CLASSIC AND NEW DATA STRUCTURE PROBLEMS IN EXTERNAL MEMORY

by

Zhewei Wei

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy
in Computer Science and Engineering

February 2012, Hong Kong

Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Zhewei Wei

2 February 2012

CLASSIC AND NEW DATA STRUCTURE PROBLEMS IN EXTERNAL MEMORY

by

Zhewei Wei

This is to certify that I have examined the above PhD thesis
and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by
the thesis examination committee have been made.

Dr. Ke Yi, Thesis Supervisor

Prof. Mounir Hamdi, Head of Department

Department of Computer Science and Engineering
2 February 2012

Acknowledgments

I would like to give my sincere thanks to the many people who have made my PhD years such a wonderful time.

First and formost, I am deeply grateful to my advisor, Dr. Ke Yi. You have been supportive since the first day I came to HKUST, and have oriented and guided me with patience and care throughout my PhD studies. Your passion for research and science sets an example, and encourages me to continue my career as an academic researcher in computer science. Above all, you are also a generous friend, which I always appreciate from my heart.

My debt of gratitude also goes to the co-authors of all my research papers: Pankaj Agarwal, Graham Cormode, Zengfeng Huang, Jeff Phillips, Rasmus Pagh, Lu Wang, Ke Yi, and Qin Zhang. It has been a pleasure to work with all of you, and I have learned a lot from your insights.

The theoretical computer science group has provided a very nice working environment for the past four years. I have benefited greatly from the discussions with a number of faculty members and graduate students in the group. For this I would like to thank Sunil Arya, Siu-Wing Cheng, Mordecai Golin, Zengfeng Huang, Jiongxin Jin, Ge Luo, Lu Wang, Qi Wang, Yajun Wang, Jian Xia and Qin Zhang. Your inspiring suggestions and comments are much appreciated.

I am also indebted to Feifei Li and Pankaj Agarwal, for hosting me during my visits to Florida State University and Duke University.

Many thanks to Siu-Wing Cheng, Mordecai Golin, Yitong Yin and Xiangtong Qi, who serve on my PhD examination committee.

I am fortunate to have spent my PhD years at The Hong Kong University of Science and Technology. The stunning seaside campus has always been a source of inspiration, for both my research and photography. I am also grateful for making so many wonderful friends in this campus. A big thank you goes to Shan Chen, Shaoming Huang, Zhifeng Lai, Yueqi Li, Tengfei Liu, Lixing Wang and Pingzhong Tang. My PhD life would not be so enjoyable without you.

Lastly but most importantly, I want to thank my family for their support and encouragement. Special thanks also go to Wenxin Yu, for all her love and support. I dedicate this dissertation to them.

Table of Contents

Title Page i
Authorization Page ii
Signature Page iii
Acknowledgments iv
Table of Contents v
List of Figures vii
Abstract viii
Chapter 1 Introduction 1
1.1 Models 3
1.1.1 The RAM Model 3
1.1.2 The I/O Model 3
1.1.3 The Cache-Oblivious Model 4
1.2 The Dictionary Problem 5
1.2.1 Classic results on hash tables 5
1.2.2 Buffering for Dynamic External Hashing 7
1.2.3 Cache-Oblivious Hashing 8
1.3 Equivalence between Priority Queues and Sorting in External Memory 9
1.4 Summary Queries 10
Chapter 2 Dynamic External Hashing 12
2.1 Introduction 12
2.2 Lower Bounds 14
2.3 Lower Bounds for Randomized Hash Tables 20
2.4 Upper Bounds 21
Chapter 3 Cache-Oblivious Hashing 24
3.1 Introduction 24
3.2 Analysis of Linear Probing in the Cache-Oblivious Model 25
3.3 Blocked Probing 27
3.3.1 Algorithm description 27
3.3.2 Cache-oblivious analysis of blocked probing 28
3.3.3 Cache-oblivious dynamic hash tables 32
3.4 Lower Bounds 33
3.4.1 The model 33
3.4.2 Good inputs and bad inputs 34
3.4.3 Lower bound for the boundary-oblivious model 36

3.4.4 Lower bound for the block-size-oblivious model 36
Chapter 4 Equivalence between Priority Queues and Sorting in External Memory 40
4.1 Introduction 40
4.2 Structure 42
4.3 Operations 44
4.4 Analysis of Amortized I/O Complexity 50
Chapter 5 Data Structure for Summary Queries 53
5.1 Introduction 53
5.1.1 Related work on data structure for aggregation queries 53
5.1.2 Related work on summaries 54
5.1.3 Other related work 56
5.1.4 Our results 56
5.2 A Baseline Solution 57
5.3 Optimal Data Structure for F_1 Based Summaries 58
5.3.1 Optimal internal memory structure 59
5.3.2 Optimal external memory data structure 61
5.4 Summaries 66
5.4.1 Heavy hitters 66
5.4.2 Quantiles 68
5.4.3 The Count-Min sketch 70
5.4.4 The AMS sketch and wavelets 73
5.5 Handling Updates 73
Chapter 6 Future Directions 75
6.1 External Hashing 75
6.2 Summary Queries 75
Bibliography 77

List of Figures

1.1 The I/O model. 4

2.1 The query-insertion tradeoff. 13

3.1 When two I/Os are needed. 33

4.1 The components of the priority queue. 43

5.1 A schematic illustration of our internal memory structure. 60

5.2 The standard B-tree blocking of a binary tree. 62

5.3 The summaries we store for an internal block $\mathcal{B}$. 63

5.4 A schematic illustration of our packed structure. 65

CLASSIC AND NEW DATA STRUCTURE PROBLEMS IN EXTERNAL MEMORY

by

Zhewei Wei

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology

Abstract

The demand of efficient data structures for query processing on massive data sets has grown tremendously in the past decades. Traditionally, data structures are designed and analyzed in the RAM model, where each memory cell can be accessed with unit cost. This assumption, however, is unrealistic for modeling modern memory hierarchies which consist of many levels of memories and caches with different sizes and access costs. As a consequence, a number of more elaborate models were introduced. Among them the most successful ones are the *I/O model* and the *cache-oblivious model*. In recent years, designing data structures that are *I/O-efficient* or *cache-oblivious* has become an active direction in both the theory and database communities.

This thesis starts by considering the *dictionary* problem, one of the most basic data structure problems, in which we want to store and access a set of (key, data) pairs. Hash tables are the most efficient and the most fundamental data structure for implementing a dictionary. In this thesis we study hash tables in both the I/O model and the cache-oblivious model. We first show an inherent query-insertion tradeoff of hashing in the I/O model, which implies that the *buffering* technique is essentially useless for hash tables. In the cache-oblivious model, we build a hash table that achieves the same search cost as its cache-aware version does, for all block sizes.

The second problem studied is another fundamental data structure problem, *priority queues*. The priority queue problem is well understood in the comparison based I/O model, where its complexity is known to be the same as sorting. In this thesis, we establish their equivalence in the I/O model without any restrictions, by providing a reduction from priority queues to sorting. Note that the other direction of the reduction is trivial.

The third problem studied in this thesis is the *summary query* problem, which is

a natural generalization of the *range searching* problem. Our goal is to design data structures that allow for extracting a statistical summary of all the records in the query range. The summaries we support include frequent items, quantiles, various sketches, and wavelets, all of which are of central importance in massive data analysis.

Chapter 1

Introduction

The demand of efficient data structures for query processing on massive data sets has grown tremendously in the past decades. These data sets could easily reach the order of terabytes or even petabytes, and the rate of this information explosion is still accelerating. A new challenge for computer scientists and engineers is to develop techniques that keep up with this increasing rate.

Traditionally, data structures and algorithms are designed and analyzed in the RAM model, where a unit cost for accessing each memory cell is assumed. However, this assumption is unrealistic for modeling modern *memory hierarchies*. A modern computer usually consists of several memory levels with different sizes and access costs. One strong justification for abandoning the RAM model is the fact that accessing a block on a disk is millions of times slower than accessing the main memory. As a consequence, a number of more elaborate memory hierarchy models were introduced. Among them the most successful ones are the *I/O model* (a.k.a. *external memory model*) and the *cache-oblivious model*. The I/O model, introduced by Aggarwal and Vitter [4], considers a two-level memory hierarchy: a small and fast *internal memory*, and a slow but conceptually unlimited *external memory*. Data is transferred between internal and external memory in terms of blocks, and the cost of an algorithm in this model is the number of block transfers (or *I/Os*). Recently Frigo [30] proposed the cache-oblivious model which takes the approach of making the data structures unaware of the block size and internal memory size, so that an optimal cache-oblivious algorithm or data structure is simultaneously optimal with respect to any two neighboring levels in a memory hierarchy with any number of levels.

The *dictionary* problem is one of the most basic data structure problems, in which we want to store and access a set of keys, where each key is associated with some data. Designing data structures for the dictionary problem has received a lot of attention in both the theory and database communities. Among these works, *hash tables* are arguably the most efficient way to implement a dictionary. A hash table supports all of the dictionary operations, namely, query, insertion and deletion, in expected constant time. It works especially well in external memory, where the storage is divided into disk blocks, each containing up to B items. Thus collisions happen only when more than B items are hashed to the same location. Large blocks help to push the performance of external hash tables to the limit: Using some common collision resolution strategies such as *linear probing* or

chaining, Knuth [47] showed that, under the ideal random hash function assumption, the expected average cost of a query is merely $1 + 1/2^{\Omega(B)}$ I/Os. As typical values of B range from a few hundreds to a thousand, the query cost is extremely close to just one I/O.

In this thesis, we study hash tables in both the I/O model and the cache-oblivious model. The first result is on the inherent query-insertion tradeoff of hashing in the I/O model, where a memory buffer is available for batching up newly inserted keys. Indeed, one could find many examples, such as the *buffer tree* [9] and *priority queues* [10, 25], in which the memory is used as a buffer space to batch up insertions and we can write them to disk periodically, fully utilizing the parallelism within one I/O and reducing the amortized insertion cost. We will investigate the possibility of lowering the insertion cost of a dynamic hash table by buffering without sacrificing its near-perfect query performance. The second result in this thesis concerns the performance of hash tables in the cache-oblivious model, where we encounter the challenge of designing a hash table that is unaware of the block size. Our goal is to lay out a hash table such that its search cost matches its cache-aware version, for all block sizes B.

The priority queue is an abstract data structure of fundamental importance. A priority queue maintains a set set of keys, each associated with some data, under insertion, deletion and findmin operations. A findmin operation returns the current minium key in the priority queue along with its associated data. The priority queue is closely related to the sorting algorithm due to the fact that a priority queue can be use to implement a sorting algorithm. Thorup [66] proved that the converse is also true in the RAM model. In particular, he designed a priority queue that uses the sorting algorithm as a black box, such that the update time of the priority queue is asymptotically the same as the per key cost of sorting. In this thesis, we investigate the possibility of establishing such a reduction in external memory. We show that priority queues are almost computationally equivalent to sorting in external memory. The reduction provides a possibility for proving lower bound for external sorting via showing a lower bound for priority queues.

In this thesis we also propose and study the *summary query* problem, a generalization of the classic *range searching* problem. Range searching queries can be broadly classified into two categories: reporting queries and aggregation queries. The former retrieves a collection of records from the data set that match the query's conditions, while the latter returns an aggregate, such as count, sum, average, or max (min), of a particular attribute of these records. However, reporting and aggregation queries provide only two extremes for exploring the data. Data analysts often need more insight into the data distribution than what those simple aggregates provide, and yet certainly do not want the sheer volume of data returned by reporting queries. In this thesis, we design data structures that allow for extracting a statistical *summary* of all the records in the query. The summaries we

support include frequent items, quantiles, various sketches, and wavelets, all of which are of central importance in massive data analysis.

1.1 Models

1.1.1 The RAM Model

The most commonly used model for designing data structures is the *unit-cost word RAM*. In this model the memory is an array of bit strings called words. Each word consists of w bits, for a positive integer parameter w, called the *word length*. In the thesis we assume the universe $[U] = \{0, \ldots, 2^w - 1\}$, or equivalently $w = \log U$. This assumption is referred to as the *trans-dichotomous* assumption [28]. All computation takes place in the CPU that has a constant number of words of registers, on which standard operations on words can be carried out in constant time. We adopt the *multiplication model*, whose instruction set includes addition, bit-wise boolean operations, shifts, and multiplication. We measure the space requirement of a word-RAM algorithm in units of w-bit words. Since accessing memory is much slower than executing instructions, typically an algorithm's cost is measured only by the number of memory words it accesses.

A major drawback of RAM is the lack of ability to capture the fundamental characteristics of modern hierarchical memory systems. As a consequence, a number of more elaborate models have been introduced in recent years. Among them the I/O model and the cache-oblivious model are the most successful ones. In this thesis we will mainly consider these two models, but the RAM model will serve as a basis for designing our data structures.

1.1.2 The I/O Model

The I/O model, introduced by Aggarwal and Vitter [4] in 1988, is the most widely used model for building and analyzing external memory data structures. In this model we consider a two-level memory hierarchy: a slow but conceptually unlimited external memory and a fast internal memory of size M. All computation takes place on data in internal memory, and the transfer of data between internal and external memory takes place in blocks of B consecutive words; the complexity of an algorithm is the number of such *I/Os* (sometimes called *block transfers*) it performs. It is assumed that algorithms have complete control over transfers of blocks between the two levels. Please refer to Figure 1.1. The I/O model captures the essence of the memory hierarchy when the memory transfer between two levels of the memory hierarchy dominates the running time, and it is sufficiently simple to make analysis of algorithms feasible. By now, a large number of results

for the I/O model have been obtained — see the surveys by Arge [8] and Vitter [69].

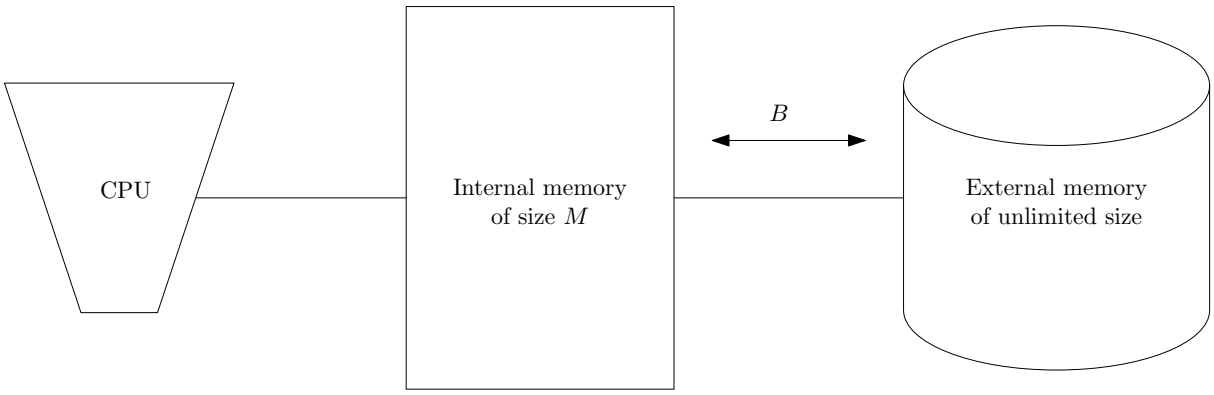

Figure 1.1: The I/O model.

1.1.3 The Cache-Oblivious Model

The disadvantage of the I/O model is that the blocking of a data structure must be programmed delicately, resulting in programs that do not adapt well when the dominating memory level changes. Starting in the late 90's, tremendous efforts have been devoted to the design and analysis of data structures that work well not only in a two-level memory model, but also in a memory hierarchy that consists of any number of levels, where each level has a different capacity M and block size B. Among them, the most successful approach is the *cache-oblivious* model [30] due to its elegance and simplicity. The cache-oblivious model is very similar to the I/O model, the only difference being that a cache-oblivious data structure is unaware of the parameter B and M. In other words, a cache-oblivious algorithm is formulated in the RAM model but analyzed in the I/O model, with the analysis required to hold for any B and M. The main idea of the cache-oblivious model is that by avoiding any memory-specific parametrization, the cache-oblivious algorithm has an optimal number of memory transfers between all levels of an unknown, multilevel memory hierarchy. Thus the cache-oblivious model is effectively a way of modeling a complicated multi-level memory hierarchy using the simple two-level I/O-model.

Another major benefit of cache-oblivious algorithms and data structures is that they achieve their guaranteed performance without any hardware-specific tuning. This is particularly important in autonomous databases, and is in fact the main motivation of the recent efforts in bringing cache-oblivious techniques to databases, such as EaseDB [41].

1.2 The Dictionary Problem

The *dictionary* is an important class of data structures in computer science. Given a subset $\mathcal{D}$ of a universe $[U]$, a dictionary stores $\mathcal{D}$, such that the following queries can be answered efficiently:

- Membership: Given a *key* $x \in [U]$, does x belong to $\mathcal{D}$?

A *static* dictionary is one that does not change over time. A *dynamic* dictionary should also support updates:

- Insertion: Include x into the dictionary;

- Deletion: Remove x from the dictionary.

In many applications each key of $\mathcal{D}$ is associated with some data, and insertion will include a pair (key, data) into the dictionary. The *lookup* operation is considered in these applications:

- Lookup: Does x belong to $\mathcal{D}$? If so, what is its associated data?

1.2.1 Classic results on hash tables

Hash tables are the most efficient way for implementing a dictionary. They are arguably one of the most fundamental data structures in computer science, due to their simplicity of implementation, excellent performance in practice, and many nice theoretical properties. A hash table supports all of the above dictionary operations in expected constant time. Based on the output, the lookup operation for hash tables are usually divided into two categories:

- Successful query: For a queried key that is present in the table, retrieve the data associated with it;

- Unsuccessful query: For a queried key that is not present in the table, return "not found".

Hash table is also one of the simplest data structures. Let N be the size of $\mathcal{D}$. Let $h : [U] \to [R]$ be a hash function, where $[U]$ is the universe of the keys and $[R]$ is the memory range. The table has size $R \geqslant N$ and we simply store key x in position $h(x)$. If that position already contains some other key, one can use various collision resolution strategies, among which *chaining* and *linear probing* are the most common ones. In chaining, we simply store all keys that are mapped to the same position in a list associated

with that position. In linear probing, if position $h(x)$ is already occupied when x is being inserted, we successively probe positions $h(x), h(x)+1, \ldots, R-1, 0, 1, \ldots, h(x)-1$ until an empty position is found and we will put x there. To perform a search on x, we follow the same probing sequence, until x is found or an empty position is encountered, in which case we know that x is not stored in the table. It is known that linear probing generally outperforms chaining in practice due to its sequential access pattern, provided that the *load factor* $\alpha = N/R$ is not too close to 1.

The mathematical analysis of hashing is usually considered as the birth of analysis of algorithms [47], and it is still attracting a lot of attention nowadays. Most analyses on hashing assume h is a truly random function, i.e., each $h(x)$ is independently uniformly distributed on $[R]$. Under such an assumption, Knuth [47] showed that the expected average number of probes during a search using linear probing is (averaged over all keys):

$$\begin{aligned} C_N &\approx \tfrac{1}{2}\left(1 + \tfrac{1}{1-\alpha}\right) && \text{(successful lookup);} \\ C'_N &\approx \tfrac{1}{2}\left(1 + \left(\tfrac{1}{1-\alpha}\right)^2\right) && \text{(unsuccessful lookup).} \end{aligned}$$

Thus, for a typical load factor $\alpha = 0.7$, we expect to make 2.17 probes if the searched key is in the table, and 6.05 probes if it is not.

Hashing with limited independence

The classic results assume that the hash function h distributes each key x independently uniformly on R. Such a function is called a truly random function. This assumption is unrealistic, since to simply store a truly random hash function requires $U \log R$ bits. To bridge the gap between hashing algorithms and their analysis, Carter and Wegman introduced *universal hashing* [21]. A family $\mathcal{H}$ of functions from $[U]$ to $[R]$ is *k-wise independent* if for any k distinct elements $x_1, \ldots, x_k \in [U]$ and h chosen uniformly at random from $\mathcal{H}$, the random variables $h(x_1), \ldots, h(x_k)$ are independent. We refer to the variable

$$\bar{\alpha} \stackrel{\text{def}}{=} N \cdot \max_{x \in [U], \rho \in [R]} \Pr_{h \in \mathcal{H}}[h(x) = \rho]$$

as the *maximum load* of $\mathcal{H}$. If $\mathcal{H}$ distributes hash function values of all elements of U uniformly on R, we will have $\bar{\alpha} = \alpha$, and in general $\bar{\alpha} \geqslant \alpha$. We assume that all families used in this thesis are uniform so we do not distinguish $\bar{\alpha}$ from α. For non-uniform families, all results in this thesis hold if we substitute α with $\bar{\alpha}$.

Carter and Wegman [73] exhibited the following family of k-wise independent hash functions where $U = p$ is a prime:

$$\mathcal{H}_k = \left\{h : h(x) = \left((a_{k-1}x^{k-1} + \cdots + a_0) \bmod p\right) \bmod R,\ a_j \in [p]\right\}.$$

This could be easily verified: observe that the family of degree $k-1$ polynomials in the finite field Z_p is k-wise independent; to obtain a smaller range $[R]$ we may map integers in $[p]$ down to $[R]$ by a modulo R operation. This operation preserves independence, only making the family (slightly) non-uniform. Specifically, the maximum load $\bar{\alpha}$ for this family is in the range $[\alpha, (1+R/p)\alpha]$. By choosing p much larger than R we can make $\bar{\alpha}$ arbitrarily close to α.

External Hashing

Hash tables work especially well in external memory, where the storage is divided into disk blocks, each containing up to B items. Thus collisions happen only when there are more than B items hashed to the same location. Large blocks help to push the performance of external hash tables to the limit: Knuth [47] showed that under the truly random hash function assumption, the external version of linear probing has a search cost of $t_q = 1+1/2^{\Omega(B)}$ I/Os (for both successful and unsuccessful searches), where B is the block size. Here and further we assume that the load factor α is bounded away from 1. In the external version of linear probing, the table consists of R/B blocks, and correspondingly we use a hash function $h : [U] \to [R/B]$. To do a search on x, we successively access blocks $h(x), h(x)+1, \ldots$ until x is found or a non-full block is encountered. The intuitive explanation for this extremely close-to-one I/O cost is that since a block has size B, we will not have a collision unless more than B keys are hashed into this block, which happens with probability exponentially small in B. Knuth [47] actually derived the constant in the big-Omega, showing that for reasonably large B (larger than 10), the number of I/Os is very close to 1, much smaller than the number of probes. Meanwhile, a natural external version of chaining also achieves the same bound. If one wants to maintain the load factor we can periodically rebuild the hash table using schemes like *extensible hashing* [27] or *linear hashing* [51], and this only adds an extra cost of $O(1/B)$ I/Os amortized. Jensen and Pagh [44] demonstrate how to maintain the load factor at $\alpha = 1 - O(1/B^{\frac{1}{2}})$ while still supporting queries in $1 + O(1/B^{\frac{1}{2}})$ I/Os and updates in $1 + O(1/B^{\frac{1}{2}})$ I/Os.

1.2.2 Buffering for Dynamic External Hashing

In Chapter 2 we study the problem of buffering updates for dynamic external hash tables, that is, achieving $o(1)$ amortized I/O cost for updates. As mentioned previously, in the I/O model, the internal memory can be used as a buffer space to batch up insertions and write them to disk periodically, fully utilizing the parallelism within one I/O and reducing the amortized insertion cost. The abundant research in the area of I/O-efficient data structures has witnessed this phenomenon numerous times, where the insertion cost

can be typically brought down to close to $O(1/B)$ I/Os. Examples include the simplest structures like stacks and queues, to more advanced ones such as the buffer tree and priority queues. Many of these results hold as long as the buffer has just a constant number of blocks; some require a larger buffer of $\Theta(B)$ blocks (known as the *tall cache* assumption). Please see the book by Vitter [70] for a complete account of the power of buffering.

Therefore the natural question is, can we (or not) lower the insertion cost of a dynamic hash table by buffering without sacrificing its near-perfect query performance? Jensen and Pagh [44] recently conjectured that the insertion cost must be $\Omega(1)$ I/Os if the query cost is required to be $O(1)$ I/Os.

In Chapter 2, we partially confirm this conjecture. Specifically, we show that for any constant $c > 1$, if the expected average successful query cost is targeted at $1 + O(1/B^c)$ I/Os, then it is not possible to support insertions in less than $1 - O(1/B^{\frac{c-1}{6}})$ I/Os amortized, which means that the memory buffer is essentially useless. While if the query cost is relaxed to $1+O(1/B^c)$ I/Os for any constant $c < 1$, there is a simple dynamic hash table with $o(1)$ insertion cost.

1.2.3 Cache-Oblivious Hashing

In Chapter 3 we study the problem of building a cache-oblivious hash table with linear space and supports efficient lookup operations. Note that the external versions of linear probing and chaining mentioned above only work for a single B, so they are not cache-oblivious. In Chapter 3, we investigate whether it is possible to lay out a hash table such that its search cost matches its cache-aware version, i.e., $1 + 1/2^{\Omega(B)}$ I/Os, for all block sizes B.

We first show that linear probing, a classical collision resolution strategy for hash tables, can be easily made cache-oblivious but it only achieves $t_q = 1 + \Theta(\alpha/B)$ even if a truly random hash function is used. Then we demonstrate that the block probing algorithm [59] achieves $t_q = 1 + 1/2^{\Omega(B)}$, thus matching the cache-aware bound, if the following two conditions hold: (a) B is a power of 2; and (b) every block starts at a memory address divisible by B. We also analyze the performance of blocked probing when the hash function has limited independence. Note that the two conditions hold on a real machine, although they are not stated in the cache-oblivious model. Interestingly, we also show that neither condition is dispensable: if either of them is removed, the best obtainable bound is $t_q = 1 + O(\alpha/B)$, which is exactly what linear probing achieves.

1.3 Equivalence between Priority Queues and Sorting in External Memory

A priority queue is a data structure that maintains a dynamic ordered set of keys, along with associated data. The basic operations are: insertion of a key, deletion of a key, and finding the smallest key. The complexity of the priority queue is closely related to the sorting algorithm. It is well known that a priority queue can be use to implement a sorting algorithm: we simply insert all keys to be sorted into the priority queue, and then repeatedly delete the minimum key to extract the keys in sorted order. Thorup showed that the converse is also true in the RAM model. In particular, he showed that given a sorting algorithm that sorts N keys in $N \cdot S(N)$ time, there is a priority queue that uses the sorting algorithm as a black box, and supports any of the three operations in $O(S(N))$ time. The reduction uses linear space. The main implication of this reduction is that we can regard the complexity of internal priority queues as settled, and just focus on establishing the complexity of sorting. However, Thorup's priority queue is fundamentally an internal data structure; it cannot provide insight for the relationship between sorting and priority queue in external memory.

In Chapter 4 we show that priority queues are almost computationally equivalent to sorting in the I/O model. More precisely, we design an external priority queue that uses the sorting algorithm as a black box, such that the update cost of the priority queue is essentially the same as the per key I/O cost of the sorting algorithm. Let $\log^{()}$ denote the nested logarithmic function, i.e., $\log^{(0)} x = x$ and $\log^{(i)} = \log(\log^{(i-1)} x)$. Given a sorting algorithm that sorts N keys in $N \cdot S(N)/B$ I/Os, our priority queue uses linear space, and supports a sequence of N insertion, deletion and findmin operations in $O(\frac{1}{B}\sum_{i\geqslant 0} S(B\log^{(i)}\frac{N}{B}))$ amortized I/Os per operation. The reduction uses $O(B)$ memory. The tightness of this reduction can be justified in two aspects. First, if the main memory has size $\Omega(B\log^{(c)}\frac{N}{B})$ for any constant c, our priority queue supports all operations with $O(S(N)/B)$ amortized I/Os. Second, if the main memory only has size $\Theta(B)$, our priority queue supports all operations with $O(S(N)/B)$ amortized I/O cost for $S(N) = \Omega(2^{\log^*\frac{N}{B}})$, and $O(S(N)\log^*\frac{N}{B}/B)$ amortized I/O cost for $S(N) = o(2^{\log^*\frac{N}{B}})$. Note that $2^{\log^*\frac{N}{B}} = o(\log^{(i)}\frac{N}{B})$ for any constant i, so it is very unlikely that a sorting algorithm could achieve $S(N) = o(2^{\log^*\frac{N}{B}})$, meaning that our reduction is essentially tight even with $\Theta(B)$ main memory.

1.4 Summary Queries

Range searching is a natural generalization of the dictionary problem. Here, instead of a single key, a query specifies a range $[q_1, q_2]$, and is interested in all keys in the data set that fall into this range. Range queries can be broadly classified into two categories: reporting queries and aggregation queries. The former enumerate all the records from the data set that fall inside the query range, while the latter only produce an aggregate, such as count, sum, average or max (min), of a particular attribute of these records. Many modern business intelligence applications, however, require ad hoc analytical queries with a rapid execution time. Users issuing these analytical queries are interested not in the actual records, but some statistics of them. This has therefore led to extensive research on how to perform aggregation queries efficiently. By constructing a data structure beforehand, aggregation queries can be answered efficiently at query time without going through the actual data records.

However, reporting and aggregation queries provide only two extremes for analyzing the data, by returning either all the records matching the query condition or one (or a few) single-valued aggregates. These simple aggregates are not expressive enough, and data analysts often need more insight into the data distribution. Consider the following queries:

(Q1) In a company's database: What is the distribution of salaries of all employees aged between 30 and 40?

(Q2) In a search engine's query logs: What are the most frequently queried keywords between May 1 and July 1, 2010?

The analyst issuing the query is perhaps not interested in listing all the records in the query range one by one, while probably not happy with a simple aggregate such as average or max, either. What would be nice is some summary on the data, which is more complex than the simple aggregates, yet still much smaller than the raw query results. Some useful summaries include the frequent items, the ϕ-quantiles for, say, $\phi = 0.1, 0.2, \ldots, 0.9$, a sketch (e.g., the Count-Min sketch [22] or the AMS sketch [6]), or some compressed data representations like wavelets. All these summaries are of central importance in massive data analysis, and have been extensively studied for offline and streaming data. Yet, to use the existing algorithms, one still has to first issue a reporting query to retrieve all query results, and then construct the desired summary afterward. This is clearly time-consuming and wasteful.

In Chapter 5, we study the problem of *summary queries* , namely, how to build a data structure so that a summary can be returned in time proportional to the size of the summary itself, not the size of the raw query results. The problem we consider can be

defined more precisely as follows. Let $\mathcal{D}$ be a data set containing N records. Each record $p \in \mathcal{D}$ is associated with a *query attribute* $A_q(p)$ and a *summary attribute* $A_s(p)$, drawing values possibly from different domains. A *summary query* specifies a range constraint $[q_1, q_2]$ on A_q and the data structure returns a summary on the A_s attribute of all records whose A_q attribute is within the range. For example, in the query (Q1) above, A_q is "age" and A_s is "salary". Note that A_s and A_q could be the same attribute, but it is more useful when they are different, as the analyst is exploring the relationship between two attributes. Our goal is to build a data structure on $\mathcal{D}$ so that a summary query can be answered efficiently. As with any data structure problem, the primary measures are the query time and the space the structure uses.

Our results can be summarized as follow. For AMS sketch and wavelets, we build a baseline data structure that has linear size and answers a summary query in $O(s_\varepsilon \log N)$ time, where s_ε is the size of the summary returned. It also works in external memory, with the query cost being $O(\frac{s_\varepsilon}{B} \log N)$ I/Os if $s_\varepsilon \geqslant B$ and $O(\log N / \log(B/s_\varepsilon))$ I/Os if $s_\varepsilon < B$. For quantiles, heavy hitters and the Count-Min sketch, we propose a more delicate data structure that uses linear space and supports summary queries in $O(\log N + s_\varepsilon)$ time. In external memory, the query cost is $O(\log_B N + s_\varepsilon/B)$ I/Os. This resembles the classical B-tree query cost, which includes an $O(\log_B N)$ search cost and an "output" cost of $O(s_\varepsilon/B)$, whereas the output in our case is a summary of size s_ε. This is clearly optimal (in the comparison model). For not-too-large summaries $s_\varepsilon = O(B)$, the query cost becomes just $O(\log_B N)$, the same as that for a simple aggregation query or a lookup on a B-tree.

Chapter 2

Dynamic External Hashing

2.1 Introduction

This chapter deals with the inherent query-insertion tradeoff of external hash tables in the presence of a memory buffer. For proving our lower bounds we make the only requirement that keys must be treated as atomic elements, i.e., they can only be moved or copied between memory and disk in their entirety, and when answering a query, the query algorithm must visit the block (in memory or on disk) that actually contains the key or one of its copies. Such an *indivisibility* assumption is made in most external memory lower bounds, such as sorting, permuting [4], and all the range searching problems [12, 43, 75]. We assume that each machine word consists of $\log U$ bits and each key occupies one machine word (it does not affect our results if an key occupies any constant number of words). A block has B words and the memory stores up to M words. Finally, we comment that our lower bounds do not depend on the size of the hash table, which implies that the hash table cannot do better by consuming more disk space.

Our Results Consider any dynamic hash table that supports insertions in expected amortized t_u I/Os and answers a successful query in expected t_q I/Os on average. We show that if $t_q \leqslant 1 + O(1/B^c)$ for any constant $c > 1$, then we must have $t_u \geqslant 1 - O(1/B^{\frac{c-1}{6}})$. This is only an additive term of $1/B^{\Omega(1)}$ away from how the standard hash table is supporting insertions, which means that buffering is essentially useless in this case. However, if the query cost is relaxed to $t_q \leqslant 1 + O(1/B^c)$ for any constant $0 < c < 1$, we present a simple dynamic hash table that supports insertions in $t_u = O(B^{c-1}) = o(1)$ I/Os (for block sizes $B = \Omega(\log^{1/c} \frac{N}{M})$). For this case we also present a matching lower bound of $t_u = \Omega(B^{c-1})$. Finally for the case $t_q = 1 + \Theta(1/B)$, we show a tight bound of $t_u = \Theta(1)$. Our results are pictorially illustrated in Figure 2.1, from which we see that we now have an almost complete understanding of the entire query-insertion tradeoff, and $t_q = 1 + \Theta(1/B)$ seems to be the sharp boundary separating effective and ineffective buffering. We prove our lower bounds for the three cases above using a unified framework in Section 2.2 and 2.3. The upper bound for the first case is simply the standard hash table following [47]; we give the upper bounds for the other two cases in Section 2.4.

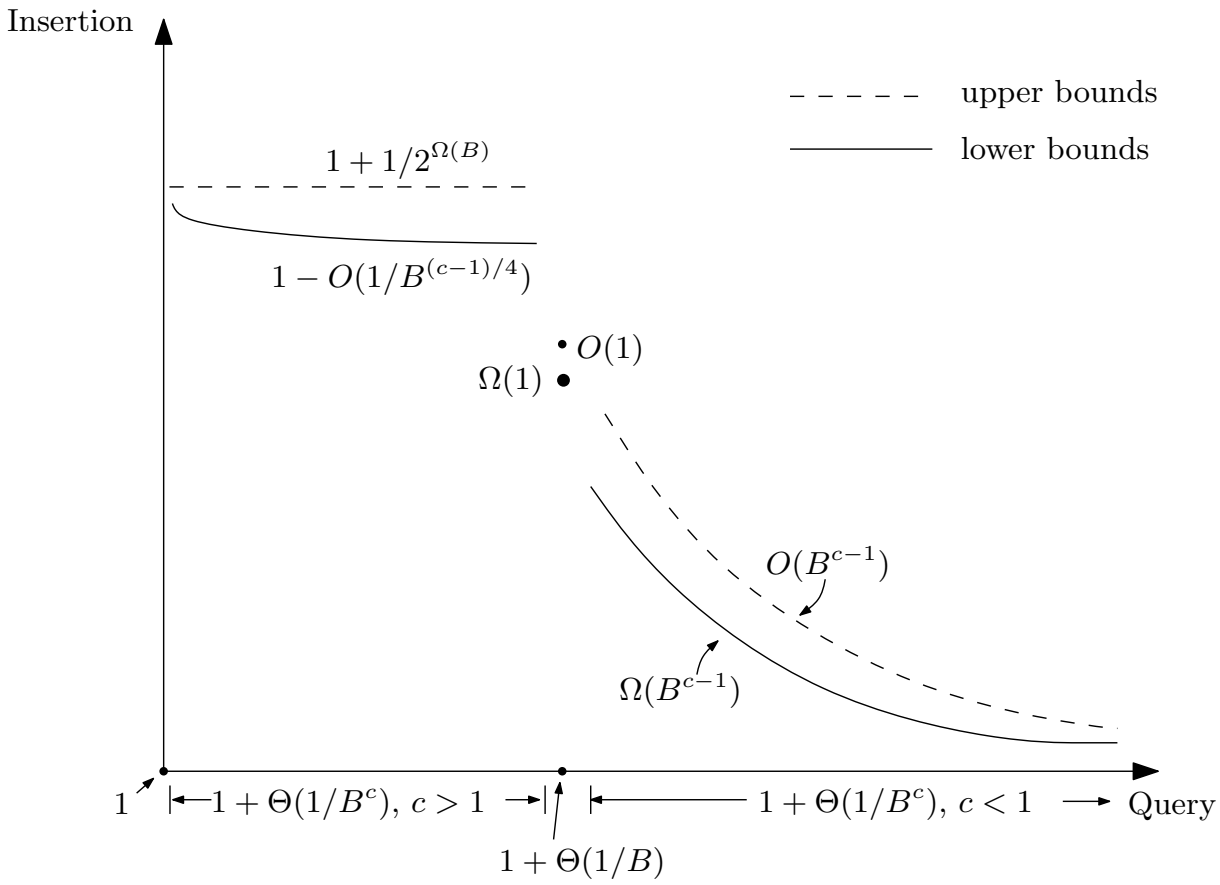

Figure 2.1: The query-insertion tradeoff.

Related results Hash tables are widely used in practice due to their simplicity and excellent performance. Knuth's analysis [47] applies to the basic version where the hash table uses an ideal random hash function and t_q is the expected average cost. Afterward, a lot of works have been done to give better theoretical guarantees, for instance removing the ideal hash function assumption [21], making t_q worst-case [24, 29, 60], etc. Lower bounds have been sparse because in internal memory, the update time cannot be lower than $\Omega(1)$, which is already achieved by the standard hash table. Only with some strong requirements, e.g., when the algorithm is deterministic and t_q is worst-case, can one obtain some nontrivial lower bounds on the update time [24]. Our lower bounds, on the other hand, hold for randomized algorithms and do not need t_q to be worst-case.

As commented earlier, in external memory there is a trivial lower bound of 1 I/O for either a query or an update, if all the changes to the hash table must be committed to disk after each update. However, the vast amount of works in the area of external memory algorithms have never made such a requirement. And indeed for many problems, the availability of a small internal memory buffer can significantly reduce the amortized update cost without affecting the query cost [9, 10, 25, 70]. Unfortunately, little is known

on the inherent limit of what buffering can do. The only nontrivial lower bounds on the update cost of any external data structure with a memory buffer are a paper by Fagerberg and Brodal [17] on the *predecessor* problem and a recent result of Yi [75] on the *range reporting* problem. But the techniques used are inapplicable to our problem. Our result is the first nontrivial lower bound on external hashing. More recently, Verbin and Zhang proved [68] that if t_q is $o(\log_{B \log N} N)$ for both successful and unsuccessful queries, then the amortized update cost has to be at least 0.99 I/Os. This completely confirms Jensen and Pagh's conjecture.

2.2 Lower Bounds

In this section, we prove a lower bound for any *deterministic* hash table under a total of N independent and random insertions, for some sufficiently large N. In Section 2.3 we will extend this lower bound to randomized hash table. We will derive a lower bound on t_u, the expected amortized number of I/Os for an insertion, while assuming that the hash table is able to answer a successful query in t_q I/Os on average in expectation after the first i keys have been inserted, for all $i = 1, \ldots, N$. We assume that all the keys are different, which happens with probability $1 - O(1/N)$ as long as $U > N^3$ by the birthday paradox. Under this setting we obtain the following tradeoffs between t_q and t_u.

Theorem 2.1 *For any constant $c > 0$, suppose we insert a sequence of $N > \Omega\left(M \log U \cdot B^{2c}\right)$ random keys into an initially empty hash table. If the total cost of these insertions is expected $N \cdot t_u$ I/Os, and the hash table is able to answer a successful query in expected average t_q I/Os at any time, then the following tradeoffs hold:*

1. *If $t_q \leqslant 1 + O(1/B^c)$ for any $c > 1$, then $t_u \geqslant 1 - O(1/B^{\frac{c-1}{4}})$;*
2. *If $t_q \leqslant 1 + O(1/B)$, then $t_u \geqslant \Omega(1)$;*
3. *If $t_q \leqslant 1 + O(1/B^c)$ for any $0 < c < 1$, then $t_u \geqslant \Omega(B^{c-1})$.*

The abstraction To abstractly model a dynamic hash table, we ignore any of its auxiliary structures but only focus on the layout of keys. Consider any snapshot of the hash table when we have inserted k keys. We divide these k keys into three zones. The *memory zone* $\mathcal{M}$ is a set of at most M keys that are kept in memory. It takes no I/O to query any key in $\mathcal{M}$. All keys not in $\mathcal{M}$ must reside on disk. Denote all the blocks on disk by $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_d$. Each $\mathcal{B}_i$ is a set of at most B keys, and it is possible that one key appears in more than one $\mathcal{B}_i$. Let $f : U \to \{1, \ldots, d\}$ be any function computable within memory, and we divide the disk-resident keys into two zones with respect to f and the set of blocks

$\mathcal{B}_1, \ldots, \mathcal{B}_d$. The *fast zone* $\mathcal{F}$ contains all keys x such that $x \in \mathcal{B}_{f(x)}$: These are the keys that are accessible with just one I/O. We allocate all the remaining keys into the *overflow zone* $\mathcal{O}$: These keys need at least two I/Os to locate. Note that under random inputs, the sets $\mathcal{M}, \mathcal{F}, \mathcal{O}, \mathcal{B}_1, \ldots, \mathcal{B}_d$ are all random sets, which the hash table will adaptively choose after seeing each random insertion. Changing $\mathcal{M}$ is free, but changing any $\mathcal{B}_i$ will cost 1 I/O.

Any query algorithm on the hash table can be modeled as described, since the only way to find a queried key in one I/O is to compute the index of a block containing x with only the information in memory. If the memory-resident computation gives an incorrect address or anything else, at least 2 I/Os will be necessary. Because any such f must be computable within memory, and the memory has $M \log U$ bits, the hash table can employ a family $\mathcal{H}$ of at most $2^{M \log U}$ distinct f's. Note that the current f adopted by the hash table is dependent upon the already inserted keys, but the family $\mathcal{H}$ has to be fixed beforehand.

Suppose the hash table answers a successful query with an expected average cost of $t_q = 1 + \delta$ I/Os, where $\delta = 1/B^c$ for some constant $c > 0$. Consider the snapshot of the hash table when k keys have been inserted. Then we must have $\mathbf{E}[|\mathcal{F}| + 2 \cdot |\mathcal{O}|]/k \leqslant 1 + \delta$. Since $|\mathcal{F}| + |\mathcal{O}| = k - |\mathcal{M}|$ and $\mathbf{E}[|\mathcal{M}|] \leqslant M$, we have

$$\mathbf{E}[|\mathcal{O}|] \leqslant M + \delta k. \tag{2.1}$$

We also have the following high-probability version of (5.1).

Lemma 2.2 *Let $\phi \geqslant 1/B^{(c-1)/4}$ and let $k \geqslant \phi N$. At the snapshot when k keys have been inserted, with probability at least $1 - 2\phi$, $|\mathcal{O}| \leqslant M + \frac{\delta}{\phi} k$.*

Proof: On this snapshot the hash table answers a query in expected average $1 + \delta$ I/Os. We claim that with probability at most 2ϕ, the average query cost is more than $1 + \delta/\phi$. Otherwise, since in any case the average query cost is at least $1 - M/k$ (assuming all keys not in memory need just one I/O), we would have an expected average cost of at least

$$(1 - 2\phi)(1 - M/k) + 2\phi \cdot (1 + \delta/\phi) > 1 + \delta,$$

provided that $\frac{N}{M} > \frac{1}{\phi\delta}$, which is valid since we assume that $\frac{N}{M} > B^{2c} \log U$. The lemma then follows from the same argument used to derive (5.1). □

Basic idea of the lower bound proof For the first ϕN keys, we ignore the cost of their insertions. Consider any $f : U \to \{1, \ldots, d\}$. For $i = 1, \ldots, d$, let $\alpha_i = |f^{-1}(i)|/U$, and we call $(\alpha_1, \ldots, \alpha_d)$ the *characteristic vector* of f. Note that $\sum_i \alpha_i = 1$. For any one of the first ϕN keys, since it is randomly chosen from U, f will direct it to $\mathcal{B}_i$ with

probability α_i. Intuitively, if α_i is large, too many keys will be directed to $\mathcal{B}_i$. Since $\mathcal{B}_i$ contains at most B keys, the extra keys will have to be pushed to the overflow zone. If there are too many large α_i's, $\mathcal{O}$ will be large enough to violate the query requirement. Thus, the hash table should use an f that distributes keys relatively evenly to the blocks. However, if f evenly distributes the first ϕN keys, it is also likely to distribute newly inserted keys evenly, leading to a high insertion cost. Below we formalize this intuition.

For the first tradeoff of Theorem 2.1, we set $\delta = 1/B^c$. We also pick the following set of parameters $\phi = 1/B^{(c-1)/4}, \rho = 2B^{(c+3)/4}/N, s = N/B^{(c+1)/2}$. We will use different values for these parameters when proving the other two tradeoffs. Given an f with characteristic vector $(\alpha_1, \ldots, \alpha_d)$, let $D^f = \{i \mid \alpha_i > \rho\}$ be the collection of block indices with large α_i's. We say that the indices in D^f form the *bad index area* and others form the *good index area*. Let $\lambda_f = \sum_{i \in D^f} \alpha_i$. Note that there are at most λ_f/ρ indices in the bad index area. We call an f with $\lambda_f > \phi$ a *bad function*; otherwise it is a *good function*. The following lemma shows that with high probability, the hash table has to use a good function f from $\mathcal{H}$.

Lemma 2.3 *At the snapshot when k keys are inserted for any $k \geqslant \phi N$, the function f used by the hash table is a good function with probability at least $1 - 2\phi - 1/2^{\Omega(B)}$.*

Proof: Consider any bad function f from $\mathcal{H}$. Let X_j be the indicator variable of the event that the j-th inserted key is mapped to the bad index area, $j = 1, \ldots, k$. Then $X = \sum_{j=1}^{k} X_j$ is the total number of keys mapped to the bad index area of f. We have $\mathbf{E}[X] = \lambda_f k$. By the Chernoff bound, we have

$$\Pr\left[X < \frac{2}{3}\lambda_f k\right] \leqslant e^{-\frac{(1/3)^2 \lambda_f k}{2}} \leqslant e^{-\frac{\phi^2 N}{18}},$$

namely with probability at least $1 - e^{-\frac{\phi^2 N}{18}}$, we have $X \geqslant \frac{2}{3}\lambda_f k$. Since the family $\mathcal{H}$ contains at most $2^{M \log U}$ bad functions, by a union bound we know that with probability at least $1 - 2^{M \log U} \cdot e^{-\frac{\phi^2 N}{18}} \geqslant 1 - 1/2^{\Omega(B)}$ (by the parameters chosen and the assumption that $N > \Omega(MB^{2c} \log U)$), for all the bad functions in $\mathcal{H}$, we have $X \geqslant \frac{2}{3}\lambda_f k$.

Consequently, since the bad index area can only accommodate $B \cdot \lambda_f/\rho$ keys in the fast zone, at least $\frac{2}{3}\lambda_f k - B\lambda_f/\rho$ cannot be in the fast zone. The memory zone can accept at most M keys, so the number of keys in the overflow zone is at least

$$|\mathcal{O}| \geqslant \frac{2}{3}\lambda_f k - B\lambda_f/\rho - M > M + \frac{\delta}{\phi}k.$$

This happens with probability at least $1 - 1/2^{\Omega(B)}$, due to the fact that f is a bad function. On the other hand, Lemma 2.2 states that $|\mathcal{O}| \leqslant M + \frac{\delta}{\phi}k$ holds with probability at least $1 - 2\phi$, thus by a union bound the hash table has to use a good function with probability at least $1 - 2\phi - 1/2^{\Omega(B)}$. □

A bin-ball game Lemma 2.3 enables us to consider only those good functions f after the initial ϕN insertions. To show that any good function will incur a large insertion cost, we first consider the following bin-ball game, which captures the essence of performing insertions using a good function.

In an (s, p, t) *bin-ball game*, we throw s balls into r (for any $r \geqslant 1/p$) bins independently at random, and the probability that any ball goes to any particular bin is no more than p. At the end of the game, an adversary removes t balls from the bins such that the remaining $s - t$ balls hit the least number of bins. The cost of the game is defined as the number of nonempty bins occupied by the $s - t$ remaining balls.

We have the following two results with respect to such a game, depending on the relationships among s, p, and t.

Lemma 2.4 *If $sp \leqslant \frac{1}{3}$, then for any $\mu > 0$, with probability at least $1 - e^{-\frac{\mu^2 s}{3}}$, the cost of an (s, p, t) bin-ball game is at least $(1 - \mu)(1 - sp)s - t$.*

Proof: Imagine that we throw the s balls one by one. Let X_j be the indicator variable denoting the event that the j-th ball is thrown into an empty bin. The number of nonempty bins in the end is thus $X = \sum_{j=1}^{s} X_j$. These X_j's are not independent, but no matter what has happened previously for the first $j - 1$ balls, we always have $\Pr[X_j = 0] \leqslant sp$. This is because at any time, at most s bins are nonempty. Let Y_j $(1 \leqslant j \leqslant s)$ be a set of independent variables such that

$$Y_i = \begin{cases} 0, & \text{with probability } sp; \\ 1, & \text{otherwise.} \end{cases}$$

Let $Y = \sum_{j=1}^{s} Y_j$. Each Y_i is stochastically dominated by X_i, so Y is stochastically dominated by X. We have $\mathbf{E}[Y] = (1 - sp)s$ and we can apply the Chernoff bound on Y:

$$\Pr\left[Y < (1 - \mu)(1 - sp)s\right] < e^{-\frac{\mu^2(1-sp)s}{2}} < e^{-\frac{\mu^2 s}{3}}.$$

Therefore with probability at least $1 - e^{-\frac{\mu^2 s}{3}}$, we have $X \geqslant (1 - \mu)(1 - sp)s$. Finally, since removing t balls will reduce the number of nonempty bins by at most t, the cost of the bin-ball game is at least $(1 - \mu)(1 - sp)s - t$ with probability at least $1 - e^{-\frac{\mu^2 s}{3}}$. □

Lemma 2.5 *If $s/2 \geqslant t$ and $s/2 \geqslant 1/p$, then with probability at least $1 - 1/2^{\Omega(s)}$, the cost of an (s, p, t) bin-ball game is at least $1/(20p)$.*

Proof: In this case, the adversary will remove at most $s/2$ balls in the end. Thus we show that with very small probability, there exist a subset of $s/2$ balls all of which are thrown into a subset of at most $1/(20p)$ bins. Before the analysis, we merge bins such

that the probability that any ball goes to any particular bin is between $p/2$ and p, and consequently, the number of of bins would be between $1/p$ to $2/p$. Note that such an operation will only make the cost of the bin-ball game smaller. Now this probability is at most

$$\sum_{i=1}^{1/(20p)} \left(\binom{2/p}{i} \binom{s}{s/2} \left(\frac{i}{1/p} \right)^{s/2} \right) \leqslant 2 \binom{2/p}{1/(20p)} \binom{s}{s/2} \left(\frac{1/(20p)}{1/p} \right)^{s/2} \leqslant 1/2^{\Omega(s)},$$

hence the lemma. □

Now we are ready to prove the main theorem.

Proof of Theorem 2.1 **Proof**: We begin with the first tradeoff. Recall that we use the following parameters: $\delta = 1/B^c$, $\phi = 1/B^{(c-1)/4}$, $\rho = 2B^{(c+3)/4}/N$, $s = N/B^{(c+1)/2}$. For the first ϕN keys, we do not count their insertion costs. We divide the rest of the insertions into rounds, with each round containing s keys. We now bound the expected cost of each round.

Focus on a particular round, and let f be the function used by the hash table at the end of this round. We only consider the set $\mathcal{R}$ of keys inserted in this round that are mapped to the good index area of f, i.e., $\mathcal{R} = \{x \mid f(x) \notin D^f\}$; other keys are assumed to have been inserted for free. Consider the block with index $f(x)$ for a particular x. If x is in the fast zone, the block $\mathcal{B}_{f(x)}$ must contain x. Thus, the number of distinct indices $f(x)$ for $x \in \mathcal{R} \cap \mathcal{F}$ is an obvious lower bound on the I/O cost of this round. Denote this number by $Z = |\{f(x) \mid x \in \mathcal{R} \cap \mathcal{F}\}|$. Below we will show that Z is large with high probability.

We first argue that at the end of this round, each of the following three events happens with high probability.

- $\mathcal{E}_1$: $|\mathcal{O}| \leqslant \delta N/\phi + M$;
- $\mathcal{E}_2$: f is a good function;
- $\mathcal{E}_3$: For all good functions $f \in \mathcal{H}$ and their corresponding overflow zones $\mathcal{O}$ and memory zones $\mathcal{M}$, $Z \geqslant (1 - O(\phi))s - t$, where $t = |\mathcal{O}| + |\mathcal{M}|$.

By Lemma 2.2, $\mathcal{E}_1$ happens with probability at least $1-2\phi$. By Lemma 2.3, $\mathcal{E}_2$ happens with probability at least $1 - 2\phi - 1/2^{\Omega(B)}$. It remains to show that $\mathcal{E}_3$ also happens with high probability.

We prove so by first claiming that for a particular good function $f \in \mathcal{H}$, with probability at least $1 - e^{-2\phi^2 s}$, Z is at least the cost of a $((1-2\phi)s, \frac{\rho}{1-\lambda_f}, t)$ bin-ball game. This is because of the following reasons:

1. Since f is a good function, by the Chernoff bound, with probability at least $1-e^{-2\phi^2 s}$, more than $(1-2\phi)s$ newly inserted keys will fall into the good index area of f, i.e., $|\mathcal{R}| > (1-2\phi)s$.

2. The probability of any key being mapped to any index in the good index area, conditioned on that it goes to the good index area, is no more than $\frac{\rho}{1-\lambda_f}$.

3. Only t keys in $\mathcal{R}$ are not in the fast zone $\mathcal{F}$, excluding them from $\mathcal{R}$ corresponds to discarding t balls at the end of the bin-ball game.

Thus by Lemma 2.4 (setting $\mu = \phi$), with probability at least $1 - e^{-\frac{\phi^2\cdot(1-2\phi)s}{3}} - e^{-2\phi^2 s}$, we have

$$\begin{aligned} Z &\geqslant (1-\phi)\left(1-(1-2\phi)s\cdot\frac{\rho}{1-\lambda_f}\right)(1-2\phi)s - t \\ &\geqslant (1-\phi)\left(1-(1-2\phi)s\cdot\frac{\rho}{1-\phi}\right)(1-2\phi)s - t \\ &\geqslant (1-O(\phi))\,s - t. \end{aligned}$$

Thus by applying a union bound on all good functions in $\mathcal{H}$, $\mathcal{E}_3$ happens with probability at least $1 - (e^{-\frac{\phi^2\cdot(1-2\phi)s}{3}} + e^{-2\phi^2 s})\cdot 2^{M\log U} = 1 - 2^{-\Omega(B)}$ (by the assumption $N > \Omega(MB^{2c}\log U)$).

Now we lower bound the expected insertion cost of one round. By a union bound, with probability at least $1 - O(\phi) - 1/2^{\Omega(B)}$, all of $\mathcal{E}_1, \mathcal{E}_2$, and $\mathcal{E}_3$ happen at the end of the round. By $\mathcal{E}_2$ and $\mathcal{E}_3$, we have $Z \geqslant (1-O(\phi))\,s - t$. Since now $t = |\mathcal{O}| + |\mathcal{M}| \leqslant \delta N/\phi + 2M = O(\phi s)$ by $\mathcal{E}_1$, we have $Z \geqslant (1-O(\phi))\,s$. Thus the expected cost of one round will be at least

$$(1-O(\phi))\,s\cdot\left(1-O(\phi)-1/2^{\Omega(B)}\right) = (1-O(\phi))\,s.$$

Finally, since there are $(1-\phi)N/s$ rounds, the expected amortized cost per insertion is at least

$$(1-O(\phi))\,s\cdot(1-\phi)N/s\cdot 1/N = 1 - O\left(1/B^{\frac{c-1}{4}}\right).$$

For the second tradeoff, we choose the following set of parameters: $\phi = 1/\kappa, \rho = 2\kappa B/N, s = N/(\kappa^2 B)$ and $\delta = 1/(\kappa^4 B)$ (for some constant κ large enough). We can check that Lemma 2.3 still holds with these parameters, and then go through the proof above. We omit the tedious details. Plugging the new parameters into the derivations we obtain a lower bound $t_u \geqslant \Omega(1)$.

For the third tradeoff, we choose the following set of parameters: $\phi = 1/8, \rho = 16B/N, s = 32N/B^c$ and $\delta = 1/B^c$. We can still check the validity of Lemma 2.3, and go through the whole proof. The only difference is that we need to use Lemma 2.5

in place of Lemma 2.4, the reason being that for our new set of parameters, we have $s\rho = \omega(1)$ thus Lemma 2.4 does not apply. By using Lemma 2.5 we can lower bound the expected insertion cost of each round by $\Omega\left((1-2\phi)/(20\rho)\right)$, so the expected amortized insertion cost is at least

$$\Omega\left(\frac{1-2\phi}{20\rho}\right) \cdot (1-\phi)N/s \cdot 1/N = \Omega(B^{c-1}),$$

as claimed. □

2.3 Lower Bounds for Randomized Hash Tables

In this section we show how to extend our lower bound to randomized hash tables. We follow the framework of Yao [74]. A randomized hash table can be viewed as a probability distribution $P_{\mathcal{A}}$ over the set $\mathcal{A}$ of all deterministic hash tables. We still consider the tradeoff between the expected average cost of a successful query t_q and the expected amortized insertion cost t_u. Now the expectation is with respect to the probability distribution $P_{\mathcal{A}}$. More precisely, let $Q(A, I, t)$ denote the average query cost over all keys in the deterministic hash table $A \in \mathcal{A}$, on input sequence I at snapshot t, and $U(A, I)$ denote the I/O cost per key of inserting all keys in I using A, then the expected average query cost and expected amortized insertion cost of a randomized hash table $P_{\mathcal{A}}$ can be expressed as $t_q = \max_I \max_t \mathbf{E}_{P_{\mathcal{A}}}[Q(A, I, t)]$ and $t_u = \max_I \mathbf{E}_{P_{\mathcal{A}}}[U(A, I)]$, respectively. For randomized hash tables we have the following tradeoffs:

Theorem 2.6 *For any randomized hash table, suppose we insert a sequence of $N > \Omega(M \log U \cdot B^{2c})$ keys into it. If the total cost of these insertions is expected at most $N \cdot t_u$ I/Os under any input, and the hash table is able to answer a successful query in expected average t_q I/Os at any time, then the following tradeoffs hold:*

1. *If $t_q \leqslant 1 + O(1/B^c)$ for any $c > 1$, then $t_u \geqslant 1 - O(1/B^{\frac{c-1}{6}})$;*
2. *If $t_q \leqslant 1 + O(1/B)$, then $t_u \geqslant \Omega(1)$;*
3. *If $t_q \leqslant 1 + O(1/B^c)$ for any $0 < c < 1$, then $t_u \geqslant \Omega(B^{c-1})$.*

Proof: For the first tradeoff, we set the parameters as follows: $\phi = 1/B^{(c-1)/6}, \rho = 2B^{(c+5)/6}, s = N/B^{(c+2)/3}$. Assuming the query cost

$$\max_I \max_t \mathbf{E}_{P_{\mathcal{A}}}[Q(A, I, t)] \leqslant 1 + O(1/B^c)$$

for any $c > 1$, we will derive a lower bound for the insertion cost $\max_I \mathbf{E}_{P_{\mathcal{A}}}[Q(A, I)]$. Let $P_{\mathcal{I}}$ denote the uniform distribution over the set $\mathcal{I}$ of all input sequences of length N.

Considering $\frac{1}{N}\sum_{t=1}^{N}\mathbf{E}_{P_{\mathcal{I}},P_{\mathcal{A}}}[Q(A,I,t)]$, we have the following bound:

$$\begin{aligned}\frac{1}{N}\sum_{t=1}^{N}\mathbf{E}_{P_{\mathcal{I}},P_{\mathcal{A}}}[Q(A,I,t)] &= \frac{1}{N}\sum_{t=1}^{N}\mathbf{E}_{P_{\mathcal{I}}}\left[\mathbf{E}_{P_{\mathcal{A}}}[Q(A,I,t)]\right]\\ &\leqslant \frac{1}{N}\sum_{t=1}^{N}\mathbf{E}_{P_{\mathcal{I}}}[1+O(1/B^c)]\\ &\leqslant 1+O(1/B^c).\end{aligned}$$

Consider any $t \geqslant \phi N$. Since $Q(A,I,t) \geqslant 1 - M/\phi N$ for all A and I, it follows that with probability at least $1-\phi$, the (deterministic) hash table A chosen according to $P_{\mathcal{A}}$ satisfies

$$\frac{1}{N}\sum_{t=1}^{N}\mathbf{E}_{P_{\mathcal{I}}}[Q(A,I,t)] \leqslant 1+O\left(\frac{1}{\phi B^c}\right),$$

by the parameters chosen above and for N large enough. We will prove that for these hash tables, the insertion cost is large. Fixing such a hash table A, it is easy to show that A satisfies $\mathbf{E}_{P_{\mathcal{I}}}[Q(A,I,t)] \leqslant 1+O\left(\frac{1}{\phi^2 B^c}\right)$ on at least $(1-2\phi)N$ snapshots. Call these snapshots *good snapshots*. We can check that Lemma 2.2 and Lemma 2.3 still hold on any good snapshot. Ignoring the first ϕN insertions, we divide the remaining $(1-\phi)N$ insertions into rounds, with each round containing exactly s good snapshots and also ending with a good one. Focusing on a particular round, we only consider the insertion cost of the s keys inserted right before the good snapshots. Since the ending snapshot of the round is good, using the same argument as in Theorem 2.1 we can prove that the cost inserting the s keys is at least $(1-O(\phi))s$. So the total insertion cost of each round is at least $(1-O(\phi))s$. Since there are $(1-2\phi)N/s$ rounds, the expected amortized cost per insertion of A is $\mathbf{E}_{P_{\mathcal{I}}}[U(A,I)] \geqslant (1-O(\phi))s\cdot(1-2\phi)N/s\cdot 1/N = 1-O(\phi)$. For the randomized hash table $P_{\mathcal{A}}$, since with probability $\geqslant 1-\phi$, A is one for which the above analysis goes through, we can bound the expected amortized insertion cost as follows:

$$\begin{aligned}\max_I \mathbf{E}_{P_{\mathcal{A}}}[U(A,I)] &\geqslant \mathbf{E}_{P_{\mathcal{A}},P_{\mathcal{I}}}[U(A,I)]\\ &\geqslant (1-\phi)(1-O(\phi))\\ &\geqslant 1-O(1/B^{\frac{c-1}{6}}).\end{aligned}$$

For the second and third tradeoffs, we choose the same parameters as in the proof of Theorem 2.1, and a similar argument will yield the desired results. □

2.4 Upper Bounds

In this section, we present some upper bounds on the query-insertion tradeoff of external hash tables, showing that all three lower bound tradeoffs of Theorem 2.1 are essentially

tight. The first tradeoff is matched by the standard external hash table, up to an additive term of $1/B^{\Omega(1)}$. Below we give matching (up to constant factors) upper bounds for the other two tradeoffs.

Specifically, we present a simple dynamic hash table that, for any constant $0 < c \leqslant 1$, supports insertions in $t_u = O(B^{c-1} + \frac{\log(N/M)}{B})$ I/Os amortized, while being able to answer a query in expected $t_q = 1 + O(1/B^c)$ I/Os on average. This means that our lower bound is tight for all block sizes $B = \Omega(\log^{1/c} \frac{N}{M})$. In the following we first state a folklore result by applying the *logarithmic method* [15] to a standard hash table [47], then we show how to improve the query cost to $1 + O(1/B^c)$ while keeping the insertion cost low.

Applying the logarithmic method Fix a parameter $\gamma \geqslant 2$. We maintain a series of hash tables $H_0, H_1, \ldots$. The hash table H_k has $\gamma^k \cdot \frac{M}{B}$ buckets and stores up to $\frac{1}{2}\gamma^k M$ keys, so that its load factor is always at most $\frac{1}{2}$. We use some standard method to resolve collisions, such as chaining. The first hash table H_0 always resides in memory while the rest stay on disk.

When a new key is inserted, it always goes to the memory-resident H_0. When H_0 is full (i.e., having $\frac{1}{2}M$ keys), we migrate all keys stored in H_0 to H_1. If H_1 is not empty, we simply merge the corresponding buckets. Note that each bucket in H_0 corresponds to γ consecutive buckets in H_1, and we can easily distribute the keys to their new buckets in H_1 by scanning the two tables in parallel, costing $O(\gamma \cdot \frac{M}{B})$ I/Os. This operation takes place inductively: Whenever H_k is full, we migrate its keys to H_{k+1}, costing $O(\gamma^{k+1} \cdot \frac{M}{B})$ I/Os. Then standard analysis shows that for N insertions, the total cost is $O(\frac{\gamma N}{B} \log \frac{N}{M})$ I/Os, or $O(\frac{\gamma}{B} \log \frac{N}{M})$ amortized I/Os per insertion. However, for a query we need to examine all the $O(\log_\gamma \frac{N}{M})$ hash tables.

Lemma 2.7 *For any parameter $\gamma \geqslant 2$, there is a dynamic hash table that supports an insertion in amortized $O(\frac{\gamma}{B} \log \frac{N}{M})$ I/Os and a successful query in expected $O(\log_\gamma \frac{N}{M})$ I/Os.*

Improving the query cost Next we show how to improve the average cost of a successful query to $1 + O(1/B^c)$ I/Os for any constant $0 < c \leqslant 1$, while keeping the insertion cost low. The idea is to try to put the majority of the keys into one single big hash table. In the standard logarithmic method described above, the last table may seem a good candidate, but sometimes it may only contain a constant fraction of all keys. Below we show how to bootstrap the structure above to obtain a better query bound.

Fix a parameter $2 \leqslant \beta \leqslant B$. For the first M keys inserted, we dump them in a hash table $\widehat{H}$ on disk. Then run the algorithm of Lemma 2.7 for the next M/β keys. After that we merge these M/β keys into $\widehat{H}$. We keep doing so until the size of $\widehat{H}$ has reached $2M$,

and then we start the next round. Generally, in the i-th round, the size of $\widehat{H}$ goes from $2^{i-1}M$ to 2^iM, and we apply the algorithm of Lemma 2.7 for every $2^{i-1}M/\beta$ keys. It is clear that $\widehat{H}$ always has at least a fraction of $1-\frac{1}{\beta}$ of all the keys inserted so far, while the series of hash tables used in the logarithmic method maintain at least a separation factor of 2 in the sizes between successive tables. Thus, the expected average query cost is at most

$$\left(1+1/2^{\Omega(B)}\right)\left(1\cdot\left(1-\frac{1}{\beta}\right)+\frac{1}{\beta}\left(2\cdot\frac{1}{2}+3\cdot\frac{1}{4}+\cdots\right)\right)=1+O(1/\beta)).$$

Next we analyze the amortized insertion cost. Since the number of keys doubles every round, it is (asymptotically) sufficient to analyze the last round. In the last round, $\widehat{H}$ is scanned β times, and we charge $O(\beta/B)$ I/Os to each of the N keys. The algorithm of Lemma 2.7 is invoked β times, but every invocation handles $O(N/\beta)$ different keys, so the amortized cost per key is still $O(\frac{\gamma}{B}\log\frac{N}{M})$ I/Os. Thus the total amortized cost per insertion is $O(\frac{1}{B}(\beta+\gamma\log\frac{N}{M}))$ I/Os. Then setting $\beta=B^c$ and $\gamma=2$ yields the desired results.

Theorem 2.8 *For any constant $0<c\leqslant 1$, there is a dynamic hash table that supports an insertion in amortized $O(B^{c-1}+\frac{\log(N/M)}{B})$ I/Os and a successful query in expected average $1+O(1/B^c)$ I/Os.*

Chapter 3

Cache-Oblivious Hashing

3.1 Introduction

In this chapter we study hash tables in the cache-oblivious model. Recall that in the cache-oblivious model, the hash table is unaware of the block size B, so the hash function can not be used to index blocks. Our goal is to lay out a hash table such that its search cost matches its cache-aware version, i.e., $1 + 1/2^{\Omega(B)}$ I/Os, for all block sizes B.

Our results A straightforward way of making the hash table cache-oblivious is to simply use linear probing but ignoring the blocking altogether[1]. One would expect it to work well irrespective of the block size since it uses only sequential probes. However, in Section 3.2 we show that its search cost is $1 + O(\alpha/B)$ I/Os even assuming a truly random hash function. In fact, we also derive the constant in the big-Oh, which depends on C_N and C'_N, the expected average number of probes for a successful and unsuccessful lookup, respectively. Recall that Under such truly random hash function assumption, Knuth [47] showed that

$$\begin{aligned} C_N &\approx \tfrac{1}{2}\left(1 + \tfrac{1}{1-\alpha}\right) && \text{(successful lookup);} \\ C'_N &\approx \tfrac{1}{2}\left(1 + \left(\tfrac{1}{1-\alpha}\right)^2\right) && \text{(unsuccessful lookup).} \end{aligned}$$

This is worse than its cache-aware version that is particularly tuned to work with a single B. The gap is in some sense exponential, if we are concerned with the fraction of keys that cannot be found with a single I/O (note that an average search cost of $t_q = 1+\varepsilon$ means that at most a fraction of ε keys need two or more I/Os).

Next, we explore other collision resolution strategies to see if they work better in the cache-oblivious model. In Section 3.3, we show that the *blocked probing* algorithm [59] achieves the desired $1 + 1/2^{\Omega(B)}$ search cost, but under the following two conditions: (a) B is a power of 2; and (b) every block starts at a memory address divisible by B. In addition, we have analyzed the performance of blocked probing when the hash function has limited independence: We show that with a k-wise independent hash function, the expected I/O cost of a search is $1 + O\left(\left(\frac{k-1}{e^{2/3}(1-\alpha)^2 B}\right)^{(k-1)/2}\right)$. Since a k-wise independent

[1] Chaining would perform worse cache-obliviously because the list associated with each position is not laid out consecutively.

hash function is also k'-wise independent for all $k' \leqslant k$, as long as $k \geqslant (1-\alpha)^2 B + 1$, the bound becomes $1 + 2^{-\Omega((1-\alpha)^2 B)} = 1 + 2^{-\Omega(B)}$, matching that of a truly random hash function.

Note that neither of the two conditions above is stated in the cache-oblivious model, but they indeed hold on all real machines. This raises the theoretical question of whether $1 + 1/2^{\Omega(B)}$ is achievable in the "true" cache-oblivious model. In Section 3.4, we show that neither condition is dispensable. Specifically, we prove that if the hash table is only required to work for a single B but an arbitrary shift of the layout, or if (B) holds but the hash table is required to work for all B, then the best obtainable search cost is $1 + O(\alpha/B)$ I/Os, which exactly matches what linear probing achieves. Our lower bound model allows a truly random hash function to be used and puts no restrictions on the structure of the hash table, except that each key is treated as an atomic element, known as the indivisibility assumption.

Related results Hashing has been well studied in the I/O model. The $1 + 1/2^{\Omega(B)}$ search cost holds as long as the load factor α is bounded away from 1 [47], and there are various techniques in the database literature to keep the load factor in a desired range, such as extensible hashing or linear hashing, as previously mentioned.

The cache-oblivious model was proposed by Frigo et al., which introduces a clean and elegant way to modeling memory hierarchies. Since then, cache-oblivious algorithms and data structures have received a lot of attention, and most fundamental problems have been solved. For example, cache-oblivious sorting takes $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os [30], and a cache-oblivious B-tree takes $O(\log_B N)$ I/Os for a search [13]. Please see the survey [23] for other results. In most cases, the cache-oblivious bounds match their cache-aware versions, and it has always be an interesting problem to see for what problems do we have a separation between the cache-oblivious model and the cache-aware model. Until today there have been only three separation results [2, 14, 19]; our lower bound adds to that list, furthering our understanding of cache-obliviousness.

3.2 Analysis of Linear Probing in the Cache-Oblivious Model

Linear probing while ignoring the blocking is naturally cache-oblivious. In this section we analyze its search I/O cost, which turns out to delicately depend on C_N and C'_N, the expected number of probes in a successful and unsuccessful query, respectively. Note that the equalities in the theorem below are exact, though we only know the asymptotic formulas for C_N and C'_N.

Theorem 3.1 *Suppose the linear probing algorithm uses a truly random hash function h. Let CO_N and CO'_N denote the expected average number of I/Os for a successful and an unsuccessful query, respectively. For any block size B, we have*

$$CO_N = 1 + (C_N - 1)/B;$$
$$CO'_N = 1 + (C'_N - 1)/B.$$

Proof: Let R be the size of the hash table, which is divided into R/B blocks $\mathcal{B}_0, \ldots, \mathcal{B}_{R/B-1}$ (assuming that R is a multiple of B for simplicity). The block $\mathcal{B}_l$ spans positions $lB, lB+1, \ldots, lB+B-1$. Consider an unsuccessful search for a key x. Define $p(i,j)$, $i \neq j$, to be the event that the hash table has positions i through j occupied (wrapping around when necessary). Note that the number of occupied positions is N, so $p(i,j) = 0$ for any $j \notin \{i, i+1, \ldots, i+N-1\}$ (wrapping around when necessary). By the circular symmetry of linear probing and the uniform hash function assumption, $p(0,k)$ is exactly the probability that an unsuccessful search for a key x takes at least $k+2$ probes. Thus we have

$$C'_N = 1 + \sum_{k=0}^{N-1} p(0,k). \tag{3.1}$$

Let p_k be the probability that an unsuccessful search takes at least $k+1$ I/Os. Below we will relate p_k with the $p(0,k)$'s. By the uniformity of the hash function h, we assume that $h(x)$ lies in the first block. Note that for a insertion to cost at least $k+1$ I/Os, positions $h(x)$ through $kB-1$ must be occupied. Since $h(x)$ hits position 0 through $B-1$ with same probability, we have

$$\begin{aligned} p_k &= \frac{1}{B}\sum_{i=0}^{B-1} p(i, kB-1) \\ &= \frac{1}{B}\sum_{i=0}^{B-1} p(0, kB-i-1) \qquad \text{(Since } h \text{ is a truly random function.).} \end{aligned}$$

Now we can compute CO'_N as follows:

$$\begin{aligned} CO'_N &= 1 + \sum_{k=1}^{N/B} p_k \\ &= 1 + \sum_{k=1}^{N/B} \frac{1}{B} \sum_{i=0}^{B-1} p(0, kB - i - 1) \\ &= 1 + \frac{1}{B} \sum_{j=0}^{N-1} p(0, j) \\ &= 1 - \frac{1}{B} + \frac{1}{B} \left(1 + \sum_{j=0}^{N-1} p(0, j) \right) \\ &= 1 - \frac{1}{B} + \frac{C'_N}{B}. \end{aligned}$$

For the expected successful query cost CO_N, we have

$$CO_N = \frac{1}{N} \sum_{k=0}^{N-1} CO'_k = 1 - \frac{1}{B} + \frac{\sum_{k=0}^{N-1} C'_k}{NB} = 1 - \frac{1}{B} + \frac{C_N}{B}.$$

□

Combing with Knuth's result that $C_N \approx \frac{1}{2}(1 + \frac{1}{1-\alpha})$ and $C'_N \approx \frac{1}{2}(1 + (\frac{1}{1-\alpha})^2)$, we conclude that the I/O cost of directly applying linear probing in the cache-oblivious model is $1 + \Theta(\alpha/B)$, which is a lot worse than its the external version that is aware of the blocking.

3.3 Blocked Probing

Standard linear probing maintains the invariant that each key x is placed as close as possible to position $h(x)$ in the probe sequence. *Blocked probing* is a variant of linear probing proposed by Pagh et al. [59], who used it to derive optimal performance (as a function of α) assuming only 5-wise independent hash functions. In this section, we demonstrate that blocked probing also achieves the desired $1 + 1/2^{-\Omega(B)}$ I/O bound in the cache-oblivious model, under the assumptions that the block size B is a power of 2 and the memory blocks are B-aligned.

3.3.1 Algorithm description

Let $[R] = \{0, 1, \ldots R - 1\}$ denote the hash table, where R is a power of two. It is also assumed that R is fixed, i.e., there is no notion of dynamically adjusting the capacity of the hash table; at the end of this section we sketch how to handle the general case. Suppose that the key x is stored in location i_x, we define the distance measure $d(x, i_x)$ to be the

position of the most significant bit in which $h(x)$ and i_x differ (the least significant bit is said to be at position 1), and $d(x, i_x) = 0$ in case $i_x = h(x)$. Let $I(x, j) = \{i \mid d(x, i) \leqslant j\}$. Note that $I(x, j)$ is the *aligned* block of size 2^j that contains $h(x)$. The invariant of blocked probing is that each key is stored as close as possible to $h(x)$ in the sense that $i_x \in I(x, j)$ if there is sufficient space, i.e., if the number of keys with hash values in $I(x, j)$ is at most $|I(x, j)| = 2^j$. Below we describe the operations of blocked probing.

When *inserting* a key x, the invariant is maintained by searching, for $j = 0, 1, 2, \ldots$, for a location $i \in I(x, j)$ where x could be placed. For each j, we first check if there is an empty location in $I(x, j)$ and put x there if there is one. Otherwise, we look for a location $i_{x'} \in I(x, j)$ that contains a key x' with $d(x', i_{x'}) > j$ (implying that $h(x') \notin I(x, j)$). If there is such an x', we swap x and x', and continue the insertion process with x'. If both attempts fail, we move on to the next j.

A *lookup* for x proceeds by inspecting, for $j = 0, 1, 2, \ldots$, the locations of $I(x, j)$ until either x is found, or we do not find x but find instead an empty location or a key x' with $d(x', i_{x'}) > j$. In the latter cases, the invariant tells us that x is not present in the hash table.

Deletion of a key $x \in I(x, j) \backslash I(x, j-1)$ needs to check if there is a key stored in $I(x, j+1) \backslash I(x, j)$ that could be stored in $I(x, j)$ — if this is the case it is copied to the empty location, and the old copy is deleted recursively.

3.3.2 Cache-oblivious analysis of blocked probing

We assume the block size B is a power of two, and the i-th block $\mathcal{B}_i$ starts at position iB and ends at position $iB + B - 1$. Then for any key x, the aligned block $I(x, \log B)$ is the block that contains $h(x)$. Let $\mathcal{D}$ denote the set of keys involved in a given operation (insertion, deletion, successful or unsuccessful search), including the key x specified by the query or update (x may or may not be in the hash table). To bound the expected I/O cost for an operation, define event $\mathcal{E}_s(x, j)$ as the aligned block $I(x, j)$ being *saturated*, that is, the number of keys in $\mathcal{D}$ with hash value in the aligned block $I(x, j)$ is 2^j or more. Let $p(x, j)$ denote the probability that $\mathcal{E}_s(x, j)$ happens. The following lemma relates $p(x, j)$ with C_{bp}, the expected I/O cost for an operation of blocked probing.

Lemma 3.2 *Suppose function h is uniformly drawn from a pairwise independent hash family $\mathcal{H}$, then*

$$C_{bp} \leqslant 1 + \sum_{j=1+\log B}^{\log R} \frac{2^{j+2}}{B} p(x, j).$$

Proof: We first note that the cost of a search for key x is bounded by that of an insertion of x, so we only need to consider insertions and deletions. Let $\mathcal{E}_f(x, j)$ denote the event

that the aligned block $I(x, j)$ is full, that is, the number of keys stored in $I(x, j)$ is 2^j. Let $q(x, j)$ denote the probability that $\mathcal{E}_f(x, j)$ happens. Observe that an insertion or a deletion would visit a location outside $I(x, j)$ only if all positions of $I(x, j)$ are occupied, so the probability that the operation takes at least $2^j/B$ I/Os is $q(x, j)$, for $j \geqslant \log B$. To compute the expected number of blocks involved in an operation, in addition to the first I/O that retrieves $I(x, \log B)$, we sum over all possible values of $j > \log B$ the cost $2^j/B$ multiplied by the probability that j steps or more are used:

$$C_{bp} \leqslant 1 + \sum_{j=1+\log B}^{\infty} \frac{2^j}{B} q(x, j). \tag{3.2}$$

Next we will relate $q(x, j)$, the probability that $I(x, j)$ is full, with $p(x, j)$, the probability that $I(x, j)$ is saturated. Divide the hash table R into $\log(R/2^B) + 1$ aligned blocks:

$$\mathcal{I} = \{I(x, j), I(x, j + 1)\backslash I(x, j), I(x, j + 2)\backslash I(x, j + 1), \ldots, I(x, R)\backslash I(x, R/2)\}.$$

The claim is that if $I(x, j)$ is full, then at least one of the aligned blocks in $\mathcal{I}$ is saturated. For a proof, assume that no aligned block in $\mathcal{I}$ is saturated. We inductively prove that each aligned block in $\mathcal{I}$ only stores keys with hash values inside it, which immediately implies that $I(x, j)$ is non-full, and thus leads to a contradiction. For the first insertion the statement is true. Now suppose the statement is true after the k-th insertion. When the $(k + 1)$-th insertion y_{k+1} comes, let $I(x, l + 1)\backslash I(x, l)$ denote the aligned block in $\mathcal{I}$ that contains $h(y_{k+1})$. By the inductive hypothesis, $I(x, l + 1)\backslash I(x, l)$ only contains the keys with hash values in it, and since $I(x, l + 1)\backslash I(x, l)$ is not saturated we know that $I(x, l + 1)\backslash I(x, l)$ is non-full. Therefore the key y_{k+1} is stored in an empty position of $I(x, l + 1)\backslash I(x, l)$, and the induction follows.

Observe that since the hash function h is drawn from a pairwise independent family, the probability that the $I(x, l+1)\backslash I(x, l)$ is saturated is the same as the probability that $I(x, l)$ is saturated, that is, $p(x, l)$. By a union bound we have the following inequality:

$$q(x, j) \leqslant p(x, j) + \sum_{l=j}^{\log R} p(x, l). \tag{3.3}$$

Combining (3.2) and (3.3) we have

$$\begin{aligned} C_{bp} &\leqslant 1 + \sum_{j=1+\log B}^{\log R} \frac{2^j}{B} q(x,j) \\ &\leqslant 1 + \sum_{j=1+\log B}^{\log R} \frac{2^j}{B} \left(p(x,j) + \sum_{l=j}^{\log R} p(x,l) \right) \\ &= 1 + \sum_{j=1+\log B}^{\log R} \frac{1}{B} \left(2^j + \sum_{l=1+\log R}^{j} 2^l \right) p(x,j) \\ &\leqslant 1 + \sum_{j=1+\log B}^{\log R} \frac{2^{j+2}}{B} p(x,j). \end{aligned}$$

□

For a truly random hash function, $p(x,j)$ can bounded using the Chernoff bound: The probability that a key is hashed to $I(x,j)$ is $2^j/R$, so the expected number of keys hashed to $I(x,j)$ is $2^j N/R = \alpha 2^j$. Recall that $p(x,j)$ is the probability that the number of keys hashed to $I(x,j)$ is 2^j or more, by the Chernoff bound, $p(x,j) \leqslant 2^{-(1-\alpha)^2\, 2^{j-1}}$. Following Lemma 3.2, we have

$$\begin{aligned} C_{bp} &\leqslant 1 + \sum_{j=1+\log B}^{\log R} 2^{j+2} p(x,j) \\ &= 1 + \sum_{j=1+\log B}^{\log R} (2^{j+2}/B) 2^{-(1-\alpha)^2\, 2^{j-1}} \\ &\leqslant 1 + 2^{-\Omega((1-\alpha)^2 B)}. \end{aligned}$$

That h is a truly random hash function is an unrealistic assumption. To analyze blocked probing with limited independence, we need the following variant of the Chernoff bound by Schmidt et al. [63]:

Lemma 3.3 ([63]) *Let $X_1, \ldots, X_N$ be a sequence of k-wise independent random variables, that satisfy $|X_i - \mathbf{E}[X_i]| \leqslant 1$. Let $X = \sum_{i=1}^N X_i$ with $\mathbf{E}[X] = \mu$, and let $\delta^2[X]$ denote the variance of X, so that $\delta^2[X] = \sum_{i=1}^N \delta^2[X_i]$ (this equation holds provided $k \geqslant 2$). Then for any even k and $C \geqslant \max\{k, \delta^2[X]\}$,*

$$\Pr[|X - \mu| \geqslant T] \leqslant \left(\frac{kC}{e^{2/3} T^2} \right)^{k/2}.$$

Lemma 3.2 and 3.3 together will lead to the following result:

Theorem 3.4 *Consider a blocked probing hash table in the cache-oblivious model where the block size B is power of 2 and every block starts at a memory address divisible by B.*

Suppose the hash table has a fixed size R and the hash function h is chosen uniformly at random from a k-wise independent hash family for odd $k \geqslant 5$. For any sequence of operations (insertions, deletions, and lookups), let α denote the load factor of the hash table during a particular operation. Then the expected number of I/Os for that operation is

$$C_{bp} = 1 + O\left(\left(\frac{k-1}{e^{2/3}(1-\alpha)^2 B}\right)^{(k-1)/2}\right).$$

Proof: Consider an operation on key x. We need to bound $p(x,j)$, the probability that the aligned block $I(x,j)$ is saturated, for $j \geqslant \log B$. Let X_i denote the random variable indicating that the i-th key has hash value in $I(x,j)$. Note that $X_1, \ldots, X_N$ are $(k-1)$-wise independent, and for each X_i we have $\mathbf{E}[X_i] = 2^j/R$ and $\delta^2[X_i] = 2^j/R(1-2^j/R) \leqslant 2^j/R$. It follows that $\mathbf{E}[X] = \sum_{i=1}^{N} \mathbf{E}[X_i] = 2^j N/R = \alpha 2^j$ and $\delta^2[X] = \sum_{i=1}^{N} \delta^2[X_i] \leqslant 2^j N/R = \alpha 2^j$. Setting $\mu = \alpha 2^j$, $T = (1-\alpha)2^j$, $C = 2^j \geqslant \max\{k, \delta^2[X]\}$ in Lemma 3.3, we derive a bound on $p(x,j)$:

$$p(x,j) = \Pr[X - \alpha 2^j \geqslant (1-\alpha)2^j] \leqslant \left(\frac{k-1}{e^{2/3}(1-\alpha)^2 2^j}\right)^{(k-1)/2}. \tag{3.4}$$

Plugging (3.4) into Lemma 3.2:

$$\begin{aligned} C_{bp} &\leqslant 1 + \sum_{j=1+\log B}^{\log R} (2^{j+2}/B) p(x,j) \\ &\leqslant 1 + \sum_{j=1+\log B}^{\log R} \frac{2^{j+2}}{B} \cdot \left(\frac{k-1}{e^{2/3}(1-\alpha)^2 2^j}\right)^{(k-1)/2} \\ &\leqslant 1 + O\left(\left(\frac{k-1}{e^{2/3}(1-\alpha)^2 B}\right)^{(k-1)/2}\right). \end{aligned}$$

The last inequality uses that fact that the terms in the sum are geometrically decreasing when $k \geqslant 5$, and hence the sum is dominated by the first term. □

Remark: Since a k-wise independent hash function is also k'-wise independent for all $k' \geqslant k$, the bound in Theorem 3.4 is actually $1 + O\left(\min_{5 \leqslant k' \leqslant k} \left(\frac{k'-1}{e^{2/3}(1-\alpha)^2 B}\right)^{(k'-1)/2}\right)$.

Theorem 3.4 immediately leads to the following corollaries.

5-wise independence The minimum independence allowed in Theorem 3.4 is 5. In this case

$$C_{bp} = 1 + O\left(\frac{1}{B^2}\right).$$

Note that the dependence on the block size B is asymptotically better than $1 + \Theta(1/B)$.

$\Omega(B)$-wise independence To achieve the same bound as that of the truly random hash function, it suffices to have $k \geqslant k' = (1-\alpha)^2 B + 1$. By Theorem 3.4, it follows that

$$\begin{aligned} C_{bp} &= 1 + O\left(\left(\frac{k'-1}{e^{2/3}(1-\alpha)^2 B}\right)^{(k'-1)/2}\right) \\ &= 1 + O\left(\left(e^{2/3}\right)^{-(1-\alpha)^2 B/2}\right) \\ &\leqslant 1 + 2^{-\Omega((1-\alpha)^2 B)}. \end{aligned}$$

3.3.3 Cache-oblivious dynamic hash tables

The standard doubling/halving strategy can be used to maintain the load factor α in the range $1/2 - \varepsilon/2 \leqslant \alpha \leqslant 1 - \varepsilon$ as we insert and delete keys in the hash table where $\varepsilon > 0$ is any small constant. In such a range the expected I/O cost per operation is $1 + 1/2^{\Omega(B)}$ I/Os using the blocked probing scheme described above. In particular, we always use a hash table of size R that is a power of 2. Let $g : [U] \to [U]$ be a "mother" hash function. When the table's size is R, we take the $\log R$ least significant bits of $g(x)$ as $h(x)$. When $\alpha = N/R$ goes beyond the range $[1/2 - \varepsilon/2, 1 - \varepsilon]$ we double or halve R accordingly. This can be done in a simple scan of the hash table in amortized $O(1/B)$ I/Os per key, by simply inserting keys in the order they occur in the table. The analysis uses the fact that the keys to be inserted in a block in the resized hash table are (w.h.p.) in at most two blocks in the original hash table. We omit the rather standard analysis.

However, the above solution has a poor space utilization. A number of methods have been proposed that maintain a higher load factor, and also allow the rehashing to be done incrementally; see [48] for an overview. To our best knowledge these methods are all cache-aware — however, we now describe how they can be made cache-oblivious while maintaining the load factor of $\alpha = 1 - \Theta(\varepsilon)$. Suppose initially R is a power of 2 and $N > (1 - 2\varepsilon)R$. Adjust ε so that εR is also a power of 2; this will not change ε by more than a factor of 2. The idea is to split the hash table into $1/\varepsilon$ parts using hashing (say, by looking at the first $\log(1/\varepsilon)$ bits of the mother hash function), where each part is handled by a cache-oblivious hash table of size εR which stores at most $(1 - \varepsilon)\varepsilon R$ keys. As N changes, the number of parts also changes to maintain the overall load factor at $\alpha = 1 - \Theta(\varepsilon)$. Now this situation is analogous to a standard cache-aware hash table with "block size" being equal to $(1 - \varepsilon)\varepsilon R$, and parts corresponding to blocks. So we may use any cache-aware method that resizes a standard hash table, such as *linear hashing* [51]. These resizing techniques will split or merge parts as needed, and cost is $O(1/B)$ I/Os per insertion/deletion amortized. When R doubles or halves, we rebuild the entire hash table using a new part size εR. The cache-aware resizing techniques ensures that only $1 + 1/2^{\Omega(B')}$ parts are accessed upon a query in expectation, where B' is the part size

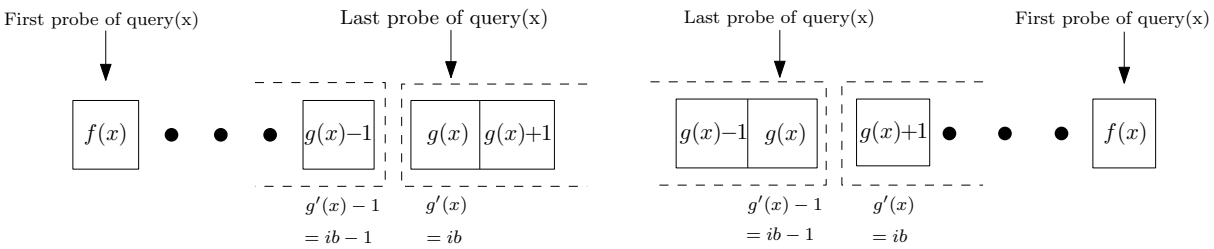

Figure 3.1: When two I/Os are needed.

$B' = (1-\varepsilon)\varepsilon R$. Within each part, our cache-oblivious scheme accesses $1+1/2^{\Omega(B)}$ blocks. So as long as $R \gg B$, the overall query cost is still $1 + 1/2^{\Omega(B)}$ I/Os, as desired.

In summary, we can dynamically update our cache-oblivious hash table while maintaining a high load factor. The additional resizing cost is only $O(1/B)$ I/Os amortized.

Theorem 3.5 *In the cache-oblivious model where the block size B is a power of 2 and every block starts at a memory address divisible by B, there is a dynamic hash table that supports queries in expected average $t_q = 1 + 1/2^{\Omega(B)}$ I/Os, and insertions and deletions of keys in expected amortized $1 + O(1/B)$ I/Os. The load factor can be maintained at $\alpha \geqslant 1 - \varepsilon$ for any constant $\varepsilon > 0$.*

Remark If a k-wise independent hash family is used, the bound on t_q in the above theorem will be replaced by the bound in Theorem 3.4.

3.4 Lower Bounds

In this section, we show that the two conditions that the analysis of blocked probing depends upon are both necessary to achieve a $1 + 1/2^{\Omega(B)}$ search cost. Specifically, we prove that when either condition is removed, the best obtainable bound for the expected average cost of a successful search is $1 + O(\alpha/B)$ I/Os. The lower bound proofs allow α to be asymptotically small, so it means that we cannot hope to do a lot better even with super-linear space.

3.4.1 The model

Before we present the exact lower bound statements let us first be more precise about our model. Let $[U]$ be the universe. The hard input we consider here is a random input in which each key is drawn from $[U]$ uniformly and independently. Let $I_{\mathbf{U}}$ be such a random input, and $\mathcal{I}$ be the set of all inputs. We will bound from below the expected average cost of a successful search on $I_{\mathbf{U}}$ where the average is taken over all keys in $I_{\mathbf{U}}$. We will only

consider deterministic hash tables; the lower bounds also hold for randomized hash tables by invoking *Yao's minimax principle* [57] because we are using a random input. The hash table can employ any hash functions to distribute the input. We assume $U > N^3$, then with probability $1 - O(1/N)$ all keys in $I_{\mathbf{U}}$ are distinct by the birthday paradox.

We assume that all the N keys are stored in a table of size R on external memory[2], possibly with duplication. We model the search algorithm by two functions $f, g : [U] \to [R]$. For any $x \in [U]$, $f(x)$ is the position where the algorithm makes its first probe, while $g(x)$ is the position of the last probe, where key x (or one of its copies) must be located. Note that the internal memory must be able to hold the description of f, thus any deterministic hash table can employ a family $\mathcal{H}$ of at most $2^{M \log U}$ such functions. Although the particular f used by the hash table of course can depend on the input $I_{\mathbf{U}}$, the family $\mathcal{H}$ has to be fixed in advance. We do not have any restrictions on g, as it is possible for the search algorithm to evaluate g after accessing external memory, except that all $g(x)$'s are distinct for the N keys.

The table is partitioned into blocks of size B. For any x such that $f(x) \neq g(x)$, define $g'(x)$ to be $g(x)$ if $f(x) < g(x)$ and $g(x) + 1$ if $f(x) > g(x)$. Then if $g'(x)$ is the first position of a block, at least two blocks must have been accessed, though the reverse is not necessarily true; please refer to Figure 3.1. For lower bound purposes we will assume optimistically that the search for x needs two I/Os if $g'(x)$ is the first position of a block, and one I/O otherwise. Note that after this abstraction, the search cost is completely characterized by the functions f, g and the blocking.

We will consider the following two blocking models. In the *boundary-oblivious* model, the hash table knows the block size B but not their boundaries. More precisely, how the keys are stored in the table is allowed to depend on B, but the layout should work for any shifting s, namely when each block spans the positions from $iB - s$ to $(i+1)B - s - 1$ for $s = 0, 1, \ldots, B - 1$. In the *block-size-oblivious* model, the blocks always start at positions that are multiples of B but the layout is required to work for all $B = 1, \ldots, R$. Below we will show that in either model, the best possible expected average cost of a successful search is $1 + O(\alpha/B)$ I/Os.

3.4.2 Good inputs and bad inputs

For any $I \in \mathcal{I}, f \in \mathcal{H}$, define $\eta_f(I) = \sum_{i \in [R]}(|\{x \in I \mid f(x) = i\}| - 1)$. Intuitively, $\eta_f(I)$ is the number of the overflowed keys; since each position i can only hold one key, at least $\eta_f(I)$ keys in I need a second probe when the hash table uses f to decide its first probe. We say an input $I \in \mathcal{I}$ is *bad* with respect to f if $\eta_f(I) \geqslant \frac{\alpha}{4}N$, otherwise it is *good*. Let

[2]Here we do not allow keys to be stored in internal memory: since the memory holds at most M keys, it does not affect the average search cost as long as N is sufficiently larger than M.

$\mathcal{I}_f$ be the set of all bad inputs with respect to f, and $\mathcal{I}_{\mathcal{H}} = \bigcap_{f\in\mathcal{H}} \mathcal{I}_f$ which is the set of inputs that are bad with respect to all $f \in \mathcal{H}$. In our lower bounds we will actually focus only on the bad inputs $\mathcal{I}_{\mathcal{H}}$. The following technical lemma ensures that almost all inputs are in $\mathcal{I}_{\mathcal{H}}$.

Lemma 3.6 *For $N > cM \log U/\alpha^2$ where c is some sufficiently large constant and $\alpha = \omega(N^{-1/2})$, $I_{\mathbf{U}}$ is a bad input with respect to all $f \in \mathcal{H}$ with probability $1-o(1)$ as $N \to \infty$.*

The general idea of the proof is the following: We first show that for a particular f and a random $I_{\mathbf{U}}$, the probability that $I_{\mathbf{U}}$ is good with respect to f is $e^{-\Omega(\alpha^2 N)}$. Thus by a union bound, $I_{\mathbf{U}}$ is good for at least one $f \in \mathcal{H}$ with probability at most $e^{-\Omega(\alpha^2 n)} \cdot 2^{M \log U}$. So as long as N is large enough, $I_{\mathbf{U}}$ will be bad with respect to all $f \in \mathcal{H}$ with high probability.

We need the following bin-ball game, which models the way how f works on a uniformly random input:

A bin-ball game In a $(N, R, \vec{\beta})$ *bin-ball game*, we throw N balls into R bins independently at random. The probability that a ball goes to the j-th bin is β_j, where $\vec{\beta} = (\beta_0, \ldots, \beta_{R-1})$ is a prefixed distribution. Let Z denote the number of empty bins after N balls are thrown in.

Lemma 3.7 *In an $(N, R, \vec{\beta})$ bin-ball game, $\Pr[Z \leqslant R - N + \frac{\alpha}{4}N] \leqslant e^{-\Omega(\alpha^2 N)}$, where $\alpha = N/R$.*

Proof: Note that if $\vec{\beta}$ is the uniform distribution, the problem is known as the *occupancy problem* and the lemma can be proved using properties of martingales [57]. The same proof actually also holds for a nonuniform $\vec{\beta}$, so we just sketch it here:

Let Z_0 be the expectation of Z before any ball is thrown in, and let the random variable Z_i be the expectation of Z after the i-th ball is thrown in (where the randomness is from the first i balls), for $i = 1, \ldots, N$. Note that $Z_0 = \mathbf{E}[Z]$ and $Z_N = Z$. It can be verified that the sequence $Z_0, Z_1, \ldots, Z_N$ is a martingale, and that $|Z_{i+1} - Z_i| \leqslant 1$ for all $0 \leqslant i < N$. Therefore by Azuma's inequality, we get

$$\Pr[Z \leqslant \mathbf{E}[Z] - \lambda N^{1/2}] \leqslant 2e^{-\lambda^2/2}.$$

Note that

$$\begin{aligned} \mathbf{E}[Z] &= \sum_{i=0}^{R-1} (1-\beta_i)^N \geqslant R\left(\frac{R - \sum_{i=0}^{R-1}\beta_i}{R}\right)^N = R\left(1 - \frac{1}{R}\right)^N \\ &\geqslant R - N + \frac{\alpha}{2}N - \frac{\alpha}{2} - \frac{(N-1)(N-2)}{6N}\alpha^2. \end{aligned}$$

Setting $\lambda = (\frac{\alpha}{4}N - \frac{\alpha}{2} - \frac{(N-1)(N-2)}{6N}\alpha^2)N^{-1/2} = \Omega(\alpha N^{1/2})$, we have $\mathbf{E}[Z] - \lambda N^{1/2} \geqslant R - N + \frac{\alpha}{4}N$, hence the lemma. □

Now we are ready to prove Lemma 3.6.

Proof:(of Lemma 3.6) Consider a particular $f : [U] \to [R]$ and a random input $I_{\mathbf{U}}$. The probability that a randomly chosen key x from $[U]$ has $f(x) = i$ is exactly $|f^{-1}(i)|/U$. This is exactly an $(N, R, \vec{\beta})$ bin-ball game where $\beta_i = |f^{-1}(i)|/U$. Let Z be the number of empty bins at the end of such a bin-ball game. Note that we have $\eta_f(I) = N - (R - Z)$, which, by Lemma 3.6, does not exceed $\frac{\alpha}{4}N$ with probability at most $e^{-\Omega(\alpha^2 N)}$. Since there are $2^{M \log U}$ different f's in $\mathcal{H}$, by a union bound, the probability that $I_{\mathbf{U}}$ is good for at least one $f \in \mathcal{H}$ is at most $e^{-\Omega(\alpha^2 N)} \cdot 2^{M \log U}$. Thus if $N > cm \log U/\alpha^2$ for some sufficiently large c, this probability is $e^{-\Omega(\alpha^2 N)} = o(1)$. □

3.4.3 Lower bound for the boundary-oblivious model

Now we prove the lower bound for the boundary-oblivious model, where the layout is required to work for any shifting s.

Theorem 3.8 *For any fixed block size B, consider any hash table that stores N uniformly random keys. There exists some shifting s for which the hash table has an expected average successful search cost at least $1 + \frac{\alpha}{5B}$, for N sufficiently large and $\alpha = \omega(N^{-1/2})$.*

Proof: Consider any input $I \in \mathcal{I}$. Suppose that the hash table uses $f_I \in \mathcal{H}$ and g_I on input I. Define $\gamma(s, I)$ to be the number of keys in I that need two I/Os to search when the shifting is s, i.e., those keys x with $f_I(x) \neq g_I(x)$ and $g'_I(x) = iB - s$ for some integer i. Note that the average search cost on I is $1+\gamma(s, I)/N$, and the expected average search cost on a random $I_{\mathbf{U}}$ is $1 + \mathbf{E}_{\mathbf{U}}[\gamma(s, I_{\mathbf{U}})]/N$, which we will show to be greater than $1 + \frac{\alpha}{5B}$.

Consider any $I \in \mathcal{I}_{\mathcal{H}}$. Since I is bad for all $f \in \mathcal{H}$, it is also bad for f_I. Thus there are at least $\frac{\alpha}{4}N$ keys x in I with $f_I(x) \neq g_I(x)$. For these keys, $g'_I(x)$ is defined and there is exactly one s such that $g'_I(x) = iB - s$ for some integer i. So we have $\sum_{s=0}^{B-1} \gamma(s, I) \geqslant \frac{\alpha}{4}N$. By Lemma 3.6, $I_{\mathbf{U}}$ belongs to $\mathcal{I}_{\mathcal{H}}$ with probability $1 - o(1)$, so

$$\sum_{s=0}^{B-1} \mathbf{E}_{\mathbf{U}}[\gamma(s, I_{\mathbf{U}})] = \mathbf{E}_{\mathbf{U}}\left[\sum_{s=0}^{B-1} \gamma(s, I_{\mathbf{U}})\right] \geqslant (1 - o(1))\frac{\alpha}{4}N \geqslant \frac{\alpha}{5}N.$$

By the pigeonhole principle, we must have one s such that $\mathbf{E}_{\mathbf{U}}[\gamma(s, I_{\mathbf{U}})] \geqslant \frac{\alpha N}{5B}$, and the lemma is proved. □

3.4.4 Lower bound for the block-size-oblivious model

Next we give the lower bound under the block-size-oblivious model, in which the block boundaries are always multiples of B, but the layout of the hash table is required to work

with any B. Since it is not possible to prove a lower bound of the form $1 + \Omega(\alpha/B)$ for all B (that would be a lower bound in the cache-aware model), instead we show that $1+o(\alpha/B)$ is not achievable, i.e., the following is false: "$\forall\epsilon\exists N_0\exists B_0\forall N > N_0\forall B > B_0$, the cost is at most $1 + \epsilon\alpha/B$." In particular, we show that this statement is false for $\epsilon = \frac{1}{17}$.

Theorem 3.9 *Consider any hash table that stores N uniformly random keys. For any B_0, there exists a block size $B \geqslant B_0$ on which the expected average success search cost on N keys is at least $1 + \frac{\alpha}{17B}$, for any N sufficiently large and $\alpha = \omega((\log\log N)^{-1/2})$.*

We follow the same framework as in the proof of Theorem 3.8. Let $\rho(B, I)$ be the number of keys x in I with $f_I(x) \neq g_I(x)$ and $B|g'_I(x)$; these keys need two I/Os to search when the block size is B in the block-size-oblivious model. On a random $I_{\mathbf{U}}$, the expected average search cost is $1 + \mathbf{E}_{\mathbf{U}}[\rho(B, I_{\mathbf{U}})]/N$. From here suppose we were to continue to follow the proof of Theorem 3.8 and consider the summation of $\mathbf{E}_{\mathbf{U}}[\rho(B, I_{\mathbf{U}})]$ over all $B \in \{B_0, B_0 + 1, \ldots, R\}$. Each x contributes 1 to the summation when $B = g'_I(x)$, so we still have $\sum_{B=B_0}^{R} \mathbf{E}_{\mathbf{U}}[\rho(B, I_{\mathbf{U}})] = \Omega(\alpha N)$. This, unfortunately, only guarantees the existence of a B such that $\mathbf{E}_{\mathbf{U}}[\rho(B, I_{\mathbf{U}})]$ is at least $\Omega(\frac{\alpha N}{R})$ or $\Omega(\frac{\alpha N}{B\log R})$, where the latter uses the fact that $\sum_{B=B_0}^{R} 1/B = \Theta(\log R)$. Neither is strong enough to give us the desired lower bound. Below we show how we prove Theorem 3.9 by restricting B to the primes and a much more careful analysis.

Lemma 3.10 *Let P_k be the set of all primes that are smaller than k, and let $P = P_R - P_{B_0}$ be the set of all primes that are in the range $[B_0, R)$. For $\alpha = \omega((\log\log N)^{-1/2})$, we have*

$$\mathbf{E}_{\mathbf{U}}\left[\sum_{B\in P} \rho(B, I_{\mathbf{U}})\right] = \sum_{B\in P} \mathbf{E}_{\mathbf{U}}[\rho(B, I_{\mathbf{U}})] > (1 - o(1))\frac{\alpha}{16} N \log\log R,$$

as $N \to \infty$.

Note that Lemma 3.10 implies that there must be a $B \in P$ such that $\mathbf{E}[\rho(B, I_{\mathbf{U}})] \geqslant \frac{\alpha}{17B}N$, proving Theorem 3.9, since otherwise we would have

$$\sum_{B\in P} \mathbf{E}[\rho(B, I_{\mathbf{U}})] \leqslant \frac{\alpha}{17} N \sum_{B\in P} \frac{1}{B} \leqslant \frac{\alpha}{17} N(\log\log R + O(1)).$$

Here we use the following approximation for the prime harmonic series [65]:

$$\sum_{B\in P_R} \frac{1}{B} = \log\log R + O(1).$$

Thus $\sum_{B\in P} \mathbf{E}[\rho(B, I_{\mathbf{U}})] \leqslant \frac{\alpha}{17} N\,(\log\log R + O(1))$, contradicting Lemma 3.10.

Proof of Lemma 3.10 In the rest of this subsection we prove Lemma 3.10. We need the following fact from number theory. Let $\mu(s)$ denote the number of distinct prime factors of s.

Lemma 3.11 ([65]) *Let $\xi(R) \to \infty$. Then*

$$\left|\left\{l \leqslant R : |\mu(l) - \log\log R| > \xi(R)\sqrt{\log\log R}\right\}\right| = O\left(\frac{R}{\xi^2(R)}\right).$$

Proof:(of Lemma 3.10) By Lemma 3.6 we know that $I_{\mathbf{U}}$ belongs to $\mathcal{I}_{\mathcal{H}}$ with probability $1 - o(1)$, so it suffices to prove that for any $I \in \mathcal{I}_{\mathcal{H}}$,

$$\sum_{B \in P} \rho(B, I) > (1 - o(1))\frac{\alpha}{16} N \log\log R.$$

Consider any $I \in \mathcal{I}_{\mathcal{H}}$. Let G be the set of distinct $g'_I(x)$'s for the keys $x \in I$. Let $\mu_P(s)$ be the number of distinct prime factors of s that are in P. By definition $\mu_{P_{B_0}}(s)$ is the number of distinct prime factors of s that are in P_{B_0}, and it follows that $\mu(s) = \mu_{P_{B_0}}(s) + \mu_P(s)$. Note that $\rho(B, I)$ is at least the number of multiples of B in G, so we have

$$\sum_{B \in P} \rho(B, I) \geqslant \sum_{l \in G} \mu_P(l) = \sum_{l \in G} \mu(l) - \sum_{l \in G} \mu_{P_{B_0}}(l). \tag{3.5}$$

Next we show that $\sum_{l \in G} \mu(l)$ is large. Firstly, observe that

$$|G| > \frac{\alpha}{8} N. \tag{3.6}$$

This is because I is bad for f_I, so at least $\frac{\alpha}{4}N$ keys in I have $f_I(x) \neq g_I(x)$ and thus their $g'_I(x)$'s are defined. The $g_I(x)$'s for these keys must be distinct, and each $g'_I(x)$ is either $g_I(x)$ or $g_I(x) + 1$, so there are at least $\frac{\alpha}{8}N$ distinct $g'_I(x)$'s for the keys in I.

Secondly, by choosing $\xi(R) = \frac{(\log\log R)^{1/4}}{\sqrt{\alpha}}$ in Lemma 3.11 we get:

$$\begin{aligned}&\left|\left\{l \leqslant R : \mu(l) \leqslant \left(1 - \frac{1}{\sqrt{\alpha}(\log\log R)^{1/4}}\right)\log\log R\right\}\right| \\ &= O\left(\frac{\alpha R}{\sqrt{\log\log R}}\right).\end{aligned}$$

Since we require $\alpha = \omega(\frac{1}{\sqrt{\log\log N}})$ which implies $\frac{\alpha R}{\sqrt{\log\log R}} = \frac{\alpha N}{\alpha\sqrt{\log\log R}} = o(\frac{\alpha}{8}N)$ and $\frac{1}{\sqrt{\alpha}(\log\log R)^{1/4}} = o(1)$, it holds that for at least $|G| - o(1)\frac{\alpha}{8}N$ distinct $l \in G$,

$$\mu(l) > (1 - o(1))\log\log R. \tag{3.7}$$

By inequalities (3.6) and (3.7), we have

$$\sum_{l \in G} \mu(l) > (1 - o(1))\frac{\alpha}{8} N \log\log R. \tag{3.8}$$

It remains to upper bound $\sum_{l\in G}\mu_{P_{B_0}}(l)$. Note that for any $B\in P_{B_0}$, the number of integers in $[R]$ that are divisible by B is at most R/B, so each B will be counted at most R/B times in $\sum_{l\in G}\mu_{P_{B_0}}(l)$. Hence,

$$\sum_{l\in G}\mu_{P_{B_0}}(l)\leqslant\sum_{B\in P_{B_0}}R/B=R\left(\log\log B_0+O(1)\right).$$

Therefore, as long as $\alpha\geqslant\sqrt{\frac{32\log\log B_0}{\log\log N}}>\sqrt{\frac{16\log\log B_0}{\log\log N/\alpha}}$, we have

$$\log\log B_0<\frac{\alpha^2}{16}\log\log\frac{N}{\alpha},$$

so

$$\begin{aligned}\sum_{l\in G}\mu_{P_{B_0}}(l) &< \frac{\alpha}{16}N\log\log R+O(R)\\ &= (1+o(1))\frac{\alpha}{16}N\log\log R.\end{aligned}\tag{3.9}$$

Finally, combining (3.5), (3.9), and (3.8) completes the proof. □

Chapter 4

Equivalence between Priority Queues and Sorting in External Memory

4.1 Introduction

In this chapter we show that priority queues are almost computationally equivalent to sorting in external memory. We design a priority queue that uses the sorting algorithm as a black box, such that the update cost of the priority queue is essentially the same as the per key I/O cost of the sorting algorithm. Our priority queue is a non-trivial generalization of Thorup's, which is fundamentally an internal structure. The main reasons why Thorup's structure does not work in the I/O model are that it cannot flush the buffers I/O-efficiently, and it does not specify any order for performing the flush and rebalance operations. Moreover, deletions are supported in a very different way in the I/O model; we have to do it in a lazy fashion in order to achieve I/O-efficiency.

Our results Our main result is stated in the following theorem:

Theorem 4.1 *Suppose we can sort up to N integer keys in $N \cdot S(N)/B$ I/Os in external memory, where S is a non-decreasing function. Then there exists an external priority queue that uses linear space and supports a sequence of N insertion, deletion and findmin operations in $O(\frac{1}{B}\sum_{i\geqslant 0} S(B\log^{(i)}\frac{N}{B}))$ amortized I/Os per operation. The reduction uses $O(B)$ internal memory.*

The first implication of Theorem 4.1 is that if the main memory has size $\Omega(B\log^{(c)}\frac{N}{B})$ for any constant c, then our priority queue supports all operations with $O(S(N)/B)$ amortized I/O cost. This is because $S(N) = 0$ when $N \leqslant M$. We claim that the reduction is tight as long as the function S grows not too slowly with N/B, even with $O(B)$ main memory. More precisely, we have the following corollary:

Corollary 4.2 *For $S(N) = \Omega(2^{\log^*\frac{N}{B}})$, the priority queue supports all operations with $O(S(N)/B)$ amortized I/O cost; for $S(N) = o(2^{\log^*\frac{N}{B}})$, the priority queue supports all operations with $O(S(N)\log^*\frac{N}{B}/B)$ amortized I/O cost.*

The first part can be obtained by plugging $S(N) = 2^{\log^*\frac{N}{B}}$ into Theorem 4.1 and showing that the $S(B\log^{(i)}\frac{N}{B})$'s decrease at least exponentially with i. For the second part, we

simply relax all the $S(B\log^{(i)}\frac{N}{B})$'s to $S(N)$. Note that $2^{\log^*\frac{N}{B}} = o(\log^{(i)}\frac{N}{B})$ for any constant i, so it is very unlikely that a sorting algorithm could achieve $S(N) = o(2^{\log^*\frac{N}{B}})$, meaning that our reduction is essentially tight.

Related results Sorting and priority queue have been well studied in the comparison-based I/O model, in which the keys can only be accessed via comparisons. Aggarwal and Vitter [4] showed that $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os are sufficient and necessary to sort N keys in the comparison-based I/O model. This bound is often referred to as the *sorting bound*. If the comparison constraint is replaced by the weaker indivisibility constraint, there is an $\Omega(\min\{\frac{N}{B}\log_{M/B}\frac{N}{B}, N\})$ lower bound, and is known as the *permuting bound*. The two bounds are the same when $\frac{N}{B}\log_{M/B}\frac{N}{B} < N$; it is conjectured that for this parameter range, $\Omega(\frac{N}{B}\log_{M/B}\frac{N}{B})$ is still the sorting lower bound even without the indivisibility constraint. For $\frac{N}{B}\log_{M/B}\frac{N}{B} > N$, the current situation in the I/O model is the same as that in the RAM model, that is, the best upper bound is just to use the best RAM algorithm (the current best RAM sorting algorithm is an $O(N\log\log N)$ deterministic algorithm by Han [39] and an $O(N\sqrt{\log\log N})$ randomized one by Han and Thorup [40]), and there is no non-trivial lower bound. In view of the indivisibility lower bound, a sorting lower bound (without any restrictions) has been considered to be more hopeful in the I/O model than in the RAM model, and it was posed as a major open problem in [4]. Thus, our result provides a way to approach a sorting lower bound via that of priority queues, while data structure lower bounds seem easier to obtain than (concrete) algorithm lower bounds, except in restricted computation models.

Since a priority queue can be used to sort N keys with N insertion and N deletemin operations, it follows that $\Omega(\frac{1}{B}\log_{M/B}\frac{N}{B})$ is also a lower bound for the amortized I/O cost per operation for any external priority queue, in the comparison-based I/O model. There are many implementations that achieve this lower bound, such as the buffer tree [9], M/B-ary heaps [26], and array heaps [20]. See the survey [72] for more details. These implementations seem hard to translate to a priority queue-to-sorting reduction , as they are all tree-based structures and a key must be moved $\Omega(\log_{M/B}\frac{N}{B})$ times to "bubble up" or "bubble down".

Arge et al. [11] developed an cache-oblivious priority queue that achieves the sorting bound with the tall cache assumption, that is, M is assumed to be of size at least B^2. We note that their structure can serve as a priority queue-to-sorting reduction in the I/O model, by replacing the cache-oblivious sort with a possibly better external sorting algorithm. The new priority queue supports all operations in $O(\frac{1}{B}\sum_{i\geqslant 0} S(N^{(2/3)^i})$ amortized I/O cost if the sorting algorithm sorts N keys in $N\cdot S(N)/B$ I/Os. However, this reduction is not tight for $S(N) = O(\log\log\frac{N}{B})$, and there seems to be no easy way to get

rid of the tall cache assumption, even if the algorithm has the knowledge of M and B.

4.2 Structure

In this section, we will construct a priority queue that achieves the desired amortized I/O cost in Theorem 4.1. The priority queue consists of multiple layers whose sizes vary from N to cB, where c is some constant to be determined later. The i'th layer from above has size $\Theta(B \log^{(i)} \frac{N}{B})$, and the priority queue has $O(\log^* N)$ layers. For the sake of simplicity we will refer to a layer by its size. Thus the layers from the largest to the smallest are layer N, layer $B \log \frac{N}{B}$, ..., layer cB. Layer cB is also called *head*, and is stored in main memory. Given a layer X, its *upper layer* and *lower layer* are layer $B2^{\frac{X}{B}}$ and layer layer $B \log \frac{X}{B}$, respectively. We use Ψ_X to denote $B2^{\frac{X}{B}}$ and Φ_X to denote $B \log \frac{X}{B}$. The priority queue maintains the invariant that the keys in layer Φ_X are smaller than the keys in layer X. In particular, the minimum key is always stored in the head and can be accessed without I/O cost.

We maintain a main memory buffer of size $O(B)$ to accomodate incoming insertion and deletion operations. In order to distribute keys in the memory buffer to different layers I/O-efficiently, we maintain a structure called *layer navigation list*. Since this structure will also be used in other components of the priority queue, we define it in a unified way. Suppose we want to distribute the keys in a buffer $\mathcal{B}$ to t sub-structures $S_1, S_2, \ldots S_t$. The keys in different sub-structures are sorted relative to each other, that is, the keys in S_i are less or equal to the keys in S_{i+1}. Each sub-structure S_i is associated with a buffer $\mathcal{B}_i$, which accommodates keys transferred from $\mathcal{B}$. The goal is to distribute the keys in $\mathcal{B}$ to each $\mathcal{B}_i$ I/O-efficiently, such that the keys that go to $\mathcal{B}_i$ have values between the minimum keys of S_i and S_{i+1}. A navigation list stores a set of t representatives, each representing a sub-structure. The representative of S_i, denoted r_i, is a triple that stores the minimum key of S_i, the number of keys stored in $\mathcal{B}_i$, and a pointer to the last non-full block of the buffer $\mathcal{B}_i$. The representatives are stored consecutively on the disk, and are sorted on the minimum keys. The layer navigation list is built for the $O(\log^* N)$ layers, so it has size $O(\log^* N)$. Please see Figure 4.1.

Now we will describe the structures inside a layer X except layer cB, which is always in the main memory. First we maintain a *layer buffer* of size $\Phi_X/2$ to store keys flushed from the memory buffer. The main structure of layer X consists of $O(\log \frac{X}{\Phi_X})$ levels with exponentially increasing sizes. The j'th level from the bottom, denoted level j, is of size $\Theta(8^j \Phi_X)$. We also keep the invariant that the keys in level j are less or equal to the keys in level $j+1$. We maintain a *level navigation list* of size $\Theta(\log \frac{X}{\Phi_X})$, which represents the $\log \frac{X}{\Phi_X}$ levels. Most keys in level j are stored in $\Theta(8^j)$ disjoint *base sets*, each of size

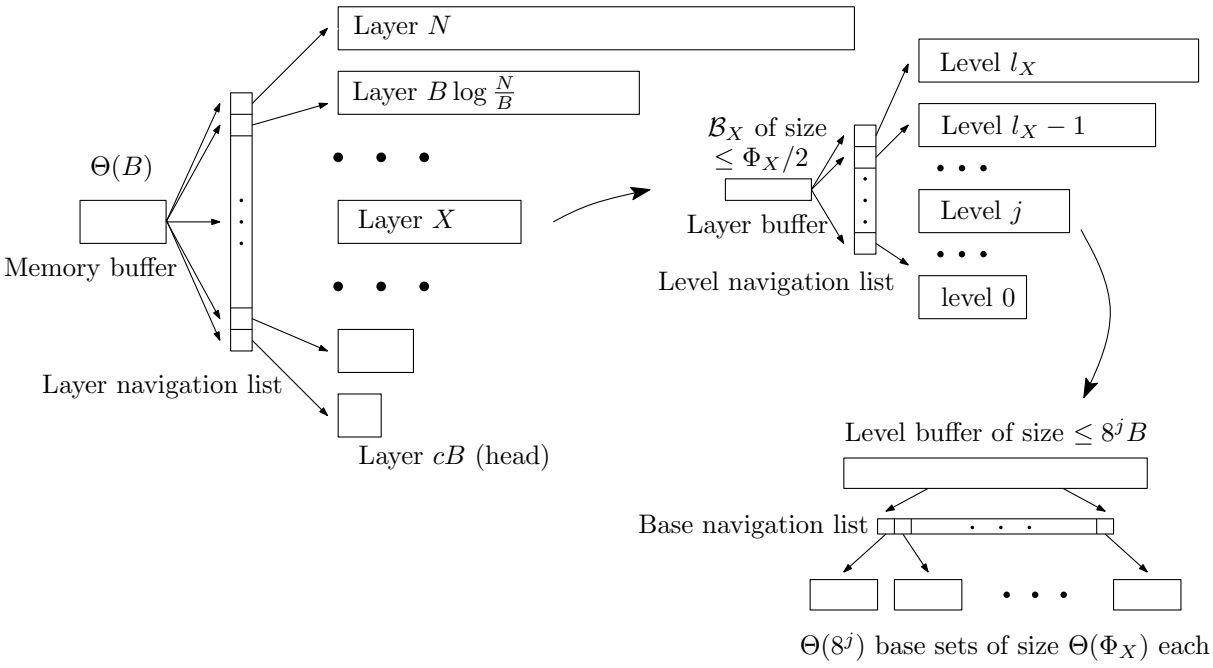

Figure 4.1: The components of the priority queue.

$\Theta(\Phi_X)$. The base sets, from left to right, are sorted relative to each other, but they are not internally sorted. Other than the base sets, there is a *level buffer* of size $8^j B$, which is used to temporarily accomodate keys before distributing them to the base set. We also maintain a *base navigation list* of size $\Theta(8^j)$ for the base sets. Note that we do not impose the level structures on layer cB since it can fit in the main memory. The components of the priority queue are illustrated in Figure 4.1.

Here we provide some intuition for this complicated structure. We first divide the keys into exponentially increasing levels, which in some sense is similar to building a heap. However, the $O(\log N)$-level structure implies that a key may be moved $O(\log N)$ times in its lifetime. To overcome this, we group the keys into base sets of logarithmic sizes so that we can move more keys with the same I/O cost (we will move pointer to the base sets rather than the base sets themselves). Finally, when a base set gets down to level 0, we need to recursively build the structure on it, which results in the the $O(\log^* N)$ layers.

Let l_X denote the top level of layer X. We use $\mathcal{B}_X$ to denote the layer buffer of layer X and $\mathcal{B}_j$ to denote the level buffer of level j when the layer is specified. Our priority queue maintains the following invariants for layer X:

Invariant 1 *The layer buffer $\mathcal{B}_X$ contains at most $\frac{1}{2}\Phi_X$ keys; the level buffer $\mathcal{B}_j$ at layer X contains at most $8^j B$ keys.*

Invariant 2 *The layer buffer $\mathcal{B}_X$ only contains keys between the minimum keys of layer*

X and its upper layer. The level buffer $\mathcal{B}_j$ only contains keys between the minimum keys of level j and its upper level.

Invariant 3 *A base set in layer X has size between $\frac{1}{2}\Phi_X$ and $2\Phi_X$; level j of layer X, for $j = 0, 1, \ldots, l_X - 1$, has size between $2 \cdot 8^j \Phi_X$ and $6 \cdot 8^j \Phi_X$, and level l_X has size between $2 \cdot 8^{l_X} \Phi_X$ and $40 \cdot 8^{l_X} \Phi_X$.*

Invariant 4 *The head contains at most $2cB$ keys.*

Note that when we talk about the size of a level, we only count the keys in its base sets and exclude the level buffer. The top level has a slightly different size range so that the construction works for any value of X.

We say a layer buffer, a level buffer, a base set, a level or the head overflows if its size exceeds its upper bound in Invariant 1, 3 or 4; we say a base set, a level underflows if its size gets below the lower bound in Invariant 3.

4.3 Operations

Recall that the priority queue supports three operations: insertion, deletion, and findmin. Since we always maintain the minimum key in the main memory (it is in either the head or the memory buffer), the cost of a findmin operation is free. We process deletions in a lazy fashion, that is, when a deletion comes we generate a delete signal with the corresponding key and a time stamp, and insert the delete signal to the priority queue. In most cases we treat the delete signals as norm insertions. We only perform the actually delete in the head so that the current minimum key is always valid. To ensure linear space usage we perform a global rebuild after every $N/8$ updates.

Our priority queue is implemented by three general operations: *global rebuild*, *flush*, and *rebalance*. A global rebuild operation sorts all keys and processes all delete signals to maintain linear size. A flush operation distributes all keys in a buffer to the buffers of corresponding sub-structures to maintain Invariant 1. A rebalance operation moves keys between two adjacent sub-structures to maintain Invariant 3.

Global Rebuild

We conduct the first global rebuild when the internal memory buffer is full. Then, after each global rebuild, we set N to be the number of keys in the priority queue, and keep it fixed until the next global rebuild. A global rebuild is triggered whenever layer N (in fact, its top level) becomes unbalanced or the priority queue has received $N/8$ new updates since the last global rebuild. We show that it takes $O(N \cdot S(N)/B)$ I/Os to

rebuild our priority queue. We first sort all keys in the priority queue and process the delete signals. Then we scan through the remaining keys and divide them into base sets of size Φ_N, except the last base set which may be smaller. This base set is merged to its predecessor if its size is less or equal to $\frac{1}{2}\Phi_N$. The first base set is used to construct the lower layers, and the rest are used to construct layer N. To rebuild the $O(\log \frac{N}{\Phi_N})$ levels of layer N, we scan through the base sets, and take the next $4 \cdot 8^j$ base sets to build level j, for $j = 0, 1, 2, \ldots$. Note that the base navigation list of these $4 \cdot 8^j$ base sets can be constructed when we scan through the keys in the base sets. The level rebuild process stops when we encounter an integer l_N such that the number of remaining base sets is more than $4 \cdot 8^{l_N}$, but less or equal to $4 \cdot (8^{l_N} + 8^{l_N+1}) = 36 \cdot 8^{l_N}$. Then we take these base sets to form the top level of layer N. After the global rebuild, level j has size $4^j\Phi_N$, and the top level l_N has size between $(4 \cdot 8^{l_N} - \frac{1}{2})\Phi_N$ and $(36 \cdot 8^{l_N} + \frac{1}{2})\Phi_N$. For $X = B \log \frac{N}{B}, B \log^{(2)} \frac{N}{B}, \ldots, cB$, layer X are constructed recursively using the same algorithm. All buffers are left empty.

Based on the global rebuild algorithm, the priority queue maintains the following invariant between two global rebuilds:

Invariant 5 *The top level l_X in layer X is determined by the maximum l_X such that*

$$1 + \sum_{j=0}^{l_X} 4 \cdot 8^j \leqslant \frac{X}{\Phi_X}.$$

The number of layers and the number of levels in each layer will not change between two global rebuilds.

As a result of Invariant 5, we have the following lemma:

Lemma 4.3 *Consider the top level l_X in layer X. We have*

$$4 \cdot 8^{l_X}\Phi_X \leqslant X \leqslant 40 \cdot 8^{l_X}\Phi_X.$$

Flush

We define the flush operation in a unified way. Suppose we have a buffer $\mathcal{B}$ and k sub-structures $S_1, S_2, \ldots, S_k$. Each S_i is associated with a buffer $\mathcal{B}_i$, and a navigation list L of size k is maintained for the k sub-structures. To flush the buffer $\mathcal{B}$ we first sort the keys in it. Then we scan through the navigation list, and for each representative r_i in L, we read the last non-full block of $\mathcal{B}_i$ to the memory, and fill it with keys in $\mathcal{B}$. When the block is full, we write it back to disk, and allocate a new block. We do so until we encounter a key that is larger than the key in r_{i+1}. Then we update r_i, and advance to r_{i+1}. The I/O cost for a flush is the cost of sorting a buffer of size $|\mathcal{B}|$ plus one I/O for each sub-structure, so we have the following lemma:

Lemma 4.4 *The I/O cost for flushing keys in buffer $\mathcal{B}$ to k sub-structures is bounded by* $O(\frac{|\mathcal{B}|S(|\mathcal{B}|)}{B} + k)$.

There are three individual flush operations. A *memory flush* distributes keys in the internal memory buffer to $O(\log^* N)$ layer buffers; a *layer flush* on layer X distributes keys in the layer buffer to $O(\log \frac{X}{\Phi})$ level buffers in the layer; and a *level flush* on level j at layer X distributes keys in the level buffer to $\Theta(8^j)$ base sets in the level.

Rebalance

Rebalancing the base sets Base rebalance is performed only after a level flush, since this is the only operation that causes a base set to be unbalanced. Consider a level flush in level j of layer X. Suppose the base set A overflows after the flush. To rebalance A we sort and scan through the keys in it, and split it into base sets of size Φ_X. If the last base set has less than $\frac{1}{2}\Phi_X$ keys we merge it into its predecessor. Note that any base set coming out of a split has between $\frac{3}{2}\Phi_X$ and $\frac{1}{2}\Phi_X$ keys, so it takes at least $\frac{1}{2}\Phi_X$ new updates to any of them before it initiates a new split. Note that after the split we should update the representatives in the base navigation list. This can be done without additional I/Os to the level flush operation: We store all new representatives in a temporary list and rebuild the navigation list after all overflowed base sets are rebalanced in level j. A base set never underflows so we do not have a join operation.

Rebalancing the levels We define two level rebalance operations: *level push* and *level pull*. Consider level j at layer X. When the number of keys in level j (except the top level) gets to more than $6 \cdot 8^j\Phi_X$, a level push operation is performed to move some of its base sets to the upper level. More precisely, we scan through the navigation list of level j to find the first representative r_k such that the number of keys before r_k is larger than $4 \cdot 8^j\Phi_X$. Then we split the navigation list of level j around r_k and attach the second half to the navigation list of level $i+1$. Note that by moving the representatives we also move their corresponding base sets to level $j+1$. By Invariant 3, the number of keys in a base set is at most $2\Phi_X$, so the new level j has size between $4 \cdot 8^j\Phi_X$ and $(4 \cdot 8^j + 2)\Phi_X$. Finally, to maintain Invariant 2 we sort level buffer $\mathcal{B}_j$ and move keys larger than the r_k to the level buffer $\mathcal{B}_{j+1}$.

Conversely, if the number of keys in level j gets below $2 \cdot 8^j\Phi_X$ (except the top level), a level pull operation is performed. We cut a proportion of the navigation list of level $j+1$ and attach it to the navigation list of level j, such that the number of keys in level i becomes between $4 \cdot 8^j\Phi_X$ and $(4 \cdot 8^j - 2)\Phi_X$. We also sort $\mathcal{B}_{j+1}$, the buffer of level $j+1$, and move the corresponding keys to level buffer $\mathcal{B}_j$.

Observe that after a level push/pull, the number of keys in level j is between $(4 \cdot 8^j - 2)\Phi_X$ and $(4 \cdot 8^j + 2)\Phi_X$, so it takes at least $\Omega(8^j \Phi_X)$ new updates before the level needs to be rebalanced again. The main reason that we adopt this level rebalance strategy is that it does not touch all keys in the level; the rebalance only takes place on the base navigation lists and the keys in the level buffers.

Rebalancing the layers When the top level l_X of layer X becomes unbalanced, we can no longer rebalance it only using navigation list. Recall that its upper level is level 0 in layer Ψ_X. For simplicity we will refer to the the two levels as level l_X and level 0, without specifying their layers. We also define two operations for rebalancing a layer: *layer push* and *layer pull*. A layer push is performed when the layer overflows, that is, the number of keys in level l_X gets more than $40 \cdot 8^{l_X} \Phi_X$. In this case we sort all keys in level j and level 0 together, then use the first $4 \cdot 8^{l_X} \Phi_X$ keys to rebuild level j and the rest to rebuild level 0. Recall that to rebuild a level we scan through the keys and divide them into base sets of size Φ_X, except the last one which has size between $\frac{1}{2}\Phi_X$ and $\frac{3}{2}\Phi_X$, and then we scan through the keys again to build the base navigation list. Note that the rebuild operation will change the minimum key in layer Ψ_X, so we update the layer navigation list accordingly. Finally we sort the keys in the layer buffer $\mathcal{B}_X$ and the level buffer $\mathcal{B}_{l_X}$, and move the keys larger than the new minimum key of layer Ψ_X to the level buffer $\mathcal{B}_0$.

A layer pull operation is performed when the layer underflows, that is, there are less than $2 \cdot 8^{l_X} \Phi_X$ keys in level l_X. A layer pull proceeds in the same way as a layer push does, except for the last step. Here we sort the layer buffer $\mathcal{B}_{\Psi_X}$ and the level buffer $\mathcal{B}_0$ and move the keys smaller than the new minimum key to the level buffer $\mathcal{B}_{l_X}$. After a layer push or pull, the number of keys in level l_X is $4 \cdot 8^j \Phi_X$. By lemma 4.3, we have $40 \cdot 8^{l_X} \Phi_X \geqslant X$, so it takes at least $2 \cdot 8^{l_X} \Phi_X = \Omega(X)$ new updates to layer X before we initiate a new push or a pull again.

Note that since we do not impose the level structure on the head layer cB, we need to design the layer push and layer pull operations specifically for it. A layer push is performed when the number of keys in the head gets to more than $2cB$. We sort all keys in it and level 0 of layer Ψ_{cB}, and use the first cB keys to rebuild the head and the rest to rebuild level 0. A layer pull is performed when the head becomes empty. The operation processes in the same way as a layer push does, except that after rebuilding both levels, we sort the layer buffer $\mathcal{B}_{\Psi_{cB}}$ and the level buffer $\mathcal{B}_0$ together, and move the keys smaller than the new minimum key of layer $\mathcal{B}_{\Psi_{cB}}$ to the head.

Scheduling Flush and Rebalance Operations

In order to achieve the I/O bounds in Theorem 4.1, we need to schedule the operations delicately. Whenever the memory buffer overflows we start to update the priority queue. This process is divided into three stages: the flush stage, the push stage, and the pull stage. In the flush stage we flush all overflowed buffers and rebalance all unbalanced base sets; in the push stage we use push operations to rebalance all overflowed layers and levels. We treat delete signals as insertions in the flush stage and the push stage. In the pull stage we deal with delete signals and use pull operations to rebalance all underflowed layers and levels.

In the flush stage, we initialize a queue Q_o to keep track of all overflowed buffers and a doubly linked list L_o to keep track of all overflowed levels. The buffers are flushed in a BFS fashion. First we flush the memory buffer into $O(\log^* N)$ layer buffers. After flushing the memory buffer, we insert the representatives of the overflowed layer buffers into Q_o, from bottom to top. We also check whether the head overflows after the memory flush. If so, we insert its representatives to the beginning of L_o. Then we start to flush the layer buffers in Q_o. Again, when flushing a layer buffer we insert the representatives of the overflowed level buffers to Q_o from bottom to top. After all layer buffers are flushed, we begin to flush level buffers in Q_o. After each level flush, we rebalance all unbalanced base sets in this level, and if the level overflows we add the representative of this level to the end of L_o. Note that the representatives in L_o are sorted on the minimum keys of the levels.

After all overflowed level buffers are flushed, we enter the push stage and start to rebalance levels in L_o in a bottom-up fashion. In each step, we take out the first level in L_o (which is also the current lowest overflowed level) and rebalance it. Suppose this level is level j of layer X. If it is not the top level or the head layer we perform a level push; otherwise we perform a layer push. Then we delete the representative of this level from L_o. A level push may cause the level buffer of level $j+1$ to be overflowed, in which case we flush it and rebalance the overflowed base sets. Then we check whether level $j+1$ overflows. If so, we insert the representative of level $j+1$ to the head of L_o (unless it is already at the beginning of L_o) and perform a level push on level $j+1$. Otherwise we take out a new level in L_o and continue the process. When the top level of layer N become unbalanced we simply perform a global rebuild.

After rebalancing all levels, we enter the pull stage and start to process the delete signals. This is done as follows. We first process all delete signals in the head. If the head becomes empty we perform a layer pull to get more keys into the head. This may cause higher levels or layers to underflow, and we keep performing level pulls and layer pulls until all levels and layers are balanced. Consider a level pull or layer pull on level j of

layer X. After the level pull or layer pull the level buffer $\mathcal{B}_j$ may overflow. If so, we flush it and rebalance the base sets when necessary. Note that this may cause the size of level j to grow, but it will not overflow, as we will show later, so that we do not need push operations in the pull stage. After all levels and layers are balanced, we process the delete signals in the head again. We repeat the pull process until there are no delete signals left in the head and the head is non-empty.

Correctness

It should be obvious that the flush and the push stage will always succeed. The following two lemmas guarantee that the pull stage will also succeed.

Lemma 4.5 *When we perform a level pull on level j, there are enough keys in level $j+1$ to rebalance level j; When we perform a layer pull on layer X, there are enough keys in level 0 of layer Ψ_X to rebalance level l_X.*

Proof: Recall that a level pull on level j transfers at most $4 \cdot 8^j \Phi_X$ keys from level $j+1$ to level j. Since we always perform pull operations in a bottom-up fashion in the pull stage, and all levels and layers are balanced before the pull stage, it follows that level $j+1$ is always balanced when performing a pull operation on level j. This implies that level $j+1$ has at least $2 \cdot 8^{j+1} \Phi_X$ keys when performing a pull operation on level j, which is sufficient to supply the level pull operation.

For a layer pull on layer X other than the head, recall that the operation transfers at most $4 \cdot 8^{l_X} \Phi_X$ keys from level 0 of layer Ψ_X to level l_X. By similar argument we know level 0 is balanced, so it has at least $2X$ keys. Following Lemma 4.3, we have $2X \geqslant 8 \cdot 8^{l_X} \Phi_X$, so it suffices to supply the layer pull operation. The same argument also works for a layer pull on the head, since it acquires at most cB keys from the upper level, and the level contains at least $2cB$ keys. □

Lemma 4.6 *A level or a layer never overflows in the pull stage.*

Proof: Consider a level pull on level j of layer X. Recall that since we move some keys from $\mathcal{B}_{j+1}$ to $\mathcal{B}_j$, it is possible that $\mathcal{B}_j$ overflows and we need to perform a level flush on level j. We claim that after this level flush, level j is still balanced. For a proof, observe that level j has size between $4 \cdot 8^j \Phi_X$ and $(4 \cdot 8^j - 2)\Phi_X$ after the level pull, so it takes at least $2 \cdot 8^j B \log \frac{X}{B} \geqslant 2 \cdot 8^j cB$ new updates before level j overflows. Since the level pull transfers at most $8^{j+1} B$ keys from $\mathcal{B}_{j+1}$, after the level pull, $\mathcal{B}_j$ has less or equal to $8^j B + 8^{j+1} B = 9 \cdot 8^j B$ keys. Setting $c \geqslant 5$ allows level j to be still balanced after the level flush. This proves that a level never overflows in the pull stage.

Now consider a layer pull on layer X other than the head. Recall that we move some keys from the layer buffer $\mathcal{B}_{\Psi_X}$ and level buffer $\mathcal{B}_0$ to $\mathcal{B}_{l_X}$, it is possible that $\mathcal{B}_{l_X}$ overflows and we need to perform a level flush on the new level l_X. We claim that after this level flush, level l_X is still balanced. For a proof, observe that after the layer pull, it takes at least $36 \cdot 8^{l_X} B \log \frac{X}{B}$ new updates before level l_X overflows. Since the layer pull transfers at most $X/2$ keys from the layer buffer $\mathcal{B}_{\Psi_X}$ and at most $8B$ keys from the level buffer $\mathcal{B}_0$ to the level buffer of level l_X, the level flush operation flushes at most $X/2 + 8B + 8^{l_X} B$ keys to level l_X. By Lemma 4.3 we have

$$\begin{aligned} 36 \cdot 8^{l_X} \Phi_X &= 20 \cdot 8^{l_X} \Phi_X + 16 \cdot 8^{l_X} \Phi_X \\ &\geqslant X/2 + 16 \cdot 8^{l_X} B \log \frac{X}{B} \\ &\geqslant X/2 + 8B + 8^{l_X} B. \end{aligned}$$

So level l_X is still balanced after the level flush. Finally, consider a layer pull on the head. Recall that it takes at least cB new update to the head before it overflows. Since the head acquires at most $cB/2$ keys from the layer buffer $\mathcal{B}_{\Psi_{cB}}$, and at most $8B$ keys from the level buffer $\mathcal{B}_0$, we can set $c > 16$ such that $cB > cB/2 + 8B$, so the head will remain balanced after the layer pull. This proves that a layer never overflows in the pull stage. □

4.4 Analysis of Amortized I/O Complexity

We analyze the amortized I/O cost for each operation during $N/8$ updates. We will show that the amortized I/O cost per update is bounded by $O(\frac{1}{B} \sum_{i=0} S(B \log^{(i)} \frac{N}{B}))$, and Theorem 4.1 will follow.

Global rebuild Recall that the I/O cost for a global rebuild is $O(N \cdot S(N)/B)$ I/Os. We claim that during $N/8$ updates only a constant number of global rebuilds are needed, so the amortized I/O cost per update is bounded by $O(S(N)/B)$. This can be verified by the fact that a global rebuild can only be triggered by $N/8$ new updates or that the level l_N becomes unbalanced, and after a global rebuild it takes $\Omega(N)$ updates before level l_N becomes unbalanced again.

Flush We first analyze the I/O cost of three different flush operations. For memory flush, we sort a set of B keys in the memory and merge them with a navigation list of size $O(\log^* N)$. By Lemma 4.4, the I/O cost is $O(\log^* N)$. Therefore we charge $O(\frac{\log^* N}{B})$ I/Os for each of the updates in the memory buffer. Now consider a layer flush at layer X. Let $|\mathcal{B}_X|$ denote the number of updates in the layer buffer. By Invariant 1 the layer

flush operation is performed only if $|\mathcal{B}_X| \geqslant B \log \frac{X}{B}$. There are $O(\log \frac{X}{\Phi_X})$ level buffers, so by Lemma 4.4, the I/O cost is

$$\begin{aligned} O\left(\frac{|\mathcal{B}_X|\, S(|\mathcal{B}_X|)}{B} + \log \frac{X}{\Phi_X}\right) &= O\left(\frac{|\mathcal{B}_X|\, S(|\mathcal{B}_X|)}{B} + \frac{|\mathcal{B}_X|}{B}\right) \\ &= O\left(\frac{|\mathcal{B}_X|\, S(|\mathcal{B}_X|)}{B}\right) \\ &= O\left(\frac{|\mathcal{B}_X|\, S(N)}{B}\right). \end{aligned}$$

Thus, we can charge $O(S(N)/B)$ I/Os for each of the $\mathcal{B}$ updates in the layer buffer.

Next, consider a level flush at level j in layer X. Let $|\mathcal{B}_j|$ denote the number of updates in the buffer when we perform the flush operation, and by Invariant 1 we have $|\mathcal{B}_j| \geqslant 8^j B$. Recall that the size of the navigation list is $\Theta(8^j)$, so by Lemma 4.4, the I/O cost is

$$\begin{aligned} O\left(\frac{|\mathcal{B}_j| S(|\mathcal{B}_j|)}{B} + 8^j\right) &= O\left(\frac{|\mathcal{B}_j| S(|\mathcal{B}_j|)}{B} + \frac{|\mathcal{B}_j|}{B}\right) \\ &= O\left(\frac{|\mathcal{B}_j| S(|\mathcal{B}_j|)}{B}\right) \\ &= O\left(\frac{|\mathcal{B}_j| S(N)}{B}\right). \end{aligned}$$

Therefore we can charge $O(S(N)/B)$ I/Os for each of the $|\mathcal{B}_j|$ updates in the level buffer.

Rebalancing the base sets Consider a rebalance operation for a base set A at layer X. When A overflows we sort and divide it into equal segments. So the I/O cost for a base set rebalance can be bounded by the sorting time of $O(|A| + \Phi_X)$ updates. Note that there are at least $\Omega(|A|)$ updates to A since the last rebalance operation on it, and by Invariant 3 we have $|A| \geqslant 2\Phi_X$. Thus, the amortized I/O cost per update is $O(S(N)/B)$.

Rebalancing the levels We first consider a level push operation on level j of layer X. First, the operation cuts the base navigation list of level j, takes the first half to form a new level j, and attaches the rest to level $j+1$. The I/O cost for this cut-attach procedure is $\Theta(8^j/B + 1)$, since the navigation list is sorted and stored consecutively on disk. Then the operation sorts and redistributes the level buffer $\mathcal{B}_j$. Recall that we always flush the level buffer before rebalancing the level, so we have $|\mathcal{B}| \leqslant 8^j B$ when the level push is performed. The I/O cost for sorting and redistributing $\mathcal{B}_j$ is bounded by $O(8^j B S(8^j B)/B) = O(8^j S(X))$. Note that after a level push, it takes at least $\Theta(8^j \Phi_X)$ new updates to level j before it overflows again. So during $N/8$ updates at most $O(N/(8^j \Phi_X))$ level push operations are performed on level j. It follows that the I/O cost of all level push operations on level j is bounded by

$$O\left(8^j S(X) \cdot \frac{N}{8^j \Phi_X}\right) = O\left(\frac{N \cdot S(X)}{B \log \frac{X}{B}}\right).$$

We charge $O(S(X)/B \log \frac{X}{B})$ for each update and for each level in layer X. Since there are $O(\log \frac{X}{B})$ levels in layer X, we charge $O(S(X)/B)$ I/Os for each update in layer X. Summing up all layers, the amortized I/O cost for each update is $O(\frac{1}{B} \sum_{i=0} S(B \log^{(i)} \frac{N}{B}))$. A similar argument shows that the amortized I/O cost for the level pulls is the same, except that the I/O cost is amortized only on the delete signals.

Rebalancing the layers Consider a layer push operation on layer X. It takes the keys in level l_X of layer X and level 0 of layer Ψ_X, sorts them, and rebuilds both levels. Since both levels have size $O(X)$, the I/O cost is $O(XS(X)/B)$. We also note that after a layer push operation, it takes at least $\Theta(X)$ updates to level l_X before it goes unbalanced again. That means at most $O(N/X)$ layer rebalance operation is needed. So the I/O cost for the layer rebalances of layer X during the $N/8$ updates is $O(N \cdot S(X)/B)$. We can charge $O(S(X)/B)$ I/Os for each update and each layer, and summing up all layers, it is amortized $\frac{1}{B} \sum_{i=0} S(B \log^{(i)} \frac{N}{B})$ I/Os for each update. Similar argument shows that the amortized I/O cost for layer pulls is the same, except that the total I/O cost is amortized only on the delete signals.

Scheduling the operations Note that in the schedule we need to pay some extra I/Os for maintaining the queue Q_o and doubly linked list L_o. We observe that an update to Q_o or L_o would trigger a flush or rebalance operation later, and the cost of a flush or rebalance operation is at least 1 I/O. So Q_o and L_o can be maintained without increasing the asymptotic I/O cost.

Chapter 5

Data Structure for Summary Queries

5.1 Introduction

In this chapter we build internal and external memory data structures for two dimensional *summary queries*. Recall that the summary query problem is formally defined as follow: Let $\mathcal{D}$ be a data set containing N records. Each record $p \in \mathcal{D}$ is associated with a *query attribute* $A_q(p)$ and a *summary attribute* $A_s(p)$, drawing values possibly from different domains. A *summary query* specifies a range constraint $[q_1, q_2]$ on A_q and the database returns a summary on the A_s attribute of all records whose A_q attribute is within the range. Note that A_s and A_q could be the same attribute, but it is more useful when they are different, as the analyst is exploring the relationship between two attributes. Our goal is to build an data structure on $\mathcal{D}$ so that a summary query can be answered efficiently. As with any data structure problem, the primary measures are the query time and the space the structure uses. The data structure should also work in external memory, where it is stored in blocks of size B, and the query cost is measured in terms of the number of blocks accessed (I/Os). Finally, we also would like the data structure to support updates, i.e., insertion and deletion of records.

5.1.1 Related work on data structure for aggregation queries

In one dimension, most aggregates can be supported easily using a binary tree (a B-tree in external memory). At each internal node of the binary tree, we simply store the aggregate of all the records below the node. This way an aggregation query can be answered in $O(\log N)$ time ($O(\log_B N)$ I/Os in external memory).

In higher dimensions, the problem becomes more difficult and has been extensively studied in both the computational geometry and the database communities. Solutions are typically based on space-partitioning hierarchies, like partition trees, quadtrees and R-trees, where an internal node stores the aggregate for its subtree. There is a large body of work on spatial data structures; please refer to the survey by Agarwal and Erickson [3] and the book by Samet [62]. When the data space forms an array, the data cube [36] is an efficient structure for answering aggregation queries.

However, all the past research, whether in computational geometry or databases, has only considered queries that return simple aggregates like count, sum, max (min), and

very recently top-k [1] and median [18, 46]. The problem of returning complex summaries has not been addressed.

5.1.2 Related work on summaries

There is also a vast literature on various summaries in both the database and algorithms communities, motivated by the fact that simple aggregates cannot well capture the data distribution. These summaries, depending on the context and community, are also called *synopses*, *sketches*, or *compressed representations*. However, all past research has focused on how to construct a summary, either offline or in a streaming fashion, on the *entire* data set. No one has considered the data structure problem where the focus is to intelligently compute and store auxiliary information in the data structure at pre-computation time, so that a summary on a *requested subset* of the records in the database can be built quickly at query time. Since we cannot afford to look at all the requested records to build the summary at query time, this poses new challenges that past research cannot address: All existing construction algorithms need to at least read the data records once. The problem of how to maintain a summary as the underlying data changes, namely under insertions and deletions of records, has also been extensively studied. But this should not be confused with our dynamic data structure problem. The former maintains a single summary for the entire dynamic data set, while the latter aims at maintaining a dynamic structure from which a summary for any queried subset can be extracted, which is more general than the former. Of course for the former there often exist small-space solutions, while for the data structure problem, we cannot hope for sublinear space, as a query range may be small enough so that the summary degenerates to the raw query results.

Below we review some of the most fundamental and most studied summaries in the literature. Let D be a bag of items, and let $f_D(x)$ be the frequency of x in D.

Heavy hitters. A heavy hitters summary allows one to extract all frequent items approximately, i.e., for a user-specified $0 < \phi < 1$, it returns all items x with $f_D(x) > \phi|D|$ and no items with $f_D(x) < (\phi - \varepsilon)|D|$, while an item x with $(\phi - \varepsilon)|D| \leqslant f_D(x) \leqslant \phi|D|$ may or may not be returned. A heavy hitters summary of size $O(1/\varepsilon)$ can be constructed in one pass over D, using the MG algorithm [56] or the SpaceSaving algorithm [55].

Sketches. Various sketches have been developed as a useful tool for summarizing massive data. In this proposal, we consider the two most widely used ones: the *Count-Min sketch* [22] and the *AMS sketch* [6]. They summarize important information about D and can be used for a variety of purposes. Most notably, they can be used to estimate the join size of two data sets, with self-join size being a special case. Given the Count-Min

sketches (resp. AMS sketches) of two data sets D_1 and D_2, we can estimate $|D_1 \bowtie D_2|$ within an additive error of $\varepsilon F_1(D_1)F_1(D_2)$ (resp. $\varepsilon\sqrt{F_2(D_1)F_2(D_2)}$) with probability at least $1-\delta$ [5, 22], where F_k is the k-th frequency moment of D: $F_k(D) = \sum_x f_D^k(x)$. Note that $\sqrt{F_2(D)} \leqslant F_1(D)$, so the error of the AMS sketch is no larger. However, its size is $O((1/\varepsilon^2)\log(1/\delta))$, which is larger than the Count-Min sketch's size $O((1/\varepsilon)\log(1/\delta))$, so they are not strictly comparable. Which one is better will depend on the skewness of the data sets. In particular, since $F_1(D) = |D|$, the error of the Count-Min sketch does not depend on the skewness of the data, but $F_2(D)$ could range from $|D|$ for uniform data to $|D|^2$ for highly skewed data.

Quantiles. The quantiles (a.k.a. the *order statistics*), which generalize the median, are important statistics about the data distribution. Recall that the *ϕ-quantile*, for $0 < \phi < 1$, of a set D of items from a totally ordered universe is the one ranked at $\phi|D|$ in D (for convenience, for the quantile problem it is usually assumed that there are no duplicates in D). A *quantile summary* contains enough information so that for any $0 < \phi < 1$, an ε-approximate ϕ-quantile can be extracted, i.e., the summary returns a ϕ'-quantile where $\phi - \varepsilon \leqslant \phi' \leqslant \phi + \varepsilon$. A quantile summary has size $O(1/\varepsilon)$, and can be easily computed by sorting D, and then taking the items ranked at $\varepsilon|D|, 2\varepsilon|D|, 3\varepsilon|D|, \ldots, |D|$.

Wavelets. *Wavelet representations* (or just *wavelets* for short) take a different approach to approximating the data distribution by borrowing ideas from signal processing. Suppose the records in D are drawn from an ordered universe $[U] = \{1, \ldots, U\}$ and let $\mathbf{f}_D = (f_D(1), \ldots, f_D(U))$ be the frequency vector of D. Briefly speaking, in wavelet transformation we take U inner products $s_i = \langle \mathbf{f}_D, \mathbf{w}_i \rangle$ where $\mathbf{w}_i, i = 1, \ldots, U$ are the *wavelet basis vectors* (please refer to [34, 53] for details on wavelet basis vectors). The s_i's are called the *wavelet coefficients* of $\mathbf{f}_D$. If we kept all U wavelet coefficients, we would be able to reconstruct $\mathbf{f}_D$ exactly, but this would not be a "summary". The observation is that, for most real-world distributions, $\mathbf{f}_D$ yields few wavelet coefficients with large absolute values. Thus for a parameter k, even if we keep the k coefficients with the largest absolute values, and assume all the other coefficients are zero, we can still reconstruct $\mathbf{f}_D$ reasonably well. In fact, it is well known that among all the choices, retaining the k largest (in absolute value) coefficients minimizes the ℓ_2 error between the original $\mathbf{f}_D$ and the reconstructed one. Matias et al. [53] were the first to apply wavelet transformation to approximating data distributions. After that, wavelets have been extensively studied [31, 33, 34, 38, 54, 71], and have been shown to be highly effective at summarizing many real-world data distributions.

All the aforementioned work studies how to construct or maintain the summary on the

given D. In our case, D is the A_s attributes of all records whose A_q attributes are within the query range. Our goal is to design an data structure so that the desired summary on D can be constructed efficiently without actually going through the elements of D.

5.1.3 Other related work

A few other lines of work also head to the general direction of addressing the gap between reporting all query results and returning some simple aggregates. Lin et al. [50] and Tao et al. [64] propose returning only a subset of the query results, called "representatives". But the "representatives" do not summarize the data as we do. They also only consider skyline queries. The line of work on *online aggregation* [42, 45] aims at producing a random sample of the query results at early stages of long-running queries, in particular, joins. A random sample indeed gives a rough approximation of the data distribution, but it is much less accurate than the summaries we consider: For heavy hitters and quantiles, a random sample of size $\Theta(1/\varepsilon^2)$ is needed [67] to achieve the same accuracy as the $O(1/\varepsilon)$-sized summaries we mentioned earlier; for estimating join sizes, a random sample of size $\Omega(\sqrt{N})$ is required to achieve a constant approximation, which is much worse than using the sketches [5]. Furthermore, the key difference is that they focus on query processing techniques for joins rather than data structure issues. Correlated aggregates [32] aim at exploring the relationship between two attributes. They are computed on one attribute subject to a certain condition on the other. However, this condition has to be specified in advance and the goal is to compute the aggregate in the streaming setting, thus the problem is fundamentally different from ours.

5.1.4 Our results

To take a unified approach we classify all the summaries mentioned in Section 5.1.2 into F_1-based ones and F_2-based ones. The former includes heavy hitters, the Count-Min sketch, and quantiles, all of which provide an error guarantee of the form $\varepsilon F_1(D)$ (note that an ε-approximate quantile is a value with a rank that is off by $\varepsilon F_1(D)$ from the correct rank). The latter includes the AMS sketch and wavelets, both of which provide an error guarantee related to $F_2(D)$.

In Section 5.2 we first design a baseline solution that works for all *decomposable* summaries. A summary is *decomposable* if given the summaries for t data sets (bags of elements) $D_1, \ldots, D_t$ with error parameter ε, we can combine them together into a summary on $D_1 \uplus \cdots \uplus D_t$ with error parameter $O(\varepsilon)$, where $\uplus$ denotes multiset addition. All the F_1 and F_2 based summaries have this property and thus can be plugged into this solution. Assuming that we can combine the summaries with cost linear to their total size,

the resulting data structure has linear size and answers a summary query in $O(s_\varepsilon \log N)$ time, where s_ε is the size of the summary returned. It also works in external memory, with the query cost being $O(\frac{s_\varepsilon}{B} \log N)$ I/Os if $s_\varepsilon \geqslant B$ and $O(\log N / \log(B/s_\varepsilon))$ I/Os if $s_\varepsilon < B$. Note that this decomposable property has been exploited in many other works on maintaining summaries in the streaming context [7, 16, 22].

In Section 5.3 we improve upon this baseline solution by identifying another, stronger decomposable property of the F_1 based summaries, which we call *exponentially decomposable*. The size of the data structure remains linear, while its query cost improves to $O(\log N + s_\varepsilon)$. In external memory, the query cost is $O(\log_B N + s_\varepsilon/B)$ I/Os. This resembles the classical B-tree query cost, which includes an $O(\log_B N)$ search cost and an "output" cost of $O(s_\varepsilon/B)$, whereas the output in our case is a summary of size s_ε. This is clearly optimal (in the comparison model). For not-too-large summaries $s_\varepsilon = O(B)$, the query cost becomes just $O(\log_B N)$, the same as that for a simple aggregation query or a lookup on a B-tree.

In Section 5.4, we demonstrate how various summaries have the desired decomposable or exponentially decomposable property and thus can be plugged into our data structures. Finally we show how to support updates in Section 5.5.

5.2 A Baseline Solution

In this and the next section, we will describe our structures without instantiating with any particular summary. Instead we just use "ε-summary" as a placeholder for any summary with error parameter ε. Let $\mathcal{S}(\varepsilon, D)$ denote an ε-summary on data set D. We use s_ε to denote the size of an ε-summary[1].

Internal memory structure. Based on the decomposable property of a summary, a baseline solution can be designed using standard techniques. We first describe the internal memory structure. Sort all the N data records in the database on the A_q attribute and partition them into N/s_ε groups, each of size s_ε. Then we build a binary tree $\mathcal{T}$ on top of these groups, where each leaf (called a *fat leaf*) stores a group of s_ε records. For each internal node u of $\mathcal{T}$, let $\mathcal{T}_u$ denote the subtree of $\mathcal{T}$ rooted at u. We attach to u an ε-summary on the A_s attribute of all records stored in the subtree below u. Since each ε-summary has size s_ε and the number of internal nodes is $O(N/s_\varepsilon)$, the total size of the structure is $O(N)$. To answer a query $[q_1, q_2]$, we do a search on $\mathcal{T}$. It is well known that any range $[q_1, q_2]$ can be decomposed into $O(\log(N/s_\varepsilon))$ disjoint *canonical* subtrees

[1]Strictly speaking we should write $s_{\varepsilon,D}$. But as most ε-summaries have sizes independent of D, we drop the subscript D for brevity.

$\mathcal{T}_u$, plus at most two fat leaves that may partially overlap. We retrieve the ε-summaries attached to the roots of these subtrees. For each of the fat leaves, we simply read all the s_ε records stored there. Then we combine all of them into an $O(\varepsilon)$-summary for the entire query using the decomposable property. We can adjust ε by a constant factor in the construction to ensure that the output is an ε-summary. The total query time is thus the time required to combine the $O(\log(N/s_\varepsilon))$ summaries. For the Count-Min sketch and AMS sketch, the combining time is linear in the total size of the summaries, so the query time is $O(s_\varepsilon \log N)$. For the quantile summary and heavy hitters summary the query time becomes $O(s_\varepsilon \log N \log\log N)$[2] as we need to merge $O(\log(N/s_\varepsilon))$ sorted lists (Details in Section 5.4).

External memory data structure. The baseline solution easily extends to external memory. If $s_\varepsilon \geqslant B$, then each internal node and fat leaf occupies $\Theta(s_\varepsilon/B)$ blocks, so we can simply store each of them separately. The space is still linear and we load $O(\log N)$ nodes on each query. The query cost becomes $O(\frac{s_\varepsilon}{B} \log N)$ I/Os for the Count-Min and AMS sketch and $O(\frac{s_\varepsilon}{B} \log N \log_{M/B} \log N)$ I/Os for the quantile and heavy hitters summary.

For $s_\varepsilon < B$, each node occupies a fraction of a block, and we can pack multiple nodes in one block. We use a standard B-tree blocking of the tree $\mathcal{T}$ where each block contains $\Theta(B/s_\varepsilon)$ nodes, except possibly the root block. Thus each block stores a subtree of height $\Theta(\log(B/s_\varepsilon))$ of $\mathcal{T}$. Then standard analysis shows that the nodes we need to access are stored in $O(\log N/\log(B/s_\varepsilon))$ blocks. This implies a query cost of $O(\log_{B/s_\varepsilon} N)$ I/Os for the Count-Min and AMS sketch and $O(\log_{B/s_\varepsilon} N \log_{M/B}(\log_{B/s_\varepsilon} N))$ I/Os for the quantile and heavy hitters summary.

5.3 Optimal Data Structure for F_1 Based Summaries

The baseline solution of the previous section is not that impressive: Its "output" term has an extra $O(\log N)$ factor; in external memory, we are missing the ideal $O(\log_B N)$ term which is the main benefit of block accesses.

The main bottleneck in the baseline solution is not the search cost, but the fact that we need to assemble $O(\log N)$ summaries, each of size s_ε. In the absence of additional properties of the summary, it is impossible to make further improvement. Fortunately, we observe that many of the F_1 based summaries have what we call the *exponentially decomposable* property, which allows us to assemble summaries of exponentially decreasing

[2]In fact, an alternative solution achieves query time $O(s_\varepsilon \log N/\log\log N)$ by issuing s_ε range-quantile queries to the data structure in [18], but this solution does not work in external memory.

sizes. This turns out to be the key to optimality for data structure these summaries.

Definition 5.1 (Exponentially decomposable) *For $0 < \alpha < 1$, a summary $\mathcal{S}$ is α-exponentially decomposable if there exists a constant $c > 1$, such that for any t multisets $D_1, \ldots, D_t$ with their sizes satisfying $F_1(D_i) \leqslant \alpha^{i-1} F_1(D_1)$ for $i = 1, \ldots, t$, given $\mathcal{S}(\varepsilon, D_1), \mathcal{S}(c\varepsilon, D_2) \ldots, \mathcal{S}(c^{t-1}\varepsilon, D_t)$, (1) we can construct an $O(\varepsilon)$-summary for $D_1 \uplus \cdots \uplus D_t$; (2) the total size of $\mathcal{S}(\varepsilon, D_1), \ldots, \mathcal{S}(c^{t-1}\varepsilon, D_t)$ is $O(s_\varepsilon)$ and they can be combined in $O(s_\varepsilon)$ time; and (3) for any multiset D, the total size of $\mathcal{S}(\varepsilon, D), \ldots, \mathcal{S}(c^{t-1}\varepsilon, D)$ is $O(s_\varepsilon)$.*

Intuitively, since an F_1 based summary $\mathcal{S}(\varepsilon, D)$ provides an error bound of $\varepsilon|D|$, the total error from $\mathcal{S}(\varepsilon, D_1)$, $\mathcal{S}(c\varepsilon, D_2)$, $\ldots$, $\mathcal{S}(c^{t-1}\varepsilon, D_t)$ is

$$\begin{aligned} & \varepsilon|D_1| + c\varepsilon|D_2| + \cdots + c^{t-1}\varepsilon|D_t| \\ \leqslant\ & \varepsilon|D_1| + (c\alpha)\varepsilon|D_1| + \cdots + (c\alpha)^{t-1}\varepsilon|D_1|. \end{aligned}$$

If we choose c such that $c\alpha < 1$, then the error is bounded by $O(\varepsilon|D_1|)$, satisfying (1). Meanwhile, the F_1 based summaries usually have size $s_\varepsilon = \Theta(1/\varepsilon)$, so (2) and (3) can be satisfied, too. In Section 5.4 we will formally prove the α-exponentially decomposable property for all the F_1 based summaries mentioned in Section 5.1.2.

5.3.1 Optimal internal memory structure

Let $\mathcal{T}$ be the binary tree built on the A_q attribute as in the previous section. Without loss of generality we assume $\mathcal{T}$ is a complete balanced binary tree; otherwise we can always add at most N dummy records to make N/s_ε a power of 2 so that $\mathcal{T}$ is complete.

We first define some notation on $\mathcal{T}$. We use $\mathcal{S}(\varepsilon, u)$ to denote the ε-summary on the A_s attribute of all records stored in u's subtree. Fix an internal node u and a descendant v of u, let $\mathcal{P}(u, v)$ to be the set of nodes on the path from u to v, excluding u. Define the *left sibling set* of $\mathcal{P}(u, v)$ to be $\mathcal{L}(u, v) = \{w \mid w \text{ is a left child and has a right sibling} \in \mathcal{P}(u, v)\}$ and similarly the *right sibling set* of $\mathcal{P}(u, v)$ to be $\mathcal{R}(u, v) = \{w \mid w \text{ is a right child and has a left sibling} \in \mathcal{P}(u, v)\}$. To answer a query $[q_1, q_2]$, we first locate the two fat leaves a and b in $\mathcal{T}$ that contain q_1 and q_2, respectively. Let u be the lowest common ancestor of a and b. We call $\mathcal{P}(u, a)$ and $\mathcal{P}(u, b)$ the left and respectively the right query path. We observe that the subtrees rooted at the nodes in $\mathcal{R}(u, a) \cup \mathcal{L}(u, b)$ make up the canonical set for the query range $[q_1, q_2]$.

Focusing on $\mathcal{R}(u, a)$, let $w_1, \ldots, w_t$ be the nodes of $\mathcal{R}(u, a)$ and let $d_1 < \ldots < d_t$ denote their depths in $\mathcal{T}$ (the root of $\mathcal{T}$ is said to be at depth 0). Since $\mathcal{T}$ is a balanced binary tree, we have $F_1(w_i) \leqslant (1/2)^{d_i - d_1} F_1(w_1)$ for $i = 1, \ldots, t$. Here we use $F_1(w)$ to denote the first frequency moment (i.e., size) of the point set rooted at node w. Thus, if the

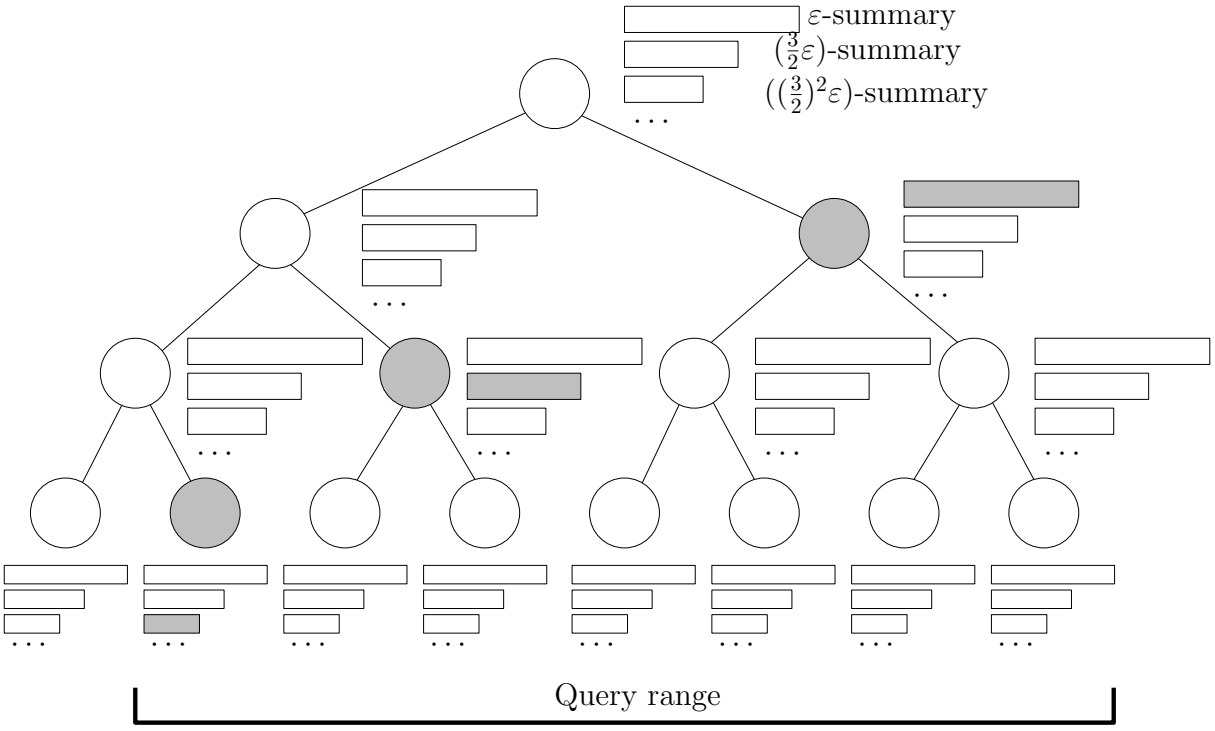

Figure 5.1: A schematic illustration of our internal memory structure. The grayed nodes form the canonical decomposition of the query range, and the grayed summaries are those we combine into the final summary for the queried data. In this example we use $c = \frac{3}{2}$.

summary is (1/2)-exponentially decomposable, and we have $\mathcal{S}(c^{d_i-d_1}\varepsilon, w_i)$ for $i = 1, \ldots, t$ at our disposal, we can combine them and form an $O(\varepsilon)$-summary for all the data covered by $w_1, \ldots, w_t$. We do the same for $\mathcal{L}(u, b)$. Finally, the two fat leaves can always supply the exact data (it is a summary with no error) of size $O(s_\varepsilon)$ in the query range. Plus the initial $O(\log N)$ search cost for locating $\mathcal{R}(u, a)$ and $\mathcal{L}(u, b)$, the query time now improves to the optimal $O(\log N + s_\varepsilon)$.

It only remains to show how to supply $\mathcal{S}(c^{d_i-d_1}\varepsilon, w_i)$ for each of the w_i's. In fact, we can afford to attach to each node $u \in \mathcal{T}$ all the summaries: $\mathcal{S}(\varepsilon, u), \mathcal{S}(c\varepsilon, u), \ldots \mathcal{S}(c^q\varepsilon, u)$ where q is an integer such that $s_{c^q\varepsilon} = O(1)$. Nicely, these summaries still have total size $O(s_\varepsilon)$ by the exponentially decomposable property, thus the space required by each node is still $O(s_\varepsilon)$ as in the previous section, and the total space remains linear. A schematic illustration of the overall structure is shown in Figure 5.1.

Theorem 5.2 *For any (1/2)-exponentially decomposable summary, a database $\mathcal{D}$ of N records can be stored in an internal memory structure of linear size so that a summary query can be answered in $O(\log N + s_\varepsilon)$ time.*

5.3.2 Optimal external memory data structure

In this section we show how to achieve the $O(\log_B N + s_\varepsilon/B)$-I/O query cost in external memory still with linear space. Here, the difficulty that we need to assemble $O(\log N)$ summaries lingers. In internal memory, we managed to get around it by the exponentially decomposable property so that the total size of these summaries is $O(s_\varepsilon)$. However, they still reside in $O(\log N)$ separate nodes. If we still use a standard B-tree blocking, for $s_\varepsilon \geqslant B$ we need to access $\Omega(\log N)$ blocks; for $s_\varepsilon < B$, we need to access $\Omega(\log N/\log(B/s_\varepsilon))$ blocks, neither of which is optimal. Below we first show how to achieve the optimal query cost by increasing the space to super-linear, then propose a packed structure to reduce the space back to linear.

Consider an internal node u and one of its descendants v. Let the sibling sets $\mathcal{R}(u,v)$ and $\mathcal{L}(u,v)$ be as previously defined. In the following we only describe how to handle the $\mathcal{R}(u,v)$'s; we will do the same for the $\mathcal{L}(u,v)$'s. Suppose $\mathcal{R}(u,v)$ contains nodes $w_1,\ldots,w_t$ at depths $d_1,\ldots,d_t$. We define the *summary set* for $\mathcal{R}(u,v)$ with error parameter ε to be

$$\mathcal{RS}(u,v,\varepsilon) = \{\mathcal{S}(\varepsilon,w_1),\mathcal{S}(c^{d_2-d_1}\varepsilon,w_2),\ldots,\mathcal{S}(c^{d_t-d_1}\varepsilon,w_t)\}.$$

The following two facts easily follow from the exponentially decomposable property.

Fact 1 *The total size of the summaries in $\mathcal{RS}(u,v,\varepsilon)$ is $O(s_\varepsilon)$;*

Fact 2 *The total size of all the summary sets $\mathcal{RS}(u,v,\varepsilon)$, $\mathcal{RS}(u,v,c\varepsilon)$,..., $\mathcal{RS}(u,v,c^t\varepsilon)$ is $O(s_\varepsilon)$.*

The data structure. We first build the binary tree $\mathcal{T}$ as before with a fat leaf size of s_ε. Before attaching any summaries, we block $\mathcal{T}$ in a standard B-tree fashion so that each block stores a subtree of $\mathcal{T}$ of size $\Theta(B)$, except possibly the root block which may contain 2 to B nodes of $\mathcal{T}$. The resulting blocked tree is essentially a B-tree where each leaf occupies $O(s_\varepsilon/B)$ blocks and each internal node occupies 1 block. Please see Figure 5.2 for an example of the standard B-tree blocking.

Consider an internal block $\mathcal{B}$ in the B-tree. Below we describe the additional structures we attach to $\mathcal{B}$. Let $\mathcal{T}_\mathcal{B}$ be the binary subtree of $\mathcal{T}$ stored in $\mathcal{B}$ and let $r_\mathcal{B}$ be the root of $\mathcal{T}_\mathcal{B}$. To achieve the optimal query cost, the summaries attached to the nodes of $\mathcal{T}_\mathcal{B}$ that we need to retrieve for answering any query must be stored consecutively, or in at most $O(1)$ consecutive chunks. Therefore, the idea is to store all the summaries for a query path in $\mathcal{T}_\mathcal{B}$ together, which is the reason we introduced the summary set $\mathcal{RS}(u,v,\varepsilon)$. The detailed structures that we attach to $\mathcal{B}$ are as follows:

1. For each internal node $u \in \mathcal{T}_\mathcal{B}$ and each leaf v in u's subtree in $\mathcal{T}_\mathcal{B}$, we store all summaries in $\mathcal{RS}(u,v,\varepsilon)$ sequentially.

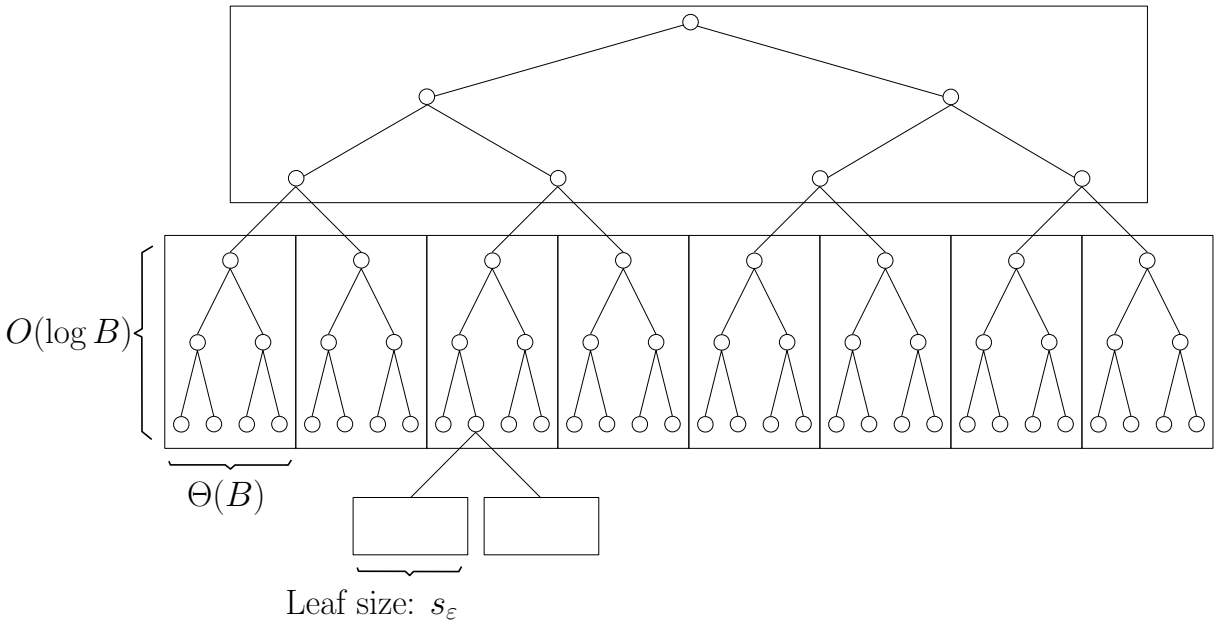

Figure 5.2: The standard B-tree blocking of a binary tree.

2. For each leaf v, we store the summaries in $\mathcal{RS}(r_\mathcal{B}, v, c^j\varepsilon)$ sequentially, for all $j = 0, \ldots, q$. Recall that q is an integer such that $s_{c^q\varepsilon} = O(1)$.

3. For the root $r_\mathcal{B}$, we store $\mathcal{S}(c^j\varepsilon, r_\mathcal{B})$ for $j = 0, \ldots, q$.

An illustration of the first and the second type of structures is shown in Figure 5.3. The size of the structure can be determined as follow:

1. For each leaf $v \in \mathcal{T}_\mathcal{B}$, there are at most $O(\log B)$ ancestors of v, so there are in total $O(B \log B)$ such pairs (u, v). For each pair we use $O(s_\varepsilon)$ space, so the space usage is $O(s_\varepsilon B \log B)$.

2. For each leaf $v \in \mathcal{T}_\mathcal{B}$ we use $O(s_\varepsilon)$ space, so the space usage is $O(s_\varepsilon B)$.

3. For the root $r_\mathcal{B}$, the space usage is $O(s_\varepsilon)$.

Summing up the above cases, the space for storing the summaries of any internal block $\mathcal{B}$ is $O(s_\varepsilon B \log B)$. Note that each internal block has fanout $\Theta(B)$, and each leaf has size $\Theta(s_\varepsilon)$, so there are in total at most $O(N/(Bs_\varepsilon))$ internal blocks, and thus the total space usage is $O(N \log B)$. Next we show that this structure can indeed be used to answer queries in the optimal $O(\log_B N + s_\varepsilon/B)$ I/Os.

Query procedure. Given a query range $[q_1, q_2]$, let a and b be the two leaves containing q_1 and q_2, respectively. We focus on how to retrieve the necessary summaries for the right sibling set $\mathcal{R}(u, a)$, where u is the lowest common ancestor of a and b; the left sibling

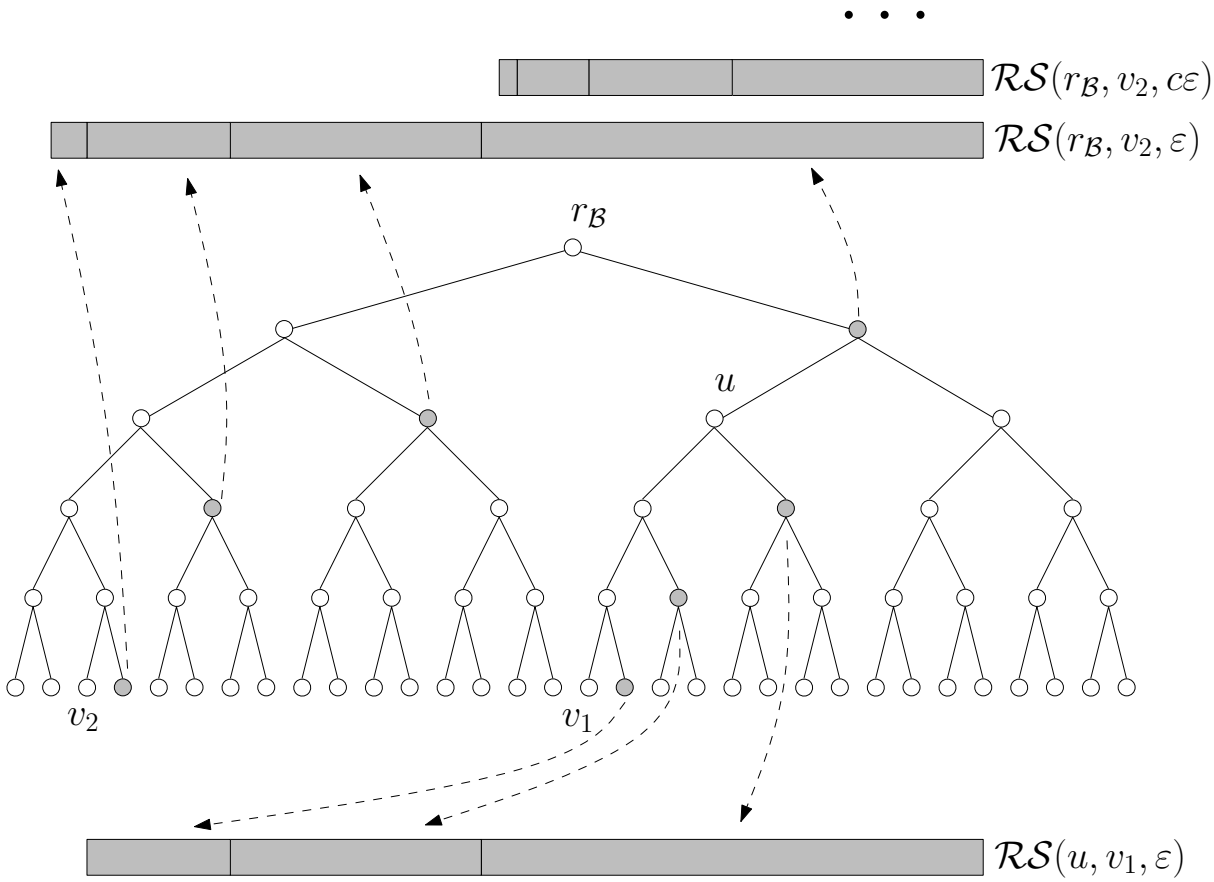

Figure 5.3: The summaries we store for an internal block $\mathcal{B}$.

set $\mathcal{L}(u, b)$ can be handled symmetrically. By the previous analysis, we need exactly the summaries in $\mathcal{RS}(u, a, \varepsilon)$. Recall that $\mathcal{R}(u, a)$ are the right siblings of the left query path $\mathcal{P}(u, a)$. Let $\mathcal{B}_0, \ldots, \mathcal{B}_l$ be the blocks that $\mathcal{P}(u, a)$ intersects from u to a. The path $\mathcal{P}(u, a)$ is partitioned into $l+1$ segments by these $l+1$ blocks. Let $\mathcal{P}(u, v_0), \mathcal{P}(r_1, v_1), \ldots, \mathcal{P}(r_l, v_l = a)$ be the $l + 1$ segments, with r_i being the root of the binary tree $\mathcal{T}_{\mathcal{B}_i}$ in block $\mathcal{B}_i$ and v_i being a leaf of $\mathcal{T}_{\mathcal{B}_i}$, $i = 0, \ldots, l$. Let $w_1, \ldots, w_t$ be the nodes in $\mathcal{R}(u, a)$, at depths $d_1, \ldots, d_t$ of $\mathcal{T}$. We claim that w_i is either a node of $\mathcal{T}_{\mathcal{B}_k}$ for some $k \in \{0, \ldots, l\}$, or a right sibling of r_k for some $k \in \{0, \ldots, l\}$, which makes w_i a root of some other block. This is because by the definition of $\mathcal{R}(u, a)$, we know that w_i is a right child whose left sibling is in some $\mathcal{B}_k$. If w_i is not in $\mathcal{B}_k$, it must be the root of some other block. Recall that we need to retrieve $\mathcal{S}(c^{d_i - d_1}\varepsilon, w_i)$ for $i = 1, \ldots, t$. Below we show how this can be done efficiently using our structure.

For the w_i's in the first block $\mathcal{B}_0$, since we have stored all summaries in $\mathcal{RS}(u, v_0, \varepsilon)$ sequentially for $\mathcal{B}_0$ (case 1.), they can be retrieved in $O(1 + s_\varepsilon/B)$ I/Os.

For any w_i being the root of some other block $\mathcal{B}'$ not on the path $\mathcal{B}_0, \ldots, \mathcal{B}_l$, since we have stored the summaries $\mathcal{S}(c^j\varepsilon, w_i)$ for $j = 0, \ldots, q$ for every block (case 3.), the required summary $\mathcal{S}(c^{d_i - d_1}\varepsilon, w_i)$ can be retrieved in $O(1 + s_{c^{d_i - d_1}\varepsilon}/B)$ I/Os. Note that the

number of such w_i's is bounded by $O(\log_B N)$, so the total cost for retrieving summaries for these nodes is at most $O(\log_B N + s_\varepsilon/B)$ I/Os.

The rest of the w_i's are in $\mathcal{B}_1, \ldots, \mathcal{B}_l$. Consider each $\mathcal{B}_k, k = 1, \ldots, l$. Recall that the segment of the path $\mathcal{P}(u, a)$ in $\mathcal{B}_k$ is $\mathcal{P}(r_k, v_k)$, and the w_i's in $\mathcal{B}_k$ are exactly $\mathcal{R}(r_k, v_k)$. We have stored $\mathcal{RS}(r_k, v_k, c^j\varepsilon)$ for $\mathcal{B}_k$ for all j (case 2.), so no matter at which relative depths $d_i - d_1$ the nodes in $\mathcal{R}(r_k, v_k)$ start and end, we can always find the required summary set. Retrieving the desired summary set takes $O\left(1 + s_{c^{d'-d_1}\varepsilon}/B\right)$ I/Os, where d' is the depth of the highest node in $\mathcal{R}(r_k, v_k)$. Summing over all blocks $\mathcal{B}_1, \ldots, \mathcal{B}_l$, the total cost is $O(\log_B N + s_\varepsilon/B)$ I/Os.

Reducing the size to linear. The structure above has a super-linear size $O(N \log B)$. Next we show how to reduce its size back to $O(N)$ while not affecting the optimal query cost.

Observe that the $\log B$ factor comes from case 1., where we store $\mathcal{RS}(u, v, \varepsilon)$ for each internal node u and each leaf v in u's subtree in u's block $\mathcal{B}$. Focus on $\mathcal{B}$ and the binary tree $\mathcal{T}_\mathcal{B}$ stored in it. Abusing notation, we use $\mathcal{T}_u$ to denote the subtree rooted at u in $\mathcal{T}_\mathcal{B}$. Assume $\mathcal{T}_u$ has height h (in $\mathcal{T}_\mathcal{B}$). Our idea is to pack the $\mathcal{RS}(u, v, \varepsilon)$'s for some leaves $v \in \mathcal{T}_u$ to reduce the space usage. Let u_l and u_r be the left and right child of u, respectively. The first observation is that we only need to store $\mathcal{RS}(u, v, \varepsilon)$ for each leaf v in u_l's subtree. This is because for any leaf v in u_r's subtree, the sibling set $\mathcal{R}(u, v)$ is the same as $\mathcal{R}(u_r, v)$, so $\mathcal{RS}(u, v, \varepsilon) = \mathcal{RS}(u_r, v, \varepsilon)$, which will be stored when considering u_r in place of u. For any leaf v in u_l's subtree, observe that the highest node in $\mathcal{R}(u, v)$ is u_r. This means for a node $w \in \mathcal{R}(u, v)$ with height i in tree $\mathcal{T}_u$, the summary for w in $\mathcal{RS}(u, v, \varepsilon)$ is $\mathcal{S}(c^{h-i-1}\varepsilon, w)$. Let u' be an internal node in u_l's subtree, and suppose u' has k_h leaves below it. We will decide later the value of k_h and, thus, the height $\log k_h$ at which u' is chosen (the leaves are defined to be at height 0). We do the following for each u' at height $\log k_h$ in u_l's subtree. Instead of storing the summary set $\mathcal{RS}(u, v, \varepsilon)$ for each leaf v in u''s subtree, we store $\mathcal{RS}(u, u', \varepsilon)$, which is the common prefix of all the $\mathcal{RS}(u, v, \varepsilon)$'s, together with a summary for each of the nodes in u''s subtree. More precisely, for each node w in u''s subtree, if its height is i, we store a summary $\mathcal{S}(c^{h-i-1}\varepsilon, w)$. All these summaries below u' are stored sequentially. A schematic illustration of our packed structure is shown in Figure 5.4.

Recall that all the summary sets we store in case 1. are used to cover the top portion of the query path $\mathcal{P}(u, v_0)$ in block $\mathcal{B}_0$, i.e., $\mathcal{RS}(u, v_0, \varepsilon)$. Clearly the packed structure still serves this purpose: We first find the u' which has v_0 as one of its descendants. Then we load $\mathcal{RS}(u, u', \varepsilon)$, followed by the summaries $\mathcal{S}(c^{h-i-1}, w)$ required in $\mathcal{RS}(u, v_0, \varepsilon)$. Loading $\mathcal{RS}(u, u', \varepsilon)$ still takes $O(1 + s_\varepsilon/B)$ I/Os, but loading the remaining individual

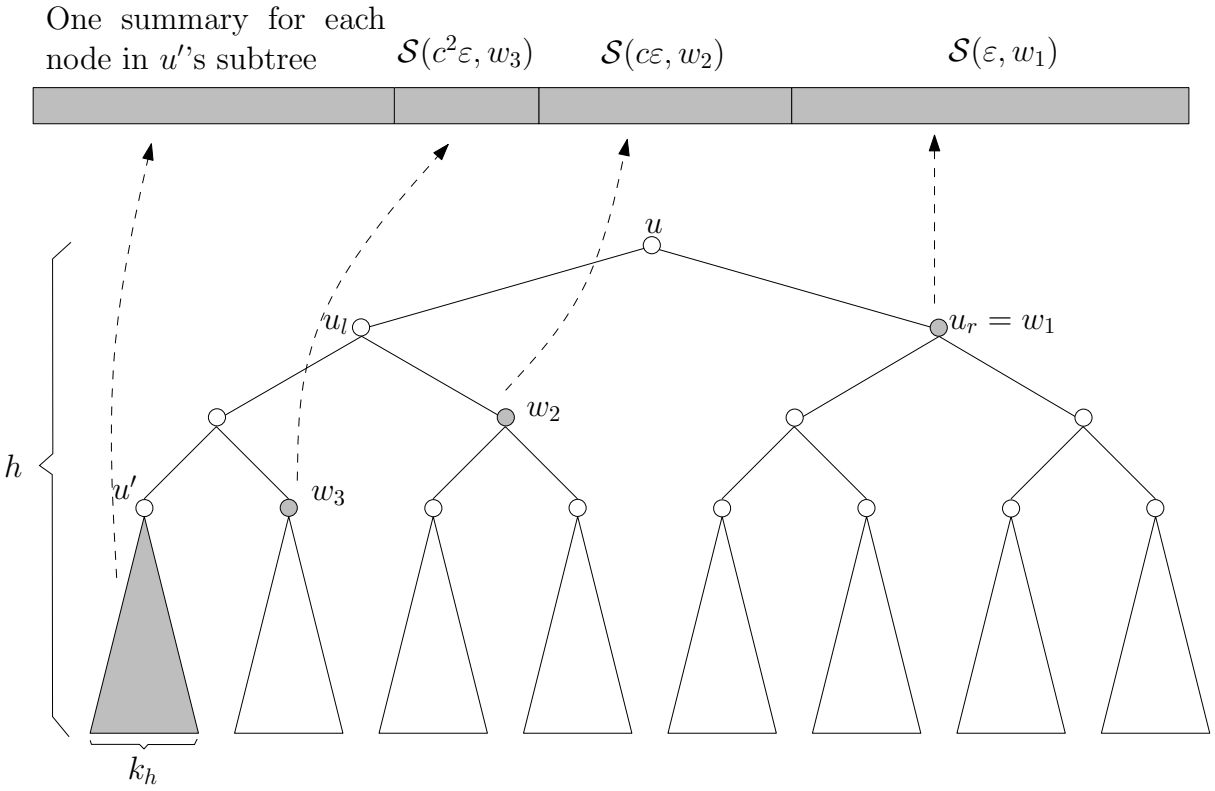

Figure 5.4: A schematic illustration of our packed structure.

summaries may incur many I/Os since they may not be stored sequentially. Nevertheless, if we ensure that all the individual summaries below u' have total size $O(s_\varepsilon)$, then loading any subset of them does not take more than $O(1 + s_\varepsilon/B)$ I/Os. Note that there are $k_h/2^i$ nodes at height i in u's subtree, the total size of all summaries below u' is

$$\sum_{i=0}^{\log k_h} \frac{k_h}{2^i} s_{c^{h-i-1}\varepsilon}. \tag{5.1}$$

Thus it is sufficient to choose k_h such that (5.1) is $\Theta(s_\varepsilon)$. Note that such a k_h always exists[3]: When $k_h = 1$, (5.1) is $s_{c^{h-1}\varepsilon} = O(s_\varepsilon)$; when k_h takes the maximum possible value $k_h = 2^{h-1}$, the last term (when $i = h$) in the summation of (5.1) is s_ε, so (5.1) is at least $\Omega(s_\varepsilon)$; every time k_h doubles, (5.1) increases by at most $O(s_\varepsilon)$.

It only remains to show that by employing the packed structure, the space usage for a block is indeed $O(Bs_\varepsilon)$. For a node u at height h in $\mathcal{T}_\mathcal{B}$, the number of u''s at height $\log k_h$ under u is $2^h/k_h$. For each such u', storing $\mathcal{RS}(u, u', \varepsilon)$, as well as all the individual summaries below u', takes $O(s_\varepsilon)$ space. So the space required for node u is $O(2^h s_\varepsilon/k_h)$.

[3]We define k_h in this implicit way for its generality. When instantiating into specific summaries, there are often closed forms for k_h. For example when $s_\varepsilon = \Theta(1/\varepsilon)$ and $1 < c < 2$, $k_h = \Theta(c^h)$.

There are $O(B/2^h)$ nodes u at height h. Thus the total space required is

$$O\left(\sum_{h=1}^{\log B} 2^h s_\varepsilon/k_h \cdot B/2^h\right) \;=\; O\left(\sum_{h=1}^{\log B} B s_\varepsilon/k_h\right).$$

Note that the choice of k_h implies that

$$s_\varepsilon/k_h = O\left(\sum_{i=0}^{\log k_h} \frac{1}{2^i} s_{c^{h-i-1}\varepsilon}\right) = O\left(\sum_{i=0}^{h-1} \frac{1}{2^i} s_{c^{h-i-1}\varepsilon}\right),$$

so the total size of the packed structures in $\mathcal{B}$ is bounded by

$$\begin{aligned}
\sum_{h=1}^{\log B} B s_\varepsilon/k_h &\leqslant B \sum_{h=0}^{\log B} \sum_{i=0}^{h-1} \frac{1}{2^i} s_{c^{h-i-1}\varepsilon} \\
&= B \sum_{h=0}^{\log B} \sum_{i=0}^{h-1} \frac{1}{2^{h-i-1}} s_{c^i\varepsilon} \\
&\leqslant B \sum_{i=0}^{\log B} s_{c^i\varepsilon} \sum_{h=i}^{\log B} \frac{1}{2^{h-i-1}} \\
&\leqslant 2B \sum_{i=0}^{\log B} s_{c^i\varepsilon} \\
&= O(B s_\varepsilon).
\end{aligned}$$

Theorem 5.3 *For any (1/2)-exponentially decomposable summary, a database $\mathcal{D}$ of N records can be stored in an external memory data structure of linear size so that a summary query can be answered in $O(\log_B N + s_\varepsilon/B)$ I/Os.*

Remark. One technical subtlety is that the $O(s_\varepsilon)$ combining time in internal memory does not guarantee that we can combine the $O(\log N)$ summaries in $O(s_\varepsilon/B)$ I/Os in external memory. However if the merging algorithm only makes linear scans on the summaries, then this is not a problem, as we shall see in Section 5.4.

5.4 Summaries

In this section we demonstrate the decomposable or exponentially decomposable properties for the summaries mentioned in Section 5.1.2. Thus, they can be used in our data structures in Section 5.2 and 5.3.

5.4.1 Heavy hitters

Given a multiset D, let $f_D(x)$ be the frequency of x in D. The MG summary [56] with error parameter ε consists of $s_\varepsilon = 1/\varepsilon$ items and their associated counters. For any item

x in the counter set, the MG summary maintains an estimated count $\hat{f}_D(x)$ such that $f_D(x) - \varepsilon F_1(D) \leqslant \hat{f}_D(x) \leqslant f_D(x)$; for any item x not in the counter set, it is guaranteed that $f_D(x) \leqslant \varepsilon F_1(D)$. Thus in either case, the MG summary provides an additive $\varepsilon F_1(D)$ error: $f_D(x) - \varepsilon F_1(D) \leqslant \hat{f}_D(x) \leqslant f_D(x)$ for any x. The SpaceSaving summary is very similar to the MG summary except that the SpaceSaving summary provides an $\hat{f}_D(x)$ overestimating $f_D(x)$: $f_D(x) \leqslant \hat{f}_D(x) < f_D(x) + \varepsilon F_1(D)$. Thus they clearly solve the heavy hitters problem.

The MG summary is clearly decomposable. Below we show that it is also α-exponentially decomposable for any $0 < \alpha < 1$. The same proof also works for the SpaceSaving summary.

Consider t multisets $D_1, \ldots, D_t$ with $F_1(D_i) \leqslant \alpha^{i-1} F_1(D_1)$ for $i = 1, \ldots, t$. We set $c = 1/\sqrt{\alpha} > 1$. Given a series of MG summaries $\mathcal{S}(\varepsilon, D_1)$, $\mathcal{S}(c\varepsilon, D_2)$, $\ldots$, $\mathcal{S}(c^{t-1}\varepsilon, D_t)$, we combine them by adding up the counters for the same item. Note that the total size of these summaries is bounded by

$$\sum_{j=0}^{t-1} s_{c^j\varepsilon} = \sum_{j=0}^{t-1} \frac{1}{c^j \varepsilon} = O(1/\varepsilon) = O(s_\varepsilon).$$

In order to analyze the error in the combined summary, let $f_j(x)$ denote the true frequency of item x in D_j and $\hat{f}_j(x)$ be the estimator of $f_j(x)$ in $\mathcal{S}(c^{j-1}\varepsilon, D_j)$. The combined summary uses $\sum_{j=1}^{t} \hat{f}_j(x)$ to estimate the true frequency of x, which is $\sum_{j=1}^{t} f_j(x)$. Note that

$$f_j(x) \geqslant \hat{f}_j(x) \geqslant f_j(x) - c^{j-1}\varepsilon F_1(D_j)$$

for $j = 1, \ldots, t$. Summing up the first inequality over all j yields $\sum_{j=1}^{t} f_j(x) \geqslant \sum_{j=1}^{t} \hat{f}_j(x)$. For the second inequality, we have

$$\begin{aligned}
\sum_{j=1}^{t} \hat{f}_j(x) &\geqslant \sum_{j=1}^{t} f_j(x) - \sum_{j=1}^{t} c^{j-1}\varepsilon F_1(D_j) \\
&\geqslant \sum_{j=1}^{t} f_j(x) - \sum_{j=1}^{t} (\frac{\alpha}{\sqrt{\alpha}})^{j-1} \varepsilon F_1(D_1) \\
&\geqslant \sum_{j=1}^{t} f_j(x) - \varepsilon F_1(D_1) \sum_{j=1}^{t} (\sqrt{\alpha})^{j-1} \\
&= \sum_{j=1}^{t} f_j(x) - O(\varepsilon F_1(D_1)).
\end{aligned}$$

Therefore the error bound is $O(\varepsilon F_1(D_1)) = O(\varepsilon(F_1(D_1 \uplus \cdots \uplus D_t))$.

To combine the summaries we require that each summary maintains its (item, counter) pairs in the increasing order of items (we impose an arbitrary ordering if the items are from an unordered domain). In this case each summary can be viewed as a sorted list

and we can merge the t sorted lists into a single list, where the counters for the same item are added up. Note that if each summary is of size s_ε, then we need to employ a t-way merging algorithm and it takes $O(s_\varepsilon t \log t)$ time in internal memory and $O(\frac{s_\varepsilon t}{B} \log_{M/B} t)$ I/Os in external memory. However, when the sizes of the t summaries form a geometrically decreasing sequence, we can repeatedly perform two-way merges in a bottom-up fashion with linear total cost. The merging algorithm starts with an empty list, at step i, it merges the current list with the summary $\mathcal{S}(\varepsilon_{t+1-i}, D_{t+1-i})$. Note that in this process every counter of $\mathcal{S}(\varepsilon_j, D_j)$ is merged j times, but since the size of $\mathcal{S}(\varepsilon_j, D_j)$ is $\frac{1}{c^{j-1}\varepsilon}$, the total running time is bounded by

$$\sum_{j=1}^{t} \frac{j}{c^{j-1}\varepsilon} = O\left(\frac{1}{\varepsilon}\right) = O(s_\varepsilon).$$

In external memory we can perform the same trick and achieve the $O(s_\varepsilon/B)$ I/O bound if the smallest summary $\mathcal{S}(c^{t-1}\varepsilon, D_t)$ has size $\frac{1}{c^{t-1}\varepsilon} > B$; otherwise we can take the smallest k summaries, where k is the maximum number such that the smallest k summaries can fit in one block, and merge them in the main memory. In either case, we can merge the t summaries in s_ε/B I/Os.

5.4.2 Quantiles

Recall that in the ε-approximate quantile problem, we are given a set D of N items from a totally ordered universe, and the goal is to have a summary $\mathcal{S}(\varepsilon, D)$ from which for any $0 < \phi < 1$, a record with rank in $[(\phi - \varepsilon)N, (\phi + \varepsilon)N]$ can be extracted. It is easy to obtain a quantile summary of size $O(1/\varepsilon)$: We simply sort D and take an item every εN consecutive items. Given any rank $r = \phi N$, there is always an element within rank $[r - \varepsilon N, r + \varepsilon N]$.

Below we show that quantile summaries are α-exponentially decomposable. Suppose we are given a series of such quantile summaries $\mathcal{S}(\varepsilon_1, D_1), \mathcal{S}(\varepsilon_2, D_2), \ldots, \mathcal{S}(\varepsilon_t, D_t)$, for data sets $D_1, \ldots, D_t$. We combine them by sorting all the items in these summaries. We claim this forms an approximate quantile summary for $D = D_1 \cup \cdots \cup D_t$ with error at most $\sum_{j=1}^{t} \varepsilon_j F_1(D_j)$, that is, given a rank r, we can find an item in the combined summary whose rank is in $[r - \sum_{j=1}^{t} \varepsilon_j F_1(D_j), r + \sum_{j=1}^{t} \varepsilon_j F_1(D_j)]$ in D. For an element x in the combined summary, let y_j and z_j be the two consecutive elements in $\mathcal{S}(\varepsilon_j, D_j)$ such that $y_j \leqslant x \leqslant z_j$. We define $r_j^{\min}(x)$ to be the rank of y_j in D_j and $r_j^{\max}(x)$ to be rank of z_j in D_j. In other words, $r_j^{\min}(x)$ (resp. $r_j^{\max}(x)$) is the minimum (resp. maximum) possible rank of x in D_j. We state the following lemma that describes the properties of $r_j^{\min}(x)$ and $r_j^{\max}(x)$:

Lemma 5.4 *(1) For an element x in the combined summary,*

$$\sum_{j=1}^{t} r_j^{\max}(x) - \sum_{j=1}^{t} r_j^{\min}(x) \leqslant \sum_{j=1}^{t} \varepsilon_j F_1(D_j).$$

(2) For two consecutive elements $x_1 \leqslant x_2$ in the combined summary,

$$\sum_{j=1}^{t} r_j^{\min}(x_2) - \sum_{j=1}^{t} r_j^{\min}(x_1) \leqslant \sum_{j=1}^{t} \varepsilon_j F_1(D_j).$$

Proof: Since $r_j^{\max}(x)$ and $r_j^{\min}(x)$ are the local ranks of two consecutive elements in $\mathcal{S}(\varepsilon_j, D_j)$, we have $r_j^{\max}(x) - r_j^{\min}(x) \leqslant \varepsilon_j F_1(D_j)$. Taking summation over all j, part (1) of the lemma follows. We also note that if x_1 and x_2 are consecutive in the combined summary, $r_j^{\min}(x_1)$ and $r_j^{\min}(x_2)$ are local ranks of either the same element or two consecutive elements of $\mathcal{S}(\varepsilon_j, D_j)$. In either case we have $r_j^{\min}(x_2) - r_j^{\min}(x_1) \leqslant \varepsilon_j F_1(D_j)$. Summing over all j proves part (2) of the lemma. □

Now for each element x in the combined summary, we compute the global minimum rank $r^{\min}(x) = \sum_{j=1}^{t} r_j^{\min}(x)$. Note that all these global ranks can be computed by scanning the combined summary in sorted order. Given a query rank r, we find the smallest element x with $r^{\min}(x) \geqslant r - \sum_{j=1}^{t} \varepsilon_j F_1(D_j)$. We claim that the actual rank of x in D is in the range $[r - \sum_{j=1}^{t} \varepsilon_j F_1(D_j), r + \sum_{j=1}^{t} \varepsilon_j F_1(D_j)]$. Indeed, we observe that the actual rank of x in set D is in the range $[\sum_{j=1}^{t} r_j^{\min}(x), \sum_{j=1}^{t} r_j^{\max}(x)]$ so we only need to prove that this range is contained by $[r - \sum_{j=1}^{t} \varepsilon_j F_1(D_j), r + \sum_{j=1}^{t} \varepsilon_j F_1(D_j)]$. The left side trivially follows from the choice of x. For the right side, let x' be the largest element in the new summary such that $x' \leqslant x$. By the choice of x, we have $\sum_{j=1}^{t} r_j^{\min}(x') < r - \sum_{j=1}^{t} \varepsilon_j F_1(D_j)$. By Lemma 5.4 we have $\sum_{j=1}^{t} r_j^{\min}(x) - \sum_{j=1}^{t} r_j^{\min}(x') \leqslant \sum_{j=1}^{t} \varepsilon_j F_1(D_j)$ and $\sum_{j=1}^{t} r_j^{\max}(x) - \sum_{j=1}^{t} r_j^{\min}(x) \leqslant \sum_{j=1}^{t} \varepsilon_j F_1(P_j)$. Summing up these three inequalities yields $\sum_{j=1}^{t} r_j^{\max}(x) \leqslant r + \sum_{j=1}^{t} \varepsilon_j F_1(D_j)$, so the claim follows.

For α-exponentially decomposability, the t data sets have $F_1(D_i) \leqslant \alpha^{i-1} F_1(D_1)$ for $i = 1, \ldots, t$. We choose $c = 1/\sqrt{\alpha} > 1$. The summaries $\mathcal{S}(\varepsilon_1, D_1), \mathcal{S}(\varepsilon_2, D_2), \ldots, \mathcal{S}(\varepsilon_t, D_t)$ have $\varepsilon_i = c^{i-1}\varepsilon$. Therefore we can combine them with error

$$\begin{aligned}
\sum_{j=1}^{t} c^{j-1} \varepsilon F_1(D_j) &\leqslant \sum_{j=1}^{t} \left(\frac{\alpha}{\sqrt{\alpha}}\right)^{j-1} \varepsilon F_1(D_1) \\
&= \varepsilon F_1(D_1) \sum_{j=1}^{t} \left(\sqrt{\alpha}\right)^{j-1} \\
&= O(\varepsilon F_1(D_1)) \\
&= O(\varepsilon F_1(D_1 \cup \cdots \cup D_t)).
\end{aligned}$$

To combine the t summaries, we notice that we are essentially merging k sorted lists with geometrically decreasing sizes, so we can adapt the algorithm in Section 5.4.1. The

cost of merging the t summaries is therefore $O(s_\varepsilon)$ in internal memory and $O(s_\varepsilon/B)$ I/Os in external memory. The size of combined summary is

$$\sum_{j=1}^{t} \frac{1}{c^{j-1}\varepsilon} = O\left(\frac{1}{\varepsilon}\right) = O(s_\varepsilon).$$

5.4.3 The Count-Min sketch

Given a multiset D where the items are drawn from a universe $[U] = \{1, \ldots, U\}$. Let $f_D(x)$ be the frequency of x in D. The Count-Min sketch makes use of a 2-universal hash function $h : [U] \to [1/\varepsilon]$ and a collection of $1/\varepsilon$ counters $C[1], \ldots, C[1/\varepsilon]$. Then it computes $C[j] = \sum_{h(x)=j} f_D(x)$ for $j = 1, \ldots, 1/\varepsilon$. A single collection of $1/\varepsilon$ counters achieve a constant success probability for a variety of estimation purposes, and the probability can be boosted to $1-\delta$ by using $O(\log(1/\delta))$ copies with independent hash functions. Here we only show the decomposability of a single copy; the same result also holds for $O(\log(1/\delta))$ copies.

Given multiple Count-Min sketches with the same h (hence the same number of counters), they can be easily combined by adding up the corresponding counters. So the Count-Min sketch is decomposable. However, for exponentially decomposability we are dealing with t Count-Min sketches with exponentially increasing ε's, hence different hash functions, so they cannot be easily combined. Thus we simply put them together without combining any counters. Although the resulting summary is not a true Count-Min sketch, we argue that it can be used to serve all the purposes a Count-Min is supposed to serve.

More precisely, for t data sets $D_1, \ldots, D_t$ with $F_1(D_i) \leqslant \alpha^{i-1} F_1(D_1)$, we have t Count-Min sketches $\mathcal{S}(\varepsilon, D_1), \ldots, \mathcal{S}(c^{t-1}\varepsilon, D_t)$. The i-th sketch $\mathcal{S}(c^{j-1}\varepsilon, D_t)$ uses a hash function $h_i : [U] \to [1/c^{j-1}\varepsilon]$. Again we set $c = 1/\sqrt{\alpha}$. Note that the total size of all the sketches is $O(1/\varepsilon + 1/c\varepsilon + 1/c^2\varepsilon + \cdots) = O(1/\varepsilon) = O(s_\varepsilon)$, so we only need to show that the error is the same as what a Count-Min sketch $\mathcal{S}(\varepsilon, D_1 \uplus \cdots \uplus D_t)$ would provide. Below we consider the problem of estimating inner products (join sizes), which has other applications, such as point queries and self-join sizes, as special cases.

Let $\mathbf{f}_i$ denote the frequency vector of D_i, and let $\mathbf{f} = \sum_{i=1}^{t} \mathbf{f}_i$ be the frequency vector of $D = D_1 \uplus \ldots \uplus D_t$. The goal is to estimate inner product $\langle \mathbf{f}, \mathbf{g} \rangle$ where $\mathbf{g}$ is the frequency vector of some other data set. Note that when $\mathbf{g}$ is a standard basic vector (i.e., containing only one "1"), $\langle \mathbf{f}, \mathbf{g} \rangle$ becomes a point query; when $\mathbf{g} = \mathbf{f}$, $\langle \mathbf{f}, \mathbf{g} \rangle$ is the self-join size of $\mathbf{f}$. We distinguish between two cases: (1) $\mathbf{g}$ is given explicitly; and (2) $\mathbf{g}$ is also represented by a summary returned by our data structure, i.e., a collection of t Count-Min sketches $\mathcal{S}(\varepsilon, G_1), \ldots, \mathcal{S}(c^{t-1}\varepsilon, G_t)$, where $\mathbf{g} = \sum_{i=1}^{t} \mathbf{g}_i$ and $\mathbf{g}_i$ is the frequency vector of G_i. Recall that the Count-Min sketch estimates $\langle \mathbf{f}, \mathbf{g} \rangle$ with an additive error of $\varepsilon F_1(\mathbf{f}) F_1(\mathbf{g})$, and we will show that we can do the same when $\mathbf{f}$ is represented by the collection of t Count-Min

sketches.

Inner product with an explicit vector. For a $\mathbf{g}$ given explicitly, we can construct a Count-Min sketch $\mathcal{S}(c^{i-1}\varepsilon, \mathbf{g})$ for $\mathbf{g}$ with hash function h_i, for $i = 1, \ldots, t$. We observe that $\langle \mathbf{f}, \mathbf{g} \rangle$ can be expressed as $\sum_{i=1}^{t} \langle \mathbf{f}_i, \mathbf{g} \rangle$, and $\langle \mathbf{f}_i, \mathbf{g} \rangle$ can be estimated using $\mathcal{S}(c^{i-1}\varepsilon, D_i)$ and $\mathcal{S}(c^{i-1}\varepsilon, \mathbf{g})$ as described in [22] since they use the same hash function. The error is $c^{i-1}\varepsilon ||\mathbf{f}_i||_1 ||\mathbf{g}||_1 \leqslant (c\alpha)^{i-1}\varepsilon ||\mathbf{f}_1||_1 ||\mathbf{g}||_1$. For $c = 1/\sqrt{\alpha}$, the total error is bounded by

$$\sum_{i=1}^{t} \alpha^{(i-1)/2} \varepsilon ||\mathbf{f}_1||_1 ||\mathbf{g}||_1 = O(\varepsilon ||\mathbf{f}_1||_1 ||\mathbf{g}||_1) = O\left(\varepsilon F_1(\mathbf{f}) F_1(\mathbf{g})\right),$$

as desired.

Inner product with a vector returned by a summary query. Next we consider the case where $\mathbf{g}$ is also represented by a series[4] of t Count-Min sketches $\mathcal{S}(\varepsilon, G_1), \ldots, \mathcal{S}(c^{t-1}\varepsilon, G_t)$ with $F_1(G_i) \geqslant \alpha^{i-1} F_1(G_1)$. We will show how to estimate $\langle \mathbf{f}, \mathbf{g} \rangle$ using the two series of sketches. This will allow the user to estimate the join size between the results of two queries. Note that this includes the special case of estimating the self-join size of $\mathbf{f}$.

In this case we will inevitably face the problem of pairing two sketches of different sizes. To do so we need more insight into the hash functions used. Suppose $1/\varepsilon$ is a power of 2. Let p be a prime within the range $[U, 2U]$ and a, b be random numbers uniformly chosen from $\{0, \ldots, p-1\}$. If we use the following 2-universal hash functions:

$$h_i(x) = ((ax + b) \bmod p)) \bmod \frac{1}{2^{i-1}\varepsilon}, i = 1, \ldots, \log(1/\varepsilon),$$

then we observe that each bucket of h_{i+1} is partitioned into two buckets of h_i. This means that given a Count-Min sketch $\mathcal{S}(2^{i-1}\varepsilon, D)$ constructed with h_i, one can convert it to a Count-Min sketch $\mathcal{S}(2^{j-1}\varepsilon, D)$ constructed with h_j for any $j \geqslant i$. Thus two sketches of different sizes can still be used together by reducing the size of the larger one to match that of the smaller one. Of course we will only get the error guarantee of the smaller sketch, but this will not be a problem as we show later.

Now, set $c = 2$ and we have the sketches $\mathcal{S}(2^{i-1}\varepsilon, D_i)$ and $\mathcal{S}(2^{i-1}\varepsilon, G_i)$ with hash function h_i, for $i = 1, \ldots, \log(1/\varepsilon)$. We express $\langle \mathbf{f}, \mathbf{g} \rangle$ as

$$\langle \mathbf{f}, \mathbf{g} \rangle = \left\langle \sum_{i=1}^{t} \mathbf{f}_i, \sum_{i=1}^{t} \mathbf{g}_i \right\rangle = \sum_{i=1}^{t} \langle \mathbf{f}_i, \mathbf{g}_i \rangle + \sum_{i<j} \langle \mathbf{f}_i, \mathbf{g}_j \rangle + \sum_{i<j} \langle \mathbf{g}_i, \mathbf{f}_j \rangle.$$

[4]More precisely, $\mathbf{g}$ is represented by two such series: one from the left query path and one from the right query path, and so is $\mathbf{f}$. But we can decompose $\langle \mathbf{f}, \mathbf{g} \rangle$ into 4 subproblems by considering the cross product of these series, where each subproblem involves only a single series of sketches for either $\mathbf{f}$ or $\mathbf{g}$.

First, $\langle \mathbf{f}_i, \mathbf{g}_i \rangle$ can be estimated using $\mathcal{S}(2^{i-1}\varepsilon, D_i)$ and $\mathcal{S}(2^{i-1}\varepsilon, G_i)$. The error is at most $2^{i-1}\varepsilon F_1(D_i)F_1(G_i) \leqslant (2\alpha^2)^{i-1}\varepsilon F_1(D_1)F_1(G_1)$. It follows that $\sum_{i=1}^{t}\langle \mathbf{f}_i, \mathbf{g}_i \rangle$ can be estimated with error $\sum_{i=1}^{t}(2\alpha^2)^{i-1}\varepsilon F_1(D_1)F_1(G_1)$. For $\alpha < 1/\sqrt{2}$, this error is bounded by $O(\varepsilon F_1(D_1)F_1(G_1))$.

For $\langle \mathbf{f}_i, \mathbf{g}_j \rangle$ with $i < j$, we first convert $\mathcal{S}(2^{i-1}\varepsilon, D_i)$ to $\mathcal{S}(2^{j-1}\varepsilon, D_i)$, and do the estimation with $\mathcal{S}(2^{j-1}\varepsilon, G_i)$, which gives us an error of $2^{j-1}\varepsilon F_1(D_i)F_1(G_i)$. Therefore the error of estimating $\sum_{i<j}\langle \mathbf{f}_i, \mathbf{g}_j \rangle$ can be bounded by

$$\begin{aligned}
\sum_{i<j} 2^{j-1}\varepsilon ||\mathbf{f}_i||_1 ||\mathbf{g}_j||_1 &= \sum_{i=1}^{t-1} \varepsilon ||\mathbf{f}_i||_1 \sum_{j=i+1}^{t} 2^{j-1} ||\mathbf{g}_j||_1 \\
&\leqslant \sum_{i=1}^{t-1} 2^{i}\varepsilon ||\mathbf{f}_i||_1 \sum_{j=i+1}^{t} (2\alpha)^{j-i-1} ||\mathbf{g}_1||_1 \\
&= \sum_{i=1}^{t-1} 2^{i}\varepsilon ||\mathbf{f}_i||_1 ||\mathbf{g}_1||_1 \sum_{j=i+1}^{t} (2\alpha)^{j-i-1}.
\end{aligned}$$

For constant $\alpha < 1/2$, we have $\sum_{j=i+1}^{t}(2\alpha)^{j-i-1} = O(1)$, so the error of estimating $\sum_{i<j}\langle \mathbf{f}_i, \mathbf{g}_j \rangle$ is at most

$$\begin{aligned}
O\left(\sum_{i=1}^{t-1} 2^{i-1}\varepsilon ||\mathbf{f}_i||_1 ||\mathbf{g}_1||_1\right) &\leqslant O\left(\sum_{i=1}^{t-1} (2\alpha^2)^{i-1}\varepsilon ||\mathbf{f}_1||_1 ||\mathbf{g}_1||_1\right) \\
&= O(\varepsilon ||\mathbf{f}_1||_1 ||\mathbf{g}_1||_1).
\end{aligned}$$

We can similarly bound $\sum_{i<j}\langle \mathbf{g}_i, \mathbf{f}_j \rangle = O(\varepsilon ||\mathbf{f}_1||_1 ||\mathbf{g}_1||_1)$.

This proves that Count-Min sketch is α-exponentially decomposable for any constant $0 < \alpha < 1/2$. One technicality is that our data structures only support $1/2$-exponentially decomposable summaries as described in Section 5.3. This is caused by the use of a binary tree $\mathcal{T}$. To get around the problem, we replace the binary tree $\mathcal{T}$ with a ternary tree, so that subtree sizes decrease by a factor of 3 from a level to the one below. Now the left query path may have two nodes on each level in the canonical decomposition of the query range, and so does the right query path. This results in 4 series of sketches for representing $\mathbf{f}$ and $\mathbf{g}$. But this does not affect our analysis by more than a constant factor as argued earlier.

Remark. One technical subtlety is that, since we are now making $O(\log^2 N)$ estimations, in order to be able to add up the errors, we need all the estimations to succeed, i.e., stay within the claimed error bounds. To achieve a $1-\delta$ overall success probability, each individual estimation should succeed with probability $1 - \delta/\log^2 N$, by the union bound. Thus each Count-Min sketch $\mathcal{S}(c^{i-1}\varepsilon, D_i)$ we use should have size $O((1/c^{i-1}\varepsilon)\log(\frac{\log N}{\delta}))$.

5.4.4 The AMS sketch and wavelets

Given a multiset D in which the frequency of x is $f_D(x)$, the AMS sketch computes $O((1/\varepsilon^2)\log(1/\delta))$ counters $Y_i = \sum_x h_i(x)f_D(x)$, where each $h_i : [U] \to \{+1, -1\}$ is a uniform 4-wise independent hash function (Dobra and Rusu [61] show that some 3-wise independent hash functions also suffice). The AMS sketch is clearly decomposable. But since it provides an error guarantee depending on $F_2(D)$, it is not exponentially decomposable. Intuitively, the size of a data set could drop by a constant factor without reducing its F_2 significantly. More precisely, for two data sets D_1 and D_2 with $F_1(D_2) \leqslant \alpha F_1(D_1)$ for a constant $\alpha < 1$, $F_2(D_1) - F_2(D_2)$ may be $o(F_2(D_1))$. Thus the AMS sketch can only be used in the baseline solution of Section 5.2.

Gilbert et al. [34] have shown that an AMS sketch of an appropriate size also incorporates enough information from which we can build a wavelet representation of the underlying data set. Thus, the baseline data structure of Section 5.2 can also be used to return a wavelet representation for the data records in the query range.

5.5 Handling Updates

If the summary itself supports updates, our data structures also support updates. In particular, the MG summary [56], the GK summary for quantiles [37] support insertions, while the Count-Min sketch and the AMS sketch support both insertions and deletions. The corresponding summary data structures then also support insertions or both insertions and deletions. In this section we briefly describe how we handle updates for the two internal memory structures in Section 5.2 and 5.3. The techniques are quite standard [58], so we just sketch the high-level ideas.

The baseline structure. We first assume that the structure of the binary tree $\mathcal{T}$ remains unchanged during updates, then we show how to maintain its balance dynamically. We will show how to handle insertions; deletions can be handled similarly, provided that the summary itself supports deletions.

To do an insertion, we first search down the tree $\mathcal{T}$ using the new record's A_q attribute and locate the fat leaf v where the new record should reside. Then we insert it into v. This new insertion affects all the summaries attached to the $O(\log N)$ nodes on the path from the root of $\mathcal{T}$ to v. For each such node u, a summary on all the items stored below u is attached, so we need to insert the new record to the summary as well. Assuming the update cost for a single summary is $O(\mu)$, the total cost of this insertion is $O(\mu \log N)$.

We can maintain the structure of $\mathcal{T}$ using a *weight-balanced* tree and *partial rebuildings* [58]. For any node $u \in \mathcal{T}$, the *weight* of u is defined to be the number of records stored

below u. Then we restrict the weight of a node u at height i to vary on the order of $\Theta(2^i s_\varepsilon)$. The fanout of each node of $\mathcal{T}$ may not be 2 any more, but the weight constraint ensures that it is still a constant. After inserting a new record into a leaf v, the weight constraints at the ancestors of v might be violated. Then we find the highest node u where this happens, and simply rebuild the whole subtree rooted at the parent of u. Suppose the parent of u is at height i. We rebuild the subtree level by level. At level j, there are 2^{i-j} summaries we need to build, each on a data set of size $O(s_\varepsilon 2^j)$. So building each summary by simply inserting the records into an initially empty summary takes $O(\mu s_\varepsilon 2^j)$ time. This is $O(\mu s_\varepsilon 2^i)$ in total for level j. Summing over all i levels, the total cost of the rebuilding is $O(\mu s_\varepsilon 2^i \cdot i) = O(\mu s_\varepsilon 2^i \log N)$. After the rebuilding, the weight of u decreases by $\Theta(2^i s_\varepsilon)$, so the cost of the rebuilding can be charged to this weight decrease. Since every insertion increases the weights of $O(\log N)$ nodes by one, the cost of all the rebuildings converts to an $O(\mu s_\varepsilon 2^i \log N / 2^i s_\varepsilon \cdot \log N) = O(\mu \log^2 N)$ cost per insertion amortized.

Theorem 5.5 *If the summary can be updated in $O(\mu)$ time, the baseline internal memory structure can be updated in $O(\mu \log^2 N)$ time amortized.*

Remark. In the above rebuilding algorithm, we do not assume any properties of the summary. In fact, for all the summaries considered in this proposal, they do not have to be built from scratch for every level. Instead, the summaries at all levels can be constructed more efficiently in $O(\mu s_\varepsilon 2^i)$ time. This will reduce the amortized update cost to $O(\mu \log N)$.

The optimal internal memory structure for F_1 based summaries. The update procedure for the optimal internal memory structure of Theorem 5.2 is almost the same as the baseline solution, except that at each node, we now have $O(\log s_\varepsilon)$ summaries with exponentially decreasing sizes. This adds an $O(\log s_\varepsilon)$ factor to the cost of updating all affected summaries upon each insertion, as well as the partial rebuilding cost. Thus we have:

Theorem 5.6 *If the summary can be updated in $O(\mu)$ time, the optimal internal memory structure of Theorem 5.2 can be updated in $O(\mu \log^2 N \log s_\varepsilon)$ time amortized.*

Remark. The above theorem does not assume any special properties of the summary. Again for all the summaries considered in this proposal, the update time can be improved to $O(\mu \log N \log s_\varepsilon)$.

Chapter 6

Future Directions

6.1 External Hashing

In Chapter 3 we build a cache-oblivious hash table that achieves $1 + 1/2^{\Omega(B)}$ query cost. An interesting open question is that we do not yet know if $t_q = 1 + 1/2^{\Omega(B)}$ is optimal in the cache-aware model (or in the cache-oblivious model with the two more conditions). It is known that we can achieve $t_q = 1$ (namely, *perfect hashing*) with an internal memory of size $M = \Theta(N/B)$ [35, 49, 52]. On the other hand, external linear probing and blocked probing achieve $t_q = 1 + 1/2^{\Omega(B)}$ with only $M = \Theta(B)$. There seems to be a tradeoff between M and t_q but this tradeoff is yet to be understood.

6.2 Summary Queries

In Chapter 5, we presented some initial positive results on supporting summary queries natively in a database system, that many useful summaries can be extracted with almost the same cost as computing simple aggregates. There are many interesting directions to explore:

1. Our data structure for the F_2 based summaries does not have the optimal query cost. Can we improve it to optimal? In fact, we can partition the data in terms of F_2 so that the F_2 based summaries are also exponentially decomposable, but we meet some technical difficulties since the resulting tree $\mathcal{T}$ is not balanced.

2. We have only considered the case where there is only one query attribute. In general there could be more than one query attribute and the query range could be any spatial constraint. For example, one could ask the following queries:

   (Q3) Return a summary on the salaries of all employees aged between 20 and 30 with ranks below VP.

   (Q4) Return a summary on the household income distribution for the area within 50 miles from Washington, DC.

   In the most general and challenging case, one could consider any `SELECT-FROM-WHERE` aggregate SQL query and replace the aggregate operator with a `summary` operator.

3. Likewise, the summary could also involve more than one attribute. When the user is interested in the joint distribution of two or more attributes, or the spatial distribution of the query results, a multi-dimensional summary would be very useful. An example is

   (Q5) What is the geographical distribution of households with annual income below $50,000?

   Note how this query serves the complementing purpose of (Q4). To summarize multi-dimensional data, one could consider the multi-dimensional extensions of quantiles and wavelets, as well as geometric summaries such as ε-approximations and various clusterings. The former is useful for multiple relational attributes, while the latter is more suitable for summarizing geometric distributions as in (Q5).

Bibliography

[1] P. Afshani, G. S. Brodal, and N. Zeh. Ordered and unordered top-k range reporting in large data sets. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 2011.

[2] P. Afshani, C. Hamilton, and N. Zeh. Cache-oblivious range reporting with optimal queries requires superlinear space. In *Proc. Annual Symposium on Computational Geometry*, 2009.

[3] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In *Advances in Discrete and Computational Geometry*, pages 1–56. American Mathematical Society, 1999.

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

[5] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. *Journal of Computer and System Sciences*, 64(3):719–747, 2002.

[6] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[7] A. Arasu and G.S. Manku. Approximate counts and quantiles over sliding windows. In *Proc. ACM Symposium on Principles of Database Systems*, 2004.

[8] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

[9] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[10] L. Arge, M. Bender, E. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority-queue and graph algorithms. In *Proc. ACM Symposium on Theory of Computing*, pages 268–276, 2002.

[11] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. ACM Symposium on Theory of Computing*, pages 268–276. ACM, 2002.

[12] L. Arge, V. Samoladas, and K. Yi. Optimal external memory planar point enclosure. *Algorithmica*, 54(3):337–352, 2009.

[13] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[14] M. A. Bender, G. S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, and A. López-Ortiz. The cost of cache-oblivious searching. In *Proc. IEEE Symposium on Foundations of Computer Science*, 2003.

[15] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.

[16] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2007.

[17] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 546–554, 2003.

[18] G. S. Brodal, B. Gfeller, A. G. Jørgensen, and P. Sanders. Towards optimal range medians. *Theoretical Computer Science*, 2011.

[19] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. ACM Symposium on Theory of Computing*, 2003.

[20] G. S. Brodal and J. Katajainen. Worst-case external-memory priority queues. In *Proc. Scandinavian Workshop on Algorithms Theory*, pages 107–118. Springer-Verlag, 1998.

[21] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[22] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[23] E. Demaine. Cache-oblivious algorithms and data structures. In *EEF Summer School on Massive Datasets*. Springer Verlag, 2002.

[24] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: upper and lower bounds. *SIAM Journal on Computing*, 23:738–761, 1994.

[25] R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999.

[26] R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoretical Computer Science*, 220(2):345–362, 1999.

[27] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[28] F. W. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 719–725, 1990.

[29] M. L. Fredman, J. Komlos, and E. Szemeredi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[30] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[31] M. Garofalakis and A. Kumar. Wavelet synopses for general error metrics. *ACM Transactions on Database Systems*, 30(4):888–928, 2005.

[32] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2001.

[33] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proc. ACM Symposium on Theory of Computing*, 2002.

[34] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proc. International Conference on Very Large Data Bases*, 2001.

[35] G. H. Gonnet and P. Larson. External hashing with limited internal storage. *Journal of the ACM*, 35(1):161–184, 1988.

[36] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[37] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2001.

[38] S. Guha, C. Kim, and K. Shim. XWAVE: Optimal and approximate extended wavelets for streaming data. In *Proc. International Conference on Very Large Data Bases*, 2004.

[39] Y. HAN. Deterministic sorting in $O(n \log\log n)$ time and linear space. *Journal of Algorithms*, 50(1):96–105, 2004.

[40] Y. Han and M. Thorup. Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear

space. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 135–144. IEEE, 2002.

[41] B. He and Q. Luo. Cache-oblivious databases: Limitations and opportunities. *ACM Transactions on Database Systems*, 33(2), Article 8, 2008.

[42] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1997.

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

[44] M. S. Jensen and R. Pagh. Optimality in external memory hashing. *Algorithmica*, 52(3):403–411, 2008.

[45] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. *ACM Transactions on Database Systems*, 33(4), Article 23, 2008.

[46] A. G. Jørgensen and K. G. Larsen. Range selection and median: Tight cell probe lower bounds and adaptive data structures. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 2011.

[47] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[48] P. Larson. Dynamic hash tables. *Communications of the ACM*, 31(4):446–457, 1988.

[49] P. Larson. Linear hashing with separators—a dynamic hashing scheme achieving one-access retrieval. *ACM Transactions on Database Systems*, 13(3):366–388, 1988.

[50] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang. Selecting stars: The k most representative skyline operator. In *Proc. IEEE International Conference on Data Engineering*, 2007.

[51] W. Litwin. Linear hashing: a new tool for file and table addressing. In *Proc. International Conference on Very Large Data Bases*, pages 212–223, 1980.

[52] H. G. Mairson. The effect of table expansion on the program complexity of perfect hash functions. *BIT*, 32(3):430–440, 1992.

[53] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1998.

[54] Y. Matias, J. S. Vitter, and M. Wang. Dynamic maintenance of wavelet-based histograms. In *Proc. International Conference on Very Large Data Bases*, 2000.

[55] A. Metwally, D. Agrawal, and A. E. Abbadi. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Transactions on Database Systems*, 31(3):1095–1133, 2006.

[56] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2:143–152, 1982.

[57] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[58] M. H. Overmars. *The Design of Dynamic Data Structures*. Springer-Verlag, LNCS 156, 1983.

[59] A. Pagh, R. Pagh, and M. Ružić. Linear probing with constant independence. In *Proc. ACM Symposium on Theory of Computing*, 2007.

[60] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51:122–144, 2004.

[61] F. Rusu and A. Dobra. Pseudo-random number generation for sketch-based estimations. *ACM Transactions on Database Systems*, 32(2), Article 11, 2007.

[62] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[63] J. P. Schmidt, A. Siegel, and A. Srinivasan. Chernoff–Hoeffding Bounds for Applications with Limited Independence. *SIAM Journal on Discrete Mathematics*, 8:223, 1995.

[64] Y. Tao, L. Ding, X. Lin, and J. Pei. Distance-based representative skyline. In *Proc. IEEE International Conference on Data Engineering*, 2009.

[65] G. Tenenbaum. *Introduction to analytic and probabilistic number theory*. Cambridge Univ Press, 1995.

[66] M. Thorup. Equivalence between priority queues and sorting. *Journal of the ACM*, 54(6):28, 2007.

[67] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16:264–280, 1971.

[68] E. Verbin and Q. Zhang. The limits of buffering: A tight lower bound for dynamic membership in the external memory model. In *Proc. ACM Symposium on Theory of Computing*, 2010.

[69] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[70] J. S. Vitter. *Algorithms and Data Structures for External Memory*. Now Publishers, 2008.

[71] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proc. SIGMOD International Conference on Management of Data*, 1999.

[72] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[73] M. N. Wegman and J. L. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981.

[74] A. C. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proc. IEEE Symposium on Foundations of Computer Science*, 1977.

[75] K. Yi. Dynamic indexability and lower bounds for dynamic one-dimensional range query indexes. In *Proc. ACM Symposium on Principles of Database Systems*, 2009.