of those entries when calculating each of the frequencies.

The total time of the construction is still quite fast. For an input size of 2 MB with about 52.000 unique values, it takes a total of ≈ 27 seconds to fully construct the data structure.

13.2.2 $(1+\epsilon)$-Approximate

The theory suggests the $(1+\epsilon)$-Approximate data structure has a construction time of $\mathrm{O}\left(n^2\right)$. Our experiments for this data structure is plotted in figure 13.4. We can see that the graph seems to match the theory. The data distribution with most unique values is a bit slower constructing the data structure. One reason is that we use rank-space-reduction, which we earlier discussed is a reason for linear to be much slower than $\Delta_{\sqrt{n}}$ and Δ_{constant} . this seems to be

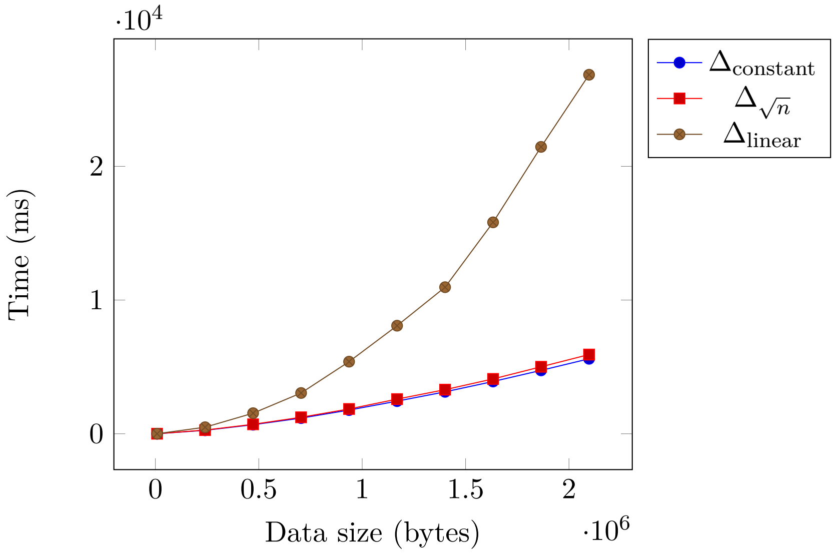

Figure 13.3: Construction time of 3-approximate

the main reason we see this gap between the linear data distribution compared to the other two data distributions.

Another reason for this could be that in construction we iterate the data until the frequency is above a certain constant. The constant is irrelevant here as this is the same for all data distributions in our experiments. As we have uniform random data distribution and we have more unique values, the probability to reach a frequency above the constant limit early in the iteration is lower. If we, on the other hand, have a higher probability to reach a frequency early in our iteration, the construction time should be lower, which is why we see Δ_{constant} and $\Delta_{\sqrt{n}}$ data distributions to be very close.

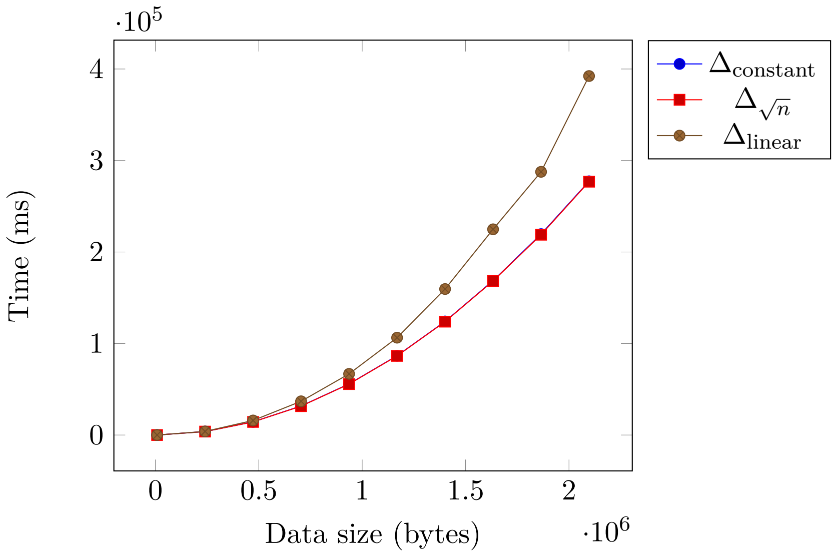

Figure 13.4: Construction time of $(1 + \epsilon)$-approximate with $\epsilon = 0.15$

The difference between the construction time of Δ_{constant} and $\Delta_{\sqrt{n}}$ is very minimal. They are basically running at almost the same speed. We think the reason for this is that the difference in unique number of values simply isn't big enough between the two. With 2 megabyte data there is about 7 times as much unique values in $\Delta_{\sqrt{n}}$ compared to Δ_{constant} , but about 70 times more unique values between Δ_{linear} and $\Delta_{\sqrt{n}}$.

13.3 Correctness

We briefly discussed correctness in Chapter 11 when decided on the ϵ to use in the experiments. Here we discuss the results for the two approximate data structures on uniformly random data.

13.3.1 3-Approx

In figure 13.5 we see the correctness of 3-approx in terms of how often the returned mode is the actual mode of a range. The data structure gradually performs better as the data size increases and as the number of unique values increases.

Regarding data size, as data size increases, the average query range size increases linearly. The size of children is defined by $n^(1/2^i)$ for level i in the tree structure, where root is level 0. This means, the sizes of the children, and thus the prefix and suffix sizes, grows less than linear. Logically it follows that the mid-range of children-pairs will be bigger for larger data sizes of n elements. As the mid-range is bigger, it is more probable that the mode is in that range.

We see the data structure's correctness on Δ_{linear} is way better than on Δ_{constant} and $\Delta_{\sqrt{n}}$ a reason for this could be the same as the reasoning as to why Δ_{linear} has a worse query time on average; the probability for either the prefix or suffix to be the mode is bigger than Δ_{constant} and $\Delta_{\sqrt{n}}$.

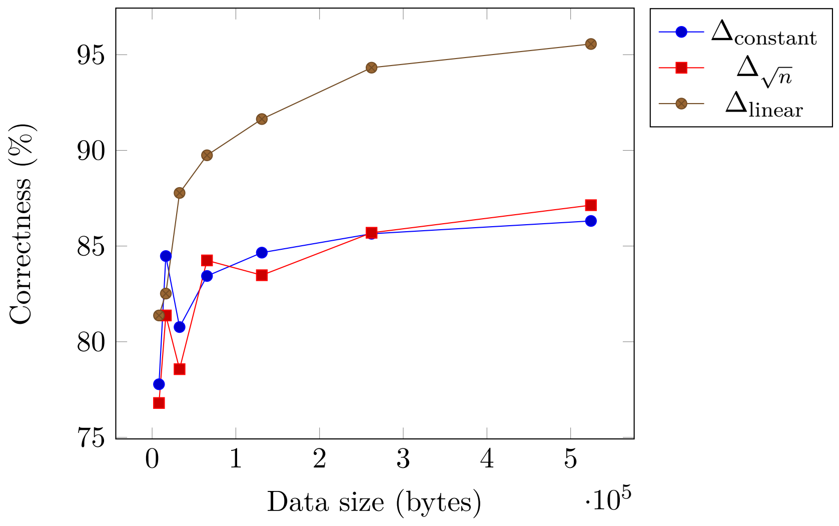

Figure 13.5: Percentage of how often 3-approx return the mode

Interestingly, the returned frequency is on average off by a larger percentile for Δ_{linear} than Δ_{constant} and $\Delta_{\sqrt{n}}$, which can be seen in Figure 13.6. As Δ_{linear} is returning the correct mode much more often, this must mean that Δ_{linear} is off by a larger margin, when the correct answer is not returned. This can be explained with the fact that the probability for returning a mode from the prefix or the suffix is bigger when Δ is large. So while the Δ_{linear} data distribution is often times right when returning a mode from prefix and suffix, the percentage it is off is bigger when the mode is wrongly returned from the prefix and the suffix, which makes a lot of sense when considering the expected frequency of the mode in prefix and suffix is less than the expected frequency of the mid range, as there are less values, or at least as many values.

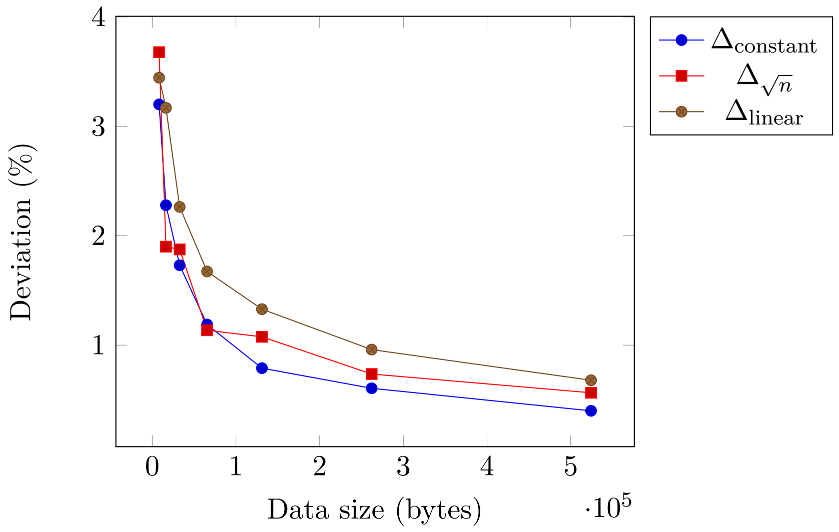

Figure 13.6: Avg. deviation of returned frequency compared to frequency of mode (for 3-approx in percentage)

13.3.2 $1 + \epsilon$

We have established our choice of $\epsilon = 0.15$. We have a guarantee that the frequency of the returned value will be at least $1/1.15$ of the frequency of the actual mode. Looking on the percentage that $\epsilon = 0.15$ gets the mode right on uniformly random data, Figure 13.7, we can see that the Δ_{linear} data distribution is returning the correct mode way more often than the two other data distributions. On the data distribution Δ_{linear} the correct mode is returned about 90% of the time for 2 MB data, whereas Δ_{constant} and $\Delta_{\sqrt{n}}$ are returning the correct mode about 73% of the time. Meanwhile it is quite interesting that the average deviation is of $\epsilon = 0.15$ is bigger for Δ_{linear} , which can be seen in Figure 13.8. As the correctness percentage is also higher for Δ_{linear} , it tells us that when $\epsilon = 0.15$ doesn't return the right answer, the deviation is way more off for Δ_{linear} . We can also see the deviation is fairly well distributed for the different data distributions, meaning when the number of unique numbers is low, the deviation is also low, however, the data structure is less likely to give the correct mode than when the number of unique values is high.

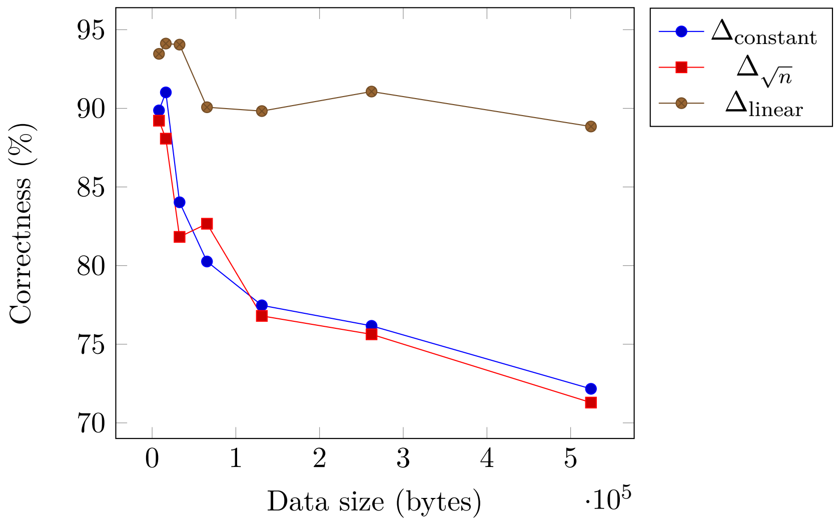

Figure 13.7: Percentage of how often $\epsilon = 0.15$ return the mode

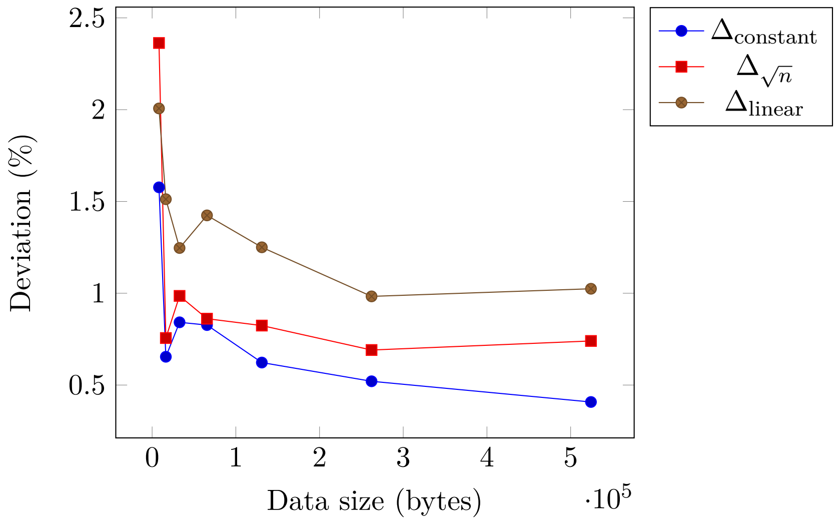

Figure 13.8: Avg. deviation of returned frequency compared to frequency of mode (for $\epsilon = 0.15$ in percentage)

13.4 3-Approx vs $1 + \epsilon$

An interesting point to be made is when to use 3-approx over $1 + \epsilon$ and vice verse.

Query Figure 13.9 shows that $1 + \epsilon$ is much faster with the data sizes we have in our experiments. However, it is also clear that 3-approx is scaling a lot better for almost all data-distributions, arguably with the exception of Δ_{linear} . Overall the results suggest that the data distribution is quite important, when determining the proper data structure. For a constant or low number of unique values, 3-approx is already querying as fast as $1+\epsilon$ at 2 MB input data, and will easily outperform it for a bigger data size.

Contrary, if the number of unique values are big compared to the total number of elements, we see a scaling that is somewhat the same for both of the data structures, in which case $1+\epsilon$ is outperforming 3-approx with quite the margin. The query time of $1+\epsilon$ is in all cases twice as fast as 3-approx for the Δ_{linear} data distribution.

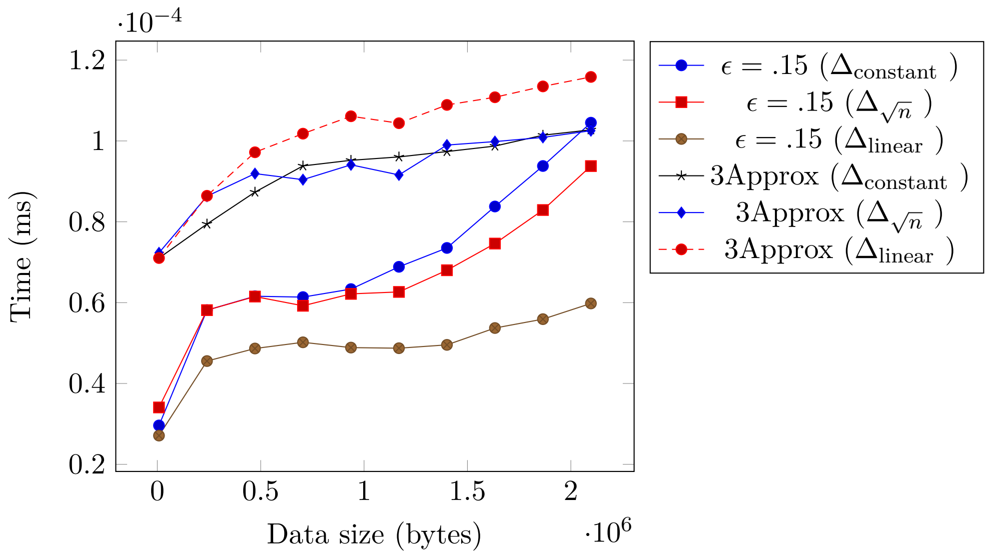

Figure 13.9: Query time comparison of the approximate data structures

Construction We established that $1+\epsilon$ has a very fast query algorithm, especially for low data sizes. The major downside of $1+\epsilon$ is seen in Figure 13.10, which is the scaling of the construction time based on the input size.

If we only consider the construction time of our implementation, there is no scenario where $1+\epsilon$ is better than 3-approx , it is slower by a very wide margin, only increasing as data sizes increase.

Correctness We chose our ϵ based on the correctness compared to 3-approx on Δ_{linear} . We ended up with $\epsilon = 0.15$ as it looked like following 3-approx. However, it does not follow for all data distributions. We can clearly see in Figure 13.11 that for data distributions $\Delta_{\sqrt{n}}$ and Δ_{constant} , ϵ has a much worse correctness than 3-approx for those data distributions. Both data structures is clearly best when there are a lot of unique values, they are both at a level around 90% correctness with 3-approx being around 95% as data size grows. However, for $\Delta_{\sqrt{n}}$ and Δ_{constant} , ϵ is around 70-75% correctness.

A very important tendency here is that ϵ is getting worse when data size increases for the data distributions with less unique values. From 1 MB to 2

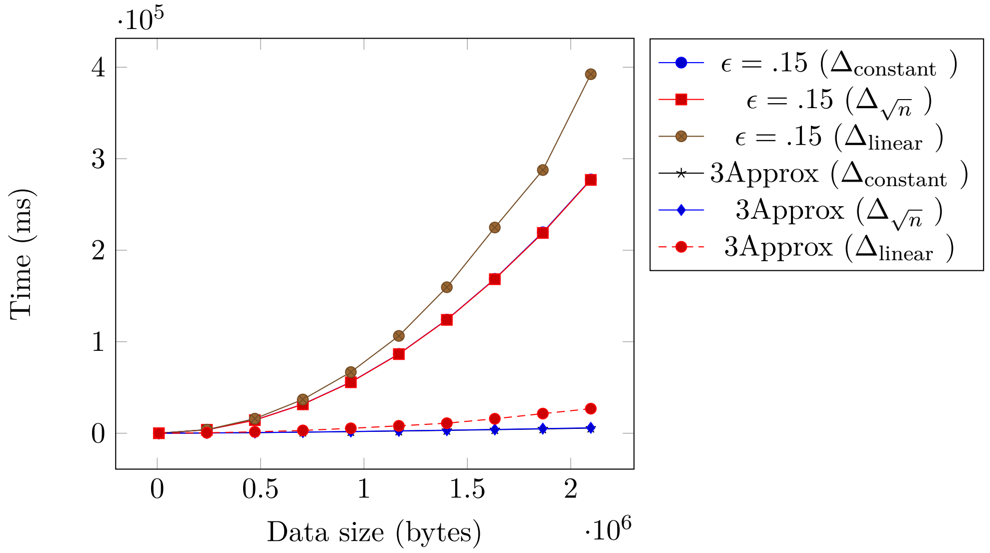

Figure 13.10: Construction time comparison of the approximate data structures

MB we see a drop from 76% to around 72% for $\Delta_{\sqrt{n}}$ and Δ_{constant} . The same tendency is arguably also happening for Δ_{linear} , however it is very difficult to conclude this is the case as our sample size seems a little vague.

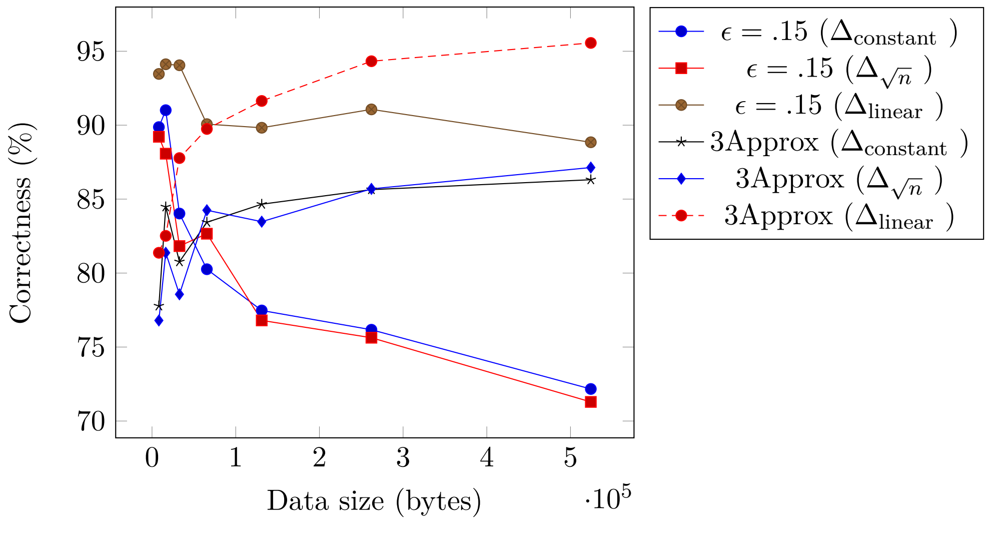

Figure 13.11: Correctness of the approximate data structures

Another aspect of correctness is how far the data structures on average is of the frequency of the actual mode. In Figure 13.12 we see the percentage-wise difference of the frequency of the returned mode versus the frequency of the actual mode.

We can see that ϵ is on average closer to the frequency of the actual mode except for Δ_{linear} , however the difference between the data structures doesn't seem to be that significant.

As we know 3-approx Δ_{linear} has a high rate of correct query responses, so seeing the percentage-wise difference in frequency is bigger for 3-approx must mean that when the 3-approx doesn't return the correct mode, it might be off by a more significant margin than the average suggest.

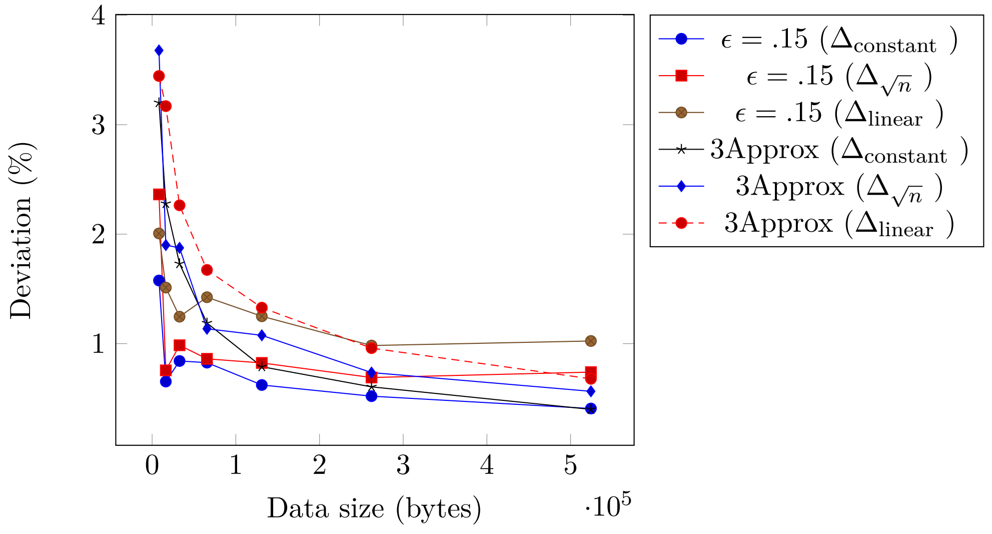

Figure 13.12: Avg. deviation of returned frequency compared to frequency of the mode (in percentage)

Guarantee While 3-approx according to the experiments has a better correctness in terms of mode in all cases, there is a substantial different in the guarantee the two data structures are theoretically proven to uphold, when answering a query. The 3-approx data structure will return a mode that has at least $\frac{1}{3}$ the frequency of the actual mode. The $(1+\epsilon)$-approximate data structure with $\epsilon = 0.15$ will return a value that has a frequency of at least $\frac{1}{1.15}$ the frequency of the correct mode. This is quite significant difference between the two data structures. Even though the correctness experiments suggest 3-approx is giving a correct answer more times than ϵ, one might be in a situation where a strict lower-bound guarantee is required.

The Data Structure of Choice The main reason to use an approximate data structure is if you need to do a lot of queries and can trade the correctness with the speed. For very large data sizes the construction time of $1+\epsilon$ is really big, and there needs to be a ton of queries to make up for the time used on construction. For a data size of 2 MB the construction time for Δ_{linear} on $1+\epsilon$ is about 392.3 seconds with a query time of 60 nanoseconds, in contrast the construction time of 3-approx is 26.8 seconds and the query time is 116 nanoseconds. For these specific times the amount of queries needed for $1+\epsilon$ with $\epsilon = 0.15$ to be better performing than 3-approx is given by the equation,

$$392.3 + 6 * 10^{-8}x = 26.8 + 1.16 * 10^{-7}x$$
$$x = 6.52679 * 10^{9}$$

Which means we need more than 6.5 billion queries in order to account for the construction time. More than that number of queries will make $(1 + \epsilon)$ more viable, otherwise 3-approx seems like a solid choice. As the data size increases the number of queries required to make up for the construction time will increase as well. Keep in mind the implementation we tested on has a $\mathrm{O}\left(n^2\right)$ construction time, however a faster construction algorithm is described by Bose et al.[2], which means that the implementation can be improved to gain a way better construction time, so this is not a reason to entirely discard the use of $1 + \epsilon$. If this is a big concern, it can be improved upon to get a $\mathrm{O}\left(n \log_{\frac{1}{\alpha}} n + n \log n\right)$ construction time by using a bit more space, $\mathrm{O}\left(n/1 - a\right)$ instead of $\mathrm{O}\left(\log_{\frac{1}{\alpha}} n\right)$, thus the space is linear instead of logarithmic.

We can see from Figure 13.9 that the query times for Δ_{linear} , $\epsilon = 0.15$ on 2MB data is about half of the query time for 3-approx, which means if the use case is very performance critical, one might need to have a very, very fast query time, in which case ϵ might be a consideration over 3-approx. Yet, the difference is only tens of nanoseconds, so it has to be a very performance critical application in order for this to be a viable concern.

There is one aspect where 3-approx is not an option, which is if you need a stronger guarantee than $\frac{1}{3}$. In that case $1 + \epsilon$ is the data structure to go for, where ϵ can be tweaked to have a specific guarantee of the frequency of the returned mode compared to the frequency of the actual mode.

13.5 Summary

Both 3-approx and $(1+\epsilon)$-approx is answering queries exceptionally fast. Clearly a lot faster than the exact data structure we considered. When considering which approximate data structure to use, the guarantee is the first aspect that is important. When using 3-approx you are bound to a guarantee of the frequency of the returned value being at least $\frac{1}{3}$ of the frequency of the range mode, however $(1 + \epsilon)$-approx can adjust this guarantee at the cost of speed and storage space. We discussed how the data distribution matters a lot, when looking on correctness of data structures. The data structure 3-approx seems to converge to a higher correctness when the data set size increase, but still is influenced by number of unique values, the more unique values the better correctness. The data distribution matters a lot for $(1+\epsilon)$-approx as well, however, unlike 3-approx it seems like the correctness of $(1 + \epsilon)$-approx is getting worse as the data set size increases. With that being said, we saw that the average deviation didn't increase even though the correctness increase.

Additionally we saw that the generally the deviation in frequency from the returned value and the actual mode is very small as data sizes increases, the average deviation is at most 1% in the experiments conducted.

While the approximate data structures are very fast at answering range mode queries, and very often gives the right answer with a small average deviation, the construction of these data structures are slow. Though, 3-approx is a lot faster than $(1+\epsilon)$-approx. We discussed how many queries needed to be done according to our experiments in order for the $(1+\epsilon)$-approx to be worth the construction compared to 3-approx, if time is the only factor. For a data size of 2 MB with uniformly distributed data and Δ_{linear} it amounted to a staggering 6.5 billion queries.

Chapter 14

Exact vs Approx

We have discussed the experiments for the exact data structures as well as the approximate data structures. In this chapter we will compare the two different types of data structures and outline which data structures are best suited in different settings.

For input data set of size 2 MB on Δ_{linear} , the total time in milliseconds it takes to run x queries by the different data structures are given by the following equations of construction and query time,

$$\begin{aligned}
\text{A:}\ & 552.452 + 0.01098834x \\
\text{B:}\ & 30.8944 + 0.272457847x \\
\text{3-approx:}\ & 26852.0667 + 0.000115831x \\
(\epsilon = 0.15)\text{-approx:}\ & 392315.3333 + 0.0000598x
\end{aligned}$$

We discussed in Chapter 12 that our experiments show that Naive is faster if you need to do less than seven queries for an average query range of $1/3n$. Data structure B is faster if the number of queries is between 5 and 1994 queries. Beyond 1994 queries data structure A becomes faster. Data structure A is the fastest until the number of queries exceeds around 2.4 million, where 3-approx becomes the fastest. This is the sweet spot where the approximate data structures become relevant. This is the first time for Δ_{linear} in our experiments, where the query time out-weights the high construction time of the approximate data structures. As discussed in Chapter 13, $\epsilon = 0.15$ becomes relevant when the number of queries exceeds 6.5 billion.

These numbers are on our specific hardware with our implementations, which means it highly implementation and hardware dependent, however it paints a picture of the state. Even though the approximate data structure intuitively should be faster, it heavily depends on the expected number of queries. Furthermore, it is worth noting we only consider one type of data in as well as only consider the speed over time in this discussion. If a data structure is needed in a performance critical setting, and the construction time of an approximate data structure can be afforded, both 3-approx and $\epsilon = 0.15$ have really fast query times. 3-approx seems to have a very good correctness as n increases, this is not the case ϵ, however the average deviation is good for both of them. The general tendency we see is that the approximate data structures are very expensive to construct, and many queries will be needed to make up for the construction cost, however they are blazingly fast and is very good in performance critical settings. If a very small number of queries are needed, Naive is fine to use, and the smaller query range size, the more queries can be answered for it to still be a good solution. As the number of unique values increases as well as data size, more and more queries will be needed to make the approximate data structures relevant beside the highly performance critical applications.

A scenario where the approximate data structures is sufficient could be a web API with a lot of old statistical data. As the data is old, the construction is a one time thing, and users might settle with an approximation of the mode. In this case we want to serve the users as fast as possible as well as being able to serve as many users as possible, and thus having an approximate data structure seems like a very good fit for this scenario.

We have not implemented data structure A+B, but we can see from our experiments that A is performing well on data that B is not, thus combining the two seems like a good trade off even though there are some constant time look up overhead. By combining the two, we have that the data structure will be better than naive for very low number of queries as well as better than the approx data structures for number of queries up to around 2.4 million.

For data that doesn't have as many unique values as Δ_{linear} , the number of queries is a lot lower. For $\Delta_{\sqrt{n}}$, we see the number of queries drops from 2.4 million to $700,000$. It is clear from the data distribution matters a lot when choosing the data structure. The main reason for this is that the construction time of 3-approx scales horrendous with the data distribution Δ_{linear} compared to the other two Δ_{constant} and $\Delta_{\sqrt{n}}$. Adding to this point, though, 3-approx was shown to return the correct mode way more often on Δ_{linear} data, so even though the construction time scales bad with unique number of values, the correctness percentage is worth factoring in as well.

Chapter 15

Conclusion

If only a very few range mode queries are needed the naive algorithm is the most efficient. But if more than a few range queries are performed data structure B becomes more efficient. In our experiment it would be faster to use data structure B if you want to do more than 7 queries.

When the input data has a high amount of unique values, data structure A will be more efficient than data structure B. Our experiments suggest that a combination of data structure A and B would be very efficient, if you have no knowledge of the input data.

If a lot of range mode queries is required, it would be faster to use the 3-approximate data structure. However, this means the correct mode is not guaranteed to be returned. In our experiments 3-approx is becoming faster than the exact data structures at about 2400000 queries. The number of queries grows with number of unique values as well as data size.

If a better approximate mode than 3-approximate is needed, the $1 + \epsilon$ data structure can be used, where ϵ is the tolerance factor. Even though $1 + \epsilon$ gives a better guarantee, our experiments show that 3-approx on average return the correct mode more often.

Chapter 16

Future work

There is a lot of interesting work still to be done in the practical field of range mode queries. Interesting aspects we would have liked to look more into include how to optimize data structure A + B practical performance. In particular exploring if the bit packing of the data structures actual gives an performance improvement in practice.

We ignored the practical space consumption of the different data structure besides discarding the data structure precomputing modes for every single range. That is another aspect that also has a big impact on which data structure to use for various scenarios.

We compared two data structures, 3-approx from Greve et al.[6] and $\frac{1}{\alpha}$-approx from Bose et al.[2]. Comparing these data structures with more complex data structures like $(1+\epsilon)$-approx from Greve et al.[6] and the data structure to answer α-majority queries by Durocher et al.[5].

The major downside with the approximate data structures in our experiments is the very expensive construction time of the data structures. Improving upon the practical implementation of these algorithms could be very interesting as it would make the approximate data structures viable with a lot less number of queries. An example of such algorithms could be the improved construction time as described by Bose et al.[2]

Something we wanted to look into was when the naive solution would be better in practice. How small do the ranges need to be, and how many queries can naive handle before becoming infeasible.

Bibliography

[1] Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '04, pages 286–296, New York, NY, USA, 2004. ACM.

[2] Prosenjit Bose, Evangelos Kranakis, Pat Morin, and Yihui Tang. Approximate range mode and range median queries. In Volker Diekert and Bruno Durand, editors, *STACS*, volume 3404 of *Lecture Notes in Computer Science*, pages 377–388. Springer, 2005.

[3] Timothy M. Chan, Stephane Durocher, Kasper Green Larsen, Jason Morrison, and Bryan T. Wilkinson. Linear-space data structures for range mode query in arrays. *29th Symposium on Theoretical Aspects of Computer Science, STACS '12*, 2012.

[4] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows: (extended abstract). In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 635–644, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[5] Stephane Durocher, Meng He, J. Ian Munro, Patrick K. Nicholson, and Matthew Skala. Range majority in constant time and linear space. In *Proceedings of the 38th International Colloquim Conference on Automata, Languages and Programming - Volume Part I*, ICALP'11, pages 244–255, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] Mark Greve, Allan Grønlund Jørgensen, Kasper Dalgaard Larsen, and Jakob Truelsen. Cell probe lower bounds and approximations for range mode. *37th International Colloquium on Automata, Languages and Programming*, 2010.

[7] Dov Harel and Robert Andre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, pages 338–335, 1984.

[8] Regant Y. S. Hung, Lap-Kei Lee, and H. F. Ting. Finding frequent items over sliding windows with constant update time. *Inf. Process. Lett.*, 110(7):257–260, March 2010.

[9] Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *CoRR*, cs.DS/0307034, 2003.

[10] L. K. Lee and H. F. Ting. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 290–297, New York, NY, USA, 2006. ACM.

[11] Holger Petersen. Improved bounds for range mode and range median queries. In Viliam Geffert, Juhani Karhumäki, Alberto Bertoni, Bart Preneel, Pavol Návrat, and Mária Bieliková, editors, *SOFSEM*, volume 4910 of *Lecture Notes in Computer Science*, pages 418–423. Springer, 2008.

[12] Holger Petersen and Szymon Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Inf. Process. Lett.*, 109(4):225–228, 2009.

Appendices

Appendix A

Exact Data Structure Results

A.1 Query Time

size	elements	unique	naive	naive-reduce	A	B
8192	2048	100.0	0.013267712	0.000960310	0.000396804	0.000535935
240296	60074	100.0	0.218043623	0.020621812	0.002142678	0.000598029
472404	118101	100.0	0.423297770	0.040329777	0.002828202	0.000626045
704508	176127	100.0	0.631767734	0.060579226	0.003358827	0.000629812
936616	234154	100.0	0.834974517	0.080287207	0.003695607	0.000636882
1168720	292180	100.0	1.036523148	0.100091268	0.003962692	0.000644349
1400828	350207	100.0	1.247144489	0.121542928	0.004198582	0.000656868
1632932	408233	100.0	1.451042738	0.140153022	0.004397167	0.000659460
1865040	466260	100.0	1.648645248	0.159478832	0.004676132	0.000658572
2097144	524286	100.0	1.845932199	0.178219018	0.004944432	0.000659866

Table A.1: Avg. query time of exact data structures on Δ_{constant} (in ms)

size	elements	unique	naive	naive-reduce	A	B
8192	2048	46.0	0.020061144	0.001563258	0.000855467	0.000565065
240296	60074	246.0	0.229543346	0.021034410	0.002303267	0.001264484
472404	118101	344.0	0.443020040	0.040775332	0.003347813	0.001710889
704508	176127	420.0	0.652965402	0.060830835	0.004165944	0.002053611
936616	234154	484.0	0.865635944	0.080728482	0.004882273	0.002349358
1168720	292180	541.0	1.064289137	0.100093321	0.005633199	0.002614763
1400828	350207	592.0	1.280894727	0.120682086	0.006324900	0.002847299
1632932	408233	639.0	1.481204579	0.140204465	0.006936129	0.003066807
1865040	466260	683.0	1.675387123	0.158608216	0.007512103	0.003262429
2097144	524286	725.0	1.886400397	0.179867374	0.008026856	0.003458539

Table A.2: Avg. query time of exact data structures on $\Delta_{\sqrt{n}}$ (in ms)

size	elements	unique	naive	naive-reduce	A	B
8192	2048	205.0	0.019048072	0.001328627	0.000402748	0.000956363
240296	60074	6007.66	0.574342324	0.037629038	0.002400936	0.026067444
472404	118101	11810.33	1.289143099	0.078744115	0.003727163	0.052629579
704508	176127	17612.66	2.097320009	0.132278749	0.004772787	0.081824779
936616	234154	23414.0	2.891258791	0.188549870	0.005789055	0.112454941
1168720	292180	29217.0	3.654046592	0.245778401	0.006426655	0.143988637
1400828	350207	35020.33	4.500222730	0.307880640	0.007457548	0.175087188
1632932	408233	40822.66	5.393431646	0.369608544	0.008365538	0.208813875
1865040	466260	46624.66	6.122635439	0.423755129	0.009663278	0.238473670
2097144	524286	52427.33	6.893377584	0.485170774	0.010988340	0.272457847

Table A.3: Avg. query time of exact data structures on Δ_{linear} (in ms)

A.2 Construction Time

size	elements	unique	naive	navie-r	A	B
8192	2048	100.0000	0.0004	0.0643	0.2420	0.0644
240296	60074	100.0000	0.0150	1.4415	16.5273	1.5132
472404	118101	100.0000	0.0313	2.7168	43.6152	2.9673
704508	176127	100.0000	0.0601	4.0765	77.8111	4.4457
936616	234154	100.0000	0.0811	5.4682	118.0630	5.9292
1168720	292180	100.0000	0.1121	6.7612	162.8370	7.4314
1400828	350207	100.0000	0.1400	8.1002	212.6533	8.9408
1632932	408233	100.0000	0.1512	9.4174	266.7110	10.4290
1865040	466260	100.0000	0.1893	10.7750	324.2400	11.9945
2097144	524286	100.0000	0.2218	12.1515	385.1607	13.4890

Table A.4: Construction time of exact data structures on Δ_{constant} (in ms)

size	elements	unique	naive	navie-r	A	B
8192	2048	46.0000	0.0012	0.1016	0.4286	0.1110
240296	60074	246.0000	0.0393	1.4287	16.4936	1.5898
472404	118101	344.0000	0.0328	2.7388	42.0989	2.9429
704508	176127	420.0000	0.0603	4.1120	74.8488	4.3885
936616	234154	484.0000	0.0822	5.5169	113.0970	5.8629
1168720	292180	541.0000	0.1166	6.8128	156.5073	7.3498
1400828	350207	592.0000	0.1396	8.1497	204.0563	8.8142
1632932	408233	639.0000	0.1537	9.4820	254.6657	10.3052
1865040	466260	683.0000	0.1892	10.8655	309.7840	11.8346
2097144	524286	725.0000	0.2204	12.2105	366.9547	13.3163

Table A.5: Construction time of exact data structures on $\Delta_{\sqrt{n}}$ (in ms)

size	elements	unique	naive	navie-r	A	B
8192	2048	205.0000	0.0004	0.0775	0.2771	0.0768
240296	60074	6008.0000	0.0143	2.2865	19.6577	2.3823
472404	118101	11810.6667	0.0310	5.5634	54.1643	5.6968
704508	176127	17611.3333	0.0597	9.2401	99.5134	9.5999
936616	234154	23414.6667	0.0812	12.7002	152.7117	13.2519
1168720	292180	29218.0000	0.1115	15.9750	213.9323	16.6865
1400828	350207	35019.6667	0.1402	19.8011	285.3350	20.6373
1632932	408233	40822.0000	0.1510	24.1257	362.1827	25.2610
1865040	466260	46624.3333	0.1895	27.5290	454.9357	29.2286
2097144	524286	52426.6667	0.2209	29.2434	552.4520	30.8944

Table A.6: Construction time of exact data structures on Δ_{linear} (in ms)

Appendix B

Approx Data Structure Results

B.1 Query Time

size	elements	unique	3-approx	$1+\epsilon$
8192	2048	100.0	0.000071023	0.000029624
240296	60074	100.0	0.000079448	0.000058135
472404	118101	100.0	0.000087342	0.000061582
704508	176127	100.0	0.000093821	0.000061370
936616	234154	100.0	0.000095211	0.000063349
1168720	292180	100.0	0.000096043	0.000068861
1400828	350207	100.0	0.000097373	0.000073528
1632932	408233	100.0	0.000098729	0.000083782
1865040	466260	100.0	0.000101417	0.000093799
2097144	524286	100.0	0.000102745	0.000104509

Table B.1: Avg. query time of approx data structures on Δ_{constant} (in ms)

size	elements	unique	approx	epsilon
8192	2048	46.0	0.000072320	0.000034075
240296	60074	246.0	0.000086323	0.000058142
472404	118101	344.0	0.000091920	0.000061505
704508	176127	420.0	0.000090424	0.000059224
936616	234154	484.0	0.000094095	0.000062165
1168720	292180	541.0	0.000091589	0.000062648
1400828	350207	592.9	0.000098982	0.000068013
1632932	408233	639.7	0.000099841	0.000074612
1865040	466260	683.0	0.000100870	0.000082908
2097144	524286	725.0	0.000102540	0.000093780

Table B.2: Avg. query time of approx data structures on $\Delta_{\sqrt{n}}$ (in ms)

size	elements	unique	approx	epsilon
8192	2048	205.0	0.000071029	0.000027111
240296	60074	6007.666666666667	0.000086431	0.000045548
472404	118101	11810.333333333334	0.000097197	0.000048654
704508	176127	17612.666666666668	0.000101797	0.000050187
936616	234154	23414.0	0.000106104	0.000048878
1168720	292180	29217.0	0.000104387	0.000048729
1400828	350207	35020.333333333336	0.000108908	0.000049553
1632932	408233	40822.666666666664	0.000110806	0.000053719
1865040	466260	46624.666666666664	0.000113491	0.000055914
2097144	524286	52427.333333333336	0.000115831	0.000059800

Table B.3: Avg. query time of approx data structures on Δ_{linear} (in ms)

B.2 Construction Time

size	elements	unique	3-approx	$1 + \epsilon$ ($\epsilon = 0.15$)
8192	2048	100.0000	3.0877	6.5656
240296	60074	100.0000	258.3273	3805.1333
472404	118101	100.0000	676.5610	14380.6000
704508	176127	100.0000	1171.6367	31716.6333
936616	234154	100.0000	1772.4000	55895.1667
1168720	292180	100.0000	2439.8033	86655.3667
1400828	350207	100.0000	3135.2967	124110.3333
1632932	408233	100.0000	3908.4267	168683.6667
1865040	466260	100.0000	4740.2067	219569.3333
2097144	524286	100.0000	5606.8267	277292.0000

Table B.4: Construction time of approx data structures on Δ_{constant} (in ms)

B.3 Correctness

size	elements	unique	3-approx	$1+\epsilon$ ($\epsilon = 0.15$)
8192	2048	46.0000	3.3604	7.4126
240296	60074	246.0000	274.2943	3762.7133
472404	118101	344.0000	706.5127	14318.0000
704508	176127	420.0000	1231.4833	31569.3333
936616	234154	484.0000	1853.2200	55693.7667
1168720	292180	541.0000	2591.5133	86467.7667
1400828	350207	592.0000	3298.0200	123899.6667
1632932	408233	639.0000	4100.9233	168340.6667
1865040	466260	683.0000	5009.7267	218875.0000
2097144	524286	725.0000	5923.2967	276842.6667

Table B.5: Construction time of approx data structures on $\Delta_{\sqrt{n}}$ (in ms)

size	elements	unique	3-approx	$1+\epsilon$ ($\epsilon = 0.15$)
8192	2048	205.0000	3.4038	6.0480
240296	60074	6008.0000	488.8340	3923.5733
472404	118101	11810.6667	1538.6367	15916.1667
704508	176127	17611.3333	3048.2867	36866.0333
936616	234154	23414.6667	5392.7833	66831.6000
1168720	292180	29218.0000	8082.2600	106376.0000
1400828	350207	35019.6667	10961.6333	159540.3333
1632932	408233	40822.0000	15811.5333	224821.6667
1865040	466260	46624.3333	21455.2000	287614.0000
2097144	524286	52426.6667	26852.0667	392315.3333

Table B.6: Construction time of approx data structures on Δ_{linear} (in ms)

size	elements	unique	3-approx	$1+\epsilon$ ($\epsilon = 0.15$)
8192	2048	100.0	77.77776666666666	89.8693
16384	4096	100.0	84.47713333333333	91.01310000000001
32768	8192	100.0	80.7661	84.0261
65536	16384	100.0	83.43509999999999	80.2605
131072	32768	100.0	84.65606666666666	77.47253333333333
262144	65536	100.0	85.64303333333334	76.1701
524288	131072	100.0	86.3116	72.16539999999999

Table B.7: Correctness of approx data structures on Δ_{constant} (in %)

size	elements	unique	3-approx	$1+\epsilon$ ($\epsilon = 0.15$)
8192	2048	46.0	76.7974	89.21570000000001
16384	4096	64.0	81.37253333333332	88.07189999999999
32768	8192	91.0	78.5656	81.8256
65536	16384	128.0	84.2491	82.6618
131072	32768	182.0	83.47579999999999	76.80096666666667
262144	65536	256.0	85.69393333333333	75.63086666666668
524288	131072	363.0	87.13566666666668	71.2905

Table B.8: Correctness of approx data structures on $\Delta_{\sqrt{n}}$ (in %)

size	elements	unique	3-approx	$1+\epsilon$ ($\epsilon = 0.15$)
8192	2048	205.0	81.37256666666666	93.46403333333332
16384	4096	410.0	82.51633333333332	94.11763333333334
32768	8192	820.0	87.77506666666666	94.05053333333335
65536	16384	1639.0	89.74360000000001	90.06920000000001
131072	32768	3276.6666666666665	91.6361	89.82496666666667
262144	65536	6554.0	94.32233333333333	91.06633333333332
524288	131072	13107.333333333334	95.55926666666666	88.8448

Table B.9: Correctness of approx data structures on Δ_{linear} (in %)

size	elements	unique	3-approx	$1+\epsilon$ ($\epsilon = 0.15$)
8192	2048	100.0	3.20073	1.5769433333333334
16384	4096	100.0	2.2793966666666665	0.6539933333333333
32768	8192	100.0	1.7302133333333334	0.841795
65536	16384	100.0	1.1885199999999998	0.8267236666666666
131072	32768	100.0	0.7895326666666667	0.6225146666666667
262144	65536	100.0	0.6055083333333333	0.5205076666666667
524288	131072	100.0	0.39957266666666663	0.40768733333333335

Table B.10: Avg. deviation in frequency compared to frequency of mode on Δ_{constant} (in %)

size	elements	unique	3-approx	$1+\epsilon$ ($\epsilon = 0.15$)
8192	2048	46.0	3.67844	2.3632
16384	4096	64.0	1.9003533333333333	0.7560743333333333
32768	8192	91.0	1.8749366666666667	0.985542
65536	16384	128.0	1.1359233333333334	0.861424
131072	32768	182.0	1.0761830000000001	0.8239779999999999
262144	65536	256.0	0.735977	0.6907813333333334
524288	131072	363.0	0.5647346666666666	0.7401076666666665

Table B.11: Avg. deviation in frequency compared to frequency of mode on $\Delta_{\sqrt{n}}$ (in %)

size	elements	unique	3-approx	$1 + \epsilon$ ($\epsilon = 0.15$)
8192	2048	205.0	3.4446499999999998	2.0067253333333332
16384	4096	410.0	3.1694166666666668	1.5122893333333334
32768	8192	820.0	2.2634133333333337	1.2464823333333335
65536	16384	1639.0	1.6734466666666667	1.4241466666666664
131072	32768	3276.6666666666665	1.3288399999999998	1.2503149999999998
262144	65536	6554.0	0.9592733333333333	0.9829923333333334
524288	131072	13107.333333333334	0.6789506666666667	1.0244273333333334

Table B.12: Avg. deviation in frequency compared to frequency of mode on Δ_{linear} (in %)