THE RECENTY PROBLEM AND ITS APPLICATIONS

SUBMITTED TO THE SCHOOL OF COMPUTING AND INFORMATION SYSTEMS
FOR A DOCTOR OF PHILOSOPHY (ENGINEERING)

WILLIAM HOLLAND (359894)
SUPERVISORS:
ANTHONY WIRTH, OLGA OHRIMENKO, JUSTIN ZOBEL

The University of Melbourne

JULY 2022
Abstract

Our thesis is centered on the question of how to calculate the recency of items over a stream of data. Informally, the recency query asks, ‘when was the last time I saw item \( x \)?’. This measure is a key component in the identification of hot data for workloads that exhibit high temporal localities. The objective is pertinent in the fields of data caching and flash storage, applications where available memory is limited. In large memory, existing structures, such as hash tables, can support a recency query by augmenting item occurrences with timestamps. Therefore, we are interested in small memory solutions for resource constrained environments.

The thesis is divided into two halves. In the first half, we exhibit two novel data structures that support recency queries in small memory. In the second, we expand the scope of the recency query and demonstrate a meaningful application in the field of oblivious storage.

Our first result, Historical Membership, builds on sliding-window dictionaries, which provide dynamic membership queries over a window of the most recent occurrences. By combining sliding-window dictionaries in a hierarchical structure, and with careful design of the underlying hash tables, Historical Membership supports recency queries with bounded relative error on top of a succinct representation of the window. The second result, the Princess List, supports stack distance queries in small memory. The stack distance is a variation of the recency query. The Princess List is the first succinct representation of a list that supports updates and index and search queries in optimal time.

To consolidate our work, we provide an application of Historical Membership in the field of oblivious storage. An oblivious RAM is a remote storage protocol that provides a client with access pattern privacy. We construct a new ORAM scheme, Rank ORAM, that leverages the support of Historical Membership to achieve better bandwidth and latency performance against existing approaches. In addition, we present a new solution to oblivious permutation, which is a key primitive in many ORAM schemes. The proven asymptotic behaviour of the algorithm, WaksmanOP, and our experimental results, show that under several realistic scenarios, compared to baseline solutions, WaksmanOP is the best trade-off.
Declaration of Authorship

This is to certify that:

• The thesis comprises only my original work towards the PhD,
• Due acknowledgement has been made in the text to all other material used,
• The thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Signed: William Holland

Date: 04/08/2022
Preface

This thesis research has been carried out in the School of Computing and Information Systems at The University of Melbourne. This is a thesis with publication and the main contributions in Chapters 3 - 6 are presented in manuscripts that are either published or under review. The manuscripts are summarized as follows.

- Chapter 3 presents a sole-authored manuscript titled “Succinct Indexing in Optimal Time” under review at the 33rd International Symposium on Algorithms and Computation (ISAAC 2022)

- Chapter 4 presents a manuscript, co-authored with Anthony Wirth and Justin Zobel, titled “Recency Queries with a Succinct Representation” published at The 31st International Symposium on Algorithms and Computation (ISAAC 2020)

- Chapter 5 presents a manuscript, co-authored with Olga Ohrimenko and Anthony Wirth, titled “Recency Queries with a Succinct Representation” under review at the 49th International Conference on Very Large Data Bases (VLDB 2023).

- Chapter 6 presents a manuscript, co-authored with Olga Ohrimenko and Anthony Wirth, titled “Efficient Oblivious Permutation via the Waksman Network” published at the ACM Asia Conference on Computer and Communications Security (AsiaCCS 2022).
Acknowledgements

This document would not be possible without the help and support of a number of people. First and foremost, I would like to thank my incredible supervision team, starting with my principal supervisor, Tony. I first met Tony in 2015 when he was lecturing a class I attended on theoretical computer science. I knew straight away I wanted to work with him on my Masters thesis. We have since worked together on a Masters thesis, a PhD thesis, an ARC linkage grant and teaching 3 different theoretical computer science subjects. More than anyone, Tony has helped me grow professionally as both a researcher and a teacher. Always full of energy and good humour, I am going to miss working with him.

I am grateful to my co-supervisor, Justin, who has been an amazing research mentor to me over the past three and half years. Justin wrote the book on computer science research and has helped elevate my writing and the communication of my ideas. Without him this thesis would be much harder to understand.

Last, but not least, I am thankful to my co-supervisor, Olya, who joined the team just over a year into my candidature. Olya introduced me to new problems in the field of data privacy. These problems allowed the scope of my thesis to grow and take shape. This research domain was new to me and she was always there to help me grasp and understand all the challenging cybersecurity concepts. I am inspired by Olya’s work and I am interested in continuing research in this field, An outcome I didn’t think would occur when I started my candidature.

I would not have made it to the finish line without the support of my friends and family. I am forever thankful to my mum, who calls me everyday and believes in me more than anyone I know. To my twin sister and best friend, Jess, who is always there for me and makes me laugh not matter how I am feeling. To my brother, Tim, for dropping off endless baked goods. To Lil, the funniest person in the world, who brought so much joy and love into my life over the final 8 months of my candidature. To Harrison and our two person skate-team. Finally, to all the housemates I have had throughout my candidature: Betty, Char, Jetaime, Hazel, Ruby, Sas and Steph. Particularly throughout the pandemic, you all provided an amazing home for me to return to after a day’s work.

To my beautiful dog, Rosie, who is so sweet and kind and always sits by my side. Lastly, to the world’s greatest dog, Mowgli, who has been a great support for both Rosie and myself.
Dedicated to Rosie
Contents

1 Introduction 1
  1.1 Application: Oblivious RAM 2
  1.2 The Data Stream 2
  1.3 Recency 3
  1.4 List update problem 4
  1.5 The Sliding Membership problem 6
  1.6 Oblivious Storage and Computation 8
  1.7 Contributions 10

2 Literature Review 12
  2.1 Preliminaries 12
  2.2 Prior solutions 16
  2.3 List indexing 23
  2.4 Sliding Membership 27
  2.5 Oblivious RAM 31
  2.6 Oblivious Permutation 40

3 Succinct List Indexing 44

4 Succinct Recency 63

5 Practical Hierarchical ORAM 79

6 Trade-offs in Oblivious Permutation 94

7 Conclusion and Future Directions 109

References 115
# List of Figures

2.1 Constant time rank queries .................................................. 14  
2.2 The Bloom Filter ................................................................. 15  
2.3 Two tier LRU cache ............................................................... 16  
2.4 Sliding Bloom Filter .............................................................. 19  
2.5 Markers Algorithm ................................................................. 21  
2.6 Bennett-Kruskal stack distance calculation .............................. 23  
2.7 Binary tree list indexing ......................................................... 24  
2.8 Data streaming models ........................................................... 28  
2.9 Circular Array ....................................................................... 30  
2.10 Hierarchical ORAM .............................................................. 34
List of Tables

2.1 Comparison of time-recency data structures ........................................ 31
2.2 Comparison of Hierarchical ORAMs .................................................. 36
2.3 Comparison of practical ORAMs ......................................................... 38
2.4 Comparison of Oblivious Permutation algorithms ................................. 40

7.1 Summary of Contributions .............................................................. 110
Chapter 1

Introduction

A data stream is a temporally indexed sequence of elements. It has three key properties: it is ordered, fast and large. These properties instruct how we capture the stream, and the information it expresses, for various applications. Typically, we cannot afford to store the whole sequence, indeed, it is often unbounded, and are thereby restricted to storing a summary. This summary is generally limited to supporting approximations to the queries we seek to evaluate. Furthermore, the ordered nature of the stream demands that these summaries adapt to change. This dynamism is significant as many applications are interested in the recent history of the stream. This is evident both in theory, which has formalized the notion that a recent item is important [80, 81, 84, 87, 137], and in practice, for example, in time-sensitive applications such as in-network caching [94] and the detection of duplicates [104].

There exist well-developed techniques to answer time-indifferent queries such as the number of distinct items in the stream [88] or the frequency of a particular item [44]. To allow for up-to-date inferences on evolving streams, these techniques can be extended to evaluate time-decayed aggregate statistics [40, 45]. For instance, Zhang et al. [153] deploy existing distinct item counters [57] to support a time-decayed variant that answers: how many distinct items occurred during the most recent W updates? A time-decay framework acknowledges the importance of the order of the data. However, these time-decayed statistics do not constitute fine-grained measures of temporality that express when particular items occurred. Examples of these measures include item recency – informally, recency asks: when was the last time we saw item x? – and the co-occurrence of items. Existing tools from the data streaming literature cannot support compact and accurate solutions to these temporal queries. Thus, to address this shortcoming, the question of how to measure recency both efficiently and accurately constitutes one of the core problems of this thesis.

Assuming a non-optimal or unrestrained memory allocation, the literature provides time-efficient exact solutions to the problem; for example, a hash table that stores timestamps with item identifiers [96]. Therefore, our interest is in low memory solutions that are required in the context of memory-bounded computation. Indeed, it is this memory constraint that makes the problem difficult to solve, as we may have to resort to approximation to satisfy the memory bound. Thus, a central challenge entails formalizing this memory-accuracy trade-off and providing a guarantee over the inaccuracy of the query.

The recency measure is a key component in the identification of hot data for workloads
that exhibit high temporal localities [120, 121]. This objective is pertinent in the fields of
data caching [3, 74] and flash memory [71, 78, 97]. Both prior and subsequent work on hot
data identification has acknowledged the need for low memory solutions to the recency query
[93, 120]. However, the query has not been subject to any meaningful theoretical investigation;
prior work is heuristically driven (see Section 2.2). Thus, we look to bridge this gap between
theory and practice. Our work contributes important theoretical insights – such as lower bounds
on memory requirements and trade-offs between execution time and query accuracy – on the
theoretical boundaries of the query (Sections 3 and 4). In addition, our work expands the prac-
tical scope of the recency measure. To complement our theoretical insights, and constituting
the second core contribution of the thesis, we demonstrate how small memory solutions to re-
cency can enable performance improvements for a class of oblivious storage schemes (Section
5). The latter is a cryptographic primitive that provides access pattern privacy on untrusted
storage. To ground the more abstract and theoretical components of this exposition, we begin
with an elaboration of this oblivious storage application.

1.1 Application: Oblivious RAM

Oblivious RAM (ORAM) [64] is a cryptographic primitive that provides access pattern privacy
in remote storage settings. At a high level, an ORAM transforms a sequence of virtual accesses
(to remote storage) into a sequence of physical accesses that is independent of the input. The
transformation eliminates information leakage in the access trace. To provide the necessary
obfuscation, an ORAM generates additional accesses. These take the form of either dummy
accesses or periodic shuffling of the server’s contents.

One class of ORAM solutions, Hierarchical ORAM [13, 32, 64, 69, 122], distributes the
server’s contents across a hierarchy of hash tables. The most recently accessed blocks are
placed in the lower levels of the hierarchy and are periodically shuffled into the higher levels if
they are not accessed again. Thus, the hierarchy is established on block recency. As a result, in
the words of Sion and Williams [146], the execution of “a request for an item depends on how
recently an item was accessed”. Further, the execution of a request can be made more efficient
with prior knowledge of the block recencies. However, maintaining this metadata in private
memory requires storage linear in the size of the outsourced database. This has prohibited the
use of recency metadata for optimizing Hierarchical ORAM schemes.

Alternatively, if recency queries can be supported in low memory, and with sufficient accu-
rency, these optimizations become feasible. We elaborate on this problem in more detail below.
Before formalizing the research questions that organize this thesis, we require some theoretical
context. Consequently, our discussion moves to the concept of the data stream [46], a flagship
of low memory computing.

1.2 The Data Stream

The data stream promotes a way of thinking about algorithms and data structures that is not
only of practical significance, given the emergence of big data, but, in addition, permits rich
theoretical inquiry. A streaming problem emerges when we want to ask a question about a
stream in some constrained computing environment. This question often takes the form of a query:

- How many times does item \( x \) appear?
- How many distinct items are there?
- Which was the most frequent item?

The preceding two decades have instituted well-developed techniques for solving the above queries in low memory [16, 36, 44, 100]. If the query is sufficiently complex, supporting an exact solution can require a prohibitive amount of memory. For example, to solve item frequency exactly, we would require a counter for every possible item. Thus, when subject to a restrictive memory bound, we store a structure that provides a summary or sketch of the observed data. The summary then enables approximate queries. Accordingly, the summary has three key attributes: compactness (memory efficiency); speed, that is, the time cost for queries and updates (time efficiency); and accuracy. The streaming attributes maintain a precarious relationship among themselves and often cannot be satisfied simultaneously. That is, we can make the summary smaller, but at the expense of time efficiency or accuracy. Therefore, the attributes maintain an economic relationship, where we purchase one form of efficiency at the expense of another. At its core, a streaming problem articulates a market-place of purchasing and trading various efficiencies.

The above list of queries represent aggregated statistics – figures that are indifferent to the order in which data occurs. In contrast, the recency measure constitutes a temporal query about data streams. As we are handling unanswered questions, it is natural to begin by asking: what type of data structure supports a temporal query? What kind of principles does it follow? In part, the purpose of this introduction is to respond to these questions. Throughout, we will refer to a data structure that supports meaningful temporal queries as historical. Before proceeding with what such a data structure might look like, we require a more concrete notion of the term “recency”.

1.3 Recency

We distinguish between two types of recency: time-recency and item-recency. Given a stream of tokens \( S = \langle s_1, \ldots, s_t \rangle \) taken from the universe \( U \), the time-recency of an item measures the time elapsed since it last appeared in the stream.

**Definition 1.** Formally, the time-recency \( r_t(x) \) of the item \( x \in U \) (at time \( t \)) is defined as:

\[
r_t(x) = \begin{cases} 
  t - \max \{i \mid s_i = x\} & \text{if } x \in S, \\
  n/a, & \text{if } x \notin S, 
\end{cases}
\]

(1.1)

following the convention that a item absent from the stream has a null recency value.

In other words, the time-recency equals the number of tokens that have arrived since the previous appearance of item \( x \). For example, the stream of tokens \( \langle 1, 2, 3, 5, 5, 4, 3, 10 \rangle \) admits
the following recency statistics: \( r_8(3) = 1 \) and \( r_8(2) = 6 \). In the Hierarchical ORAM setting (see Section 1.1), the sequence of virtual accesses constitutes the stream of tokens and the recency value determines how a request is executed. This statistic recognizes the repetitions of elements. In contrast, item-recency is concerned with the order of the elements themselves.

**Definition 2.** Formally, the item-recency \( q_x \) of the token \( x \in U \) is defined as:

\[
q_t(x) = \begin{cases} 
|\{y \mid s_i = y, i > r_t(x)\}| & \text{if } x \in S, \\
-1 & \text{if } x \not\in S,
\end{cases}
\]  

(1.2)

In other words, \( q_t(x) \) equals the number of distinct items in the stream that occurred after the previous appearance of \( x \).

Following the same example as above, we observe the following item-recency statistics: \( q_8(3) = 1 \) and \( q_8(2) = 4 \).

In our research, we introduce a suite of data structures that answer these queries either accurately, quickly, in small memory, or in some combination of these attributes. That is, we are interested in trade-offs. These data structures, in the sense that they understand that items occur in some order and thereby belong to some epoch of the sequence, are historical. This is in contrast to data structures supporting the standard frequency query, where the time of the occurrences are irrelevant. Our solutions borrow ideas from both the sliding-window model of computation [48] and the literature of self-organizing data structures [136]. In the former, the scope of a data streaming query is restricted exclusively to the most recent \( W \) items of the data stream – otherwise known as the window. It requires the assumption that old data is less relevant – which is the case in many applications – and, in this way, acknowledges the significance of the order of the data. Self-organizing data structures are some of the earliest examples of search structures that respect recency. We expand on this precedent, in the context of the list update problem, in the following subsection.

### 1.4 List update problem

In the domain of self-organizing data structures, the objective is to minimize the cost of a sequence of operations (to some data structure) by applying a rule that determines whether we modify the structure after an operation. The update rules and modifications are designed to increase execution efficiency. The analysis is amortized as it concerns the cost across the whole sequence. The notion of recency, and the idea that a recently queried item should occasion a low query time cost, influence many of the self-organization rules. In this sense, it is understood that the distribution of the input can be leveraged to improve efficiency.

The earliest problem in self organizing data structures is the list update problem [38]. A list is a linear collection of items that implements the dictionary abstraction. An access to the list scans the list from the head until the requested item is found. After the item is located, a list algorithm is permitted to rearrange the list according to some rule. Similarly, if the item is not found, it is placed in the list according to the specified rule. In the list update problem, a list algorithm is designed to minimize the cost of a sequence of access requests.
A well-established list update algorithm, the move-to-front list [136], offers an exact solution to item-recency. The data structure is noted for achieving the asymptotically optimal cost, for deterministic online list algorithms, in a number of cost models of interest; specifically, competitive analysis [83] and bijective analysis [9]. The update rule for the move-to-front list is the following: after an item is accessed, move it to the front of the list. Thus, the algorithm “abstracts a temporal ordering” [132], as the \( i \)th most recently accessed item sits at the \( i \)th location in the list. To evaluate a recency query, we scan the list from the head until the queried item is found, incrementing a counter at each step. The key limitation with this approach is that worst-case queries require a full scan of the list and complete in linear time. Further, for a list of length \( m \), it is not difficult to design access sequences that require \( \Omega(m) \) time on average.

To overcome this drawback, we could apply the move-to-front rule to an indexed list. The latter is a list abstraction that (efficiently) supports the additional index and position, which enable faster access to the elements of the list. The operations are summarized in the following problem statement.

**Problem 1 (List Indexing).** Construct a representation of a list \( L \) that supports the following operations:

- \( \text{insert}(L, x, y) \): insert element \( y \) at the index succeeding element \( x \).
- \( \text{delete}(L, x) \): delete element \( x \) from the list.
- \( \text{index}(L, i) \): return the element at index \( i \) in the list.
- \( \text{position}(L, x) \): return the index of element \( x \) in the list.

Thus, under a move-to-front arrangement, a position query is equal to item-recency. For a list encoding an arbitrarily ordered set \( L \subseteq [n] \) of size \( m \), Dietz provides a solution that admits updates and queries, optimally, in \( \Omega((\log m)/(\log \log m)) \) time. As is noted by Andersson [8], the solution requires item pointers be provided to the operations. Thus, the data structure of Dietz appears in three components: a hash-table to retrieve item pointers; the underlying linked-list; and the structure to support (fast) indexing. This configuration requires \( O(m \log n) \) bits of space with a constant factor around 12.

In contrast, the information theoretic lower bound for ordered sets is \( m(\log n - O(1)) \) bits. A low-memory solution can be constructed by positioning the items consecutively in an array that has \( \log n \)-bit cells. The per-item cost of the encoding is \( O(1) \) bits above the information-theoretic lower bound. However, updates to the list and position queries are slow, requiring \( O(m) \) time. This observation leads to an interesting question: can a list be encoded at close to \( \log n \) bits per item and support dynamic indexing operations efficiently?

A data structure, supporting a particular query (or set of queries), is succinct if it accommodates a memory commitment close to the information-theoretic lower bound and admits the prescribed query operation(s) efficiently. Formally, if a minimum of \( \mathcal{B} \) bits are required, in the information-theoretic sense, to store the data, a succinct data structure occupies \( \mathcal{B} + o(\mathcal{B}) \) bits.

\[ ^{1} \text{These times are amortized for updates.} \]
of memory [86]. Despite the affluent and diverse state of the field of succinct data structures [63, 73, 110, 128], a succinct representation of an indexed list does not exist and we pursue the problem of finding such a representation.

Research Question 1. Is there a succinct representation of an indexed list that admits queries and updates in optimal time?

In response to this question, we first present a simple data structure, the Princess List (PrincessList), that significantly improves the update and query efficiency of a linked-list at a small space overhead (Chapter 3). We then detail an optimized implementation of the Princess List (PrincessList+) that occupies $m(\log n + o(\log m))$ bits of space, with high probability, and admits optimal query and update times. Thus, PrincessList+ achieves a succinct representation and matches the prior state-of-the-art in run time. The structure pays, per-item, a sublogarithmic number of bits above the information-theoretic lower bound for ordered sets.

Within the broader field of succinct data structures, the list indexing problem has proximity to the well known rank-select problem [66, 67]. The latter involves indexing a dynamic string $S$ (which may have repetitions) to support the access, rank and select operations [115]. The select operation is equivalent to the position query and an access is exactly the index query. The two problems are closely related and, indeed, the latter inherits the time lower bounds of the former. The main difference between the two problems is the assumption about the universe size. For the succinct string problem it is reasonable to focus on texts drawn from a small universe. However, these solutions become intractable for problems on larger universes and, particularly, where repetitions do not occur. A succinct indexed list remedies this restriction.

We now shift our attention to time-recency and the origins of our solution in the sliding membership problem.

1.5 The Sliding Membership problem

Many queries in the data streaming literature, with respect to their histories, are cumulative, in the sense that they aggregate everything that has occurred into one summary. The order in which the data occurred is not pertinent. Adhering to our terminology, these data structures can be described as ahistorical.

Often, the tactic within the data stream community, towards developing time-conscious algorithms, has been to constrain the streaming model to acknowledge the history of the stream. For example, authors may either restrict queries to sliding-windows [48] or time-decay [43] their data structures. These models allow statistics that are time-indifferent, such as the median or sum of a sequence of numbers [135] to be embedded, via the model, into a time-conscious algorithm. In this arrangement, the queries are now indexed to epochs of the stream.

Despite the variety of work classified under the above models, there has been less interest in exploring the scope of more fine-grained temporal statistics, such as those pertaining to recency or, say, the co-occurrence of items. Nonetheless, these models supply a wealth of meaningful templates. The baseline for a time-recency query is fairly simple and intuitive: retrieve a time marker from the past. Data structures that meet this requirement can be found in the sliding membership literature [15, 96, 111, 145]. A sliding membership query asks whether an item is
a member of the last $W$ elements of the stream. The standard solution involves augmenting a hash table with timestamps or some identifier of a point in history. However, this strategy has not been accomplished in both small space and with sufficient accuracy. Thus, we look at the problem of balancing these demands.

The commonly cited naïve solution for sliding membership, with slack parameter $D > 0$, entails dividing the window into blocks of width $D$. Each block is stored in a static dictionary. To evaluate item membership on the sliding-window it suffices to query each block, in turn, and return the logical disjunction of the results. Similarly, the time-recency of an item $x$, given that the item is a member of the window, can be approximated by returning the recency of the (youngest) block it belongs to. This approach returns approximations with absolute error at most $D$. When viewed from the perspective of relative error, estimates are less accurate for items with low recency. Accurate estimates may be required throughout the window and are arguably more valuable for more recent items. This observation leads to the following formalization of the recency problem.

**Problem 2 (Time Recency Problem).** Let $S_W(t) = (s_{t-W+1},\ldots,s_t)$. Given $W,D \in \mathbb{Z}^+$ and $\varepsilon \in (0,1)$, and the sequence $S(t) = (s_1,s_2,\ldots,s_t)$ drawn from universe $[n]$, when presented with some item $x \in [n]$, return an estimate $\hat{r}$ for $r_t(x)$ where

$$\hat{r} \in \begin{cases} 
(1 + \varepsilon)r_t(x), & \text{if } x \in S_W(t), \\
-1, & \text{if } x \notin S_{W+D}(t), \\
(1 + \varepsilon)r_t(x) \cup \{-1\}, & \text{otherwise.}
\end{cases}$$

An exact time-recency query is supported trivially by storing each item with its most recent time-stamp. This arrangement requires $O(W(\log n + \log t))$ bits of memory. In many settings, this memory allocation is too large and a succinct representation is preferred [111]. Consequently, balancing this triangle of accuracy, memory and running time suggests the following research questions.

**Research Question 2.** What is the memory lower bound for the Time Recency Problem? Can a summary of the stream meet this lower bound and support efficient query and update operations?

In response to the question, we introduce a data structure named Historical Membership (HistoricalMembership) that achieves tight memory allocation and bounded relative error in return for logarithmic update and query times (Chapter 4). Our solution builds on existing approaches, particularly the tactic of dividing the window into blocks of items of equivalent age. However, to achieve both relative $(1 \pm \varepsilon)$ accuracy in time-recency and also space efficiency, the structure is hierarchical, comprising levels of geometrically increasing size.

To solidify the utility of HistoricalMembership, we now explore an application where it can be leveraged to improve the state-of-the-art performance of oblivious RAMs.
1.6 Oblivious Storage and Computation

The notion of oblivious storage was introduced above in Section 1.1. To recapitulate, an ORAM is a cryptographic primitive that provides access pattern privacy to untrusted storage. Access pattern leakage has been demonstrated in a number of domains. For example, query recovery attacks in searchable encryptions can be realized through the search pattern [31, 85].

In the typical setting, the client stores a collection of \( n \) blocks of size \( B \) bits. An ORAM transforms an input sequence of logical accesses to the data blocks into a sequence of physical accesses that is independent of the input sequence. An adversary observing the (physical) access pattern produced by an ORAM should not be able to distinguish between the executions on any two distinct inputs of the same length. To provide sufficient obfuscation, dummy blocks are retrieved and portions of the blocks at the server are periodically shuffled, creating an overhead. The key metric in the literature is this bandwidth overhead to the server. The location of a block at the server, which is dictated by the periodic shuffling of portions of blocks, depends on the block’s access recency.

Arguably the second most important metric, and certainly the most contested, is the amount of memory available to the client. Earlier solutions [64] and most theoretical works [13, 122, 140] restrict the client to \( O(B) \) bits. However, works that focus on practical deployments have relaxed this restriction and, by allowing a large client, have procured improved bandwidth and latency performance [58, 130]. The latter is accomplished by allowing the client to store information about the physical location of each block at the server – known as the position map. The position map requires \( \Omega(n \log n) \) bits to store and is justified in settings where the block size is large, that is, \( \omega(\log n) \) bits. However, for applications with small block size, the position map dominates the client memory commitment.

Under the observation that, for certain classes of ORAM solutions [124, 140], the recency of the block determines its value in the position map, we consider whether smaller \( O(n) \)-bit clients are possible with a compressed recency data structure. The consideration of a \( O(n) \)-bit client is novel, but nonetheless compatible with the literature. Under this constraint, we are able to revisit some existing problems in oblivious storage and computation and explore new solutions and trade-offs.

Application of HistoricalMembership

A canonical class of solutions, known as Hierarchical ORAMs [32, 64, 70, 122, 13], are yet to leverage the advantage of larger client memory. In the hierarchical scheme, all virtual addresses are stored, with corresponding blocks, in a sequence of levels of exponentially increasing size. The hierarchy is configured such that recently accessed blocks are stored in the lower levels and are gradually moved into the higher levels if they are not accessed again.

As the levels are reorganized on a deterministic schedule, if the client knew how recently a block was accessed, it would know which level the block belonged to. With some additional information, such as the rank of the block within the level, the performance of the hierarchical ORAM protocol can be significantly improved. Notably, HistoricalMembership can support the required functionalities and boasts the additional property of having an optimal memory allocation. Thus, we arrive at the following research question.
Research Question 3. To store a data set of \( n \) blocks, the class of practical ORAMs achieve optimal online bandwidth and require \( \Omega(n \log n) \) bits to store a position map in client memory. With HistoricalMembership, is it possible to reduce this client memory overhead to \( \mathcal{O}(n) \) bits without degrading bandwidth performance?

Our version of Hierarchical ORAM is named Rank ORAM (RankORAM). The significance here is that HistoricalMembership is asymptotically smaller than all prior solutions (position maps) on non-hierarchical ORAMs. It therefore opens the door for application on low-memory secure processors, such as the Intel SGX [7], a setting that has been reserved for a class of ORAMs that do not store the position maps at the client [130].

Oblivious Permutation

To utilize the compressed position map, our resultant ORAM protocol requires careful modifications to the underlying oblivious permutation (OP) algorithm. Oblivious permutation allows a client to shuffle the contents of the server’s memory according to a given permutation \( \pi \), without revealing \( \pi \) to the server through its memory accesses. We explore various approaches to OP under the \( \mathcal{O}(n) \) bit client constraint. The relaxation of the problem, under this constraint, is novel with respect to the literature and, consequently, led to some new insights on the trade-off between bandwidth and client memory.

Existing solutions fall on two sides of this trade-off. On the one hand solutions based on sorting networks [4, 20, 68, 112] use \( \mathcal{O}(B) \) bits of client memory, enough to read and sort a constant number of blocks and, for the theoretical state-of-the-art, complete in \( \mathcal{O}(n \log n) \) time\(^2\) [4, 68]. On the other hand, solutions based on shuffling [117, 123] rely on \( \mathcal{O}(\sqrt{n} \cdot B) \) bits of client memory in order to read larger batches of blocks, but complete in linear time. However, the literature is yet to explore another dimension in this trade-off. Neither of these versions of the problem allow for \( \mathcal{O}(n) \) bit clients, which is the allocation we grant RankORAM. We are therefore interested in the existence of practical OP algorithms that can operate within this allocation. This is further motivated by observations within the ORAM literature that, in many applications, the \( \mathcal{O}(\sqrt{n} \cdot B) \) bound for oblivious shuffle is much larger than the \( \Omega(n \log n) \) bits set aside for the position map [37, 124, 130]. This opens the door for deterministic permutation networks, in particular, the Waksman network [143], which are much smaller and practical than the state-of-the-art sorting networks, but are generally avoided as standard solutions require \( \Omega(n \log n) \) bits to store the network configuration (obliviously) in client memory. Given our stated constraint on client memory, we arrive at a new research question.

Research Question 4. Follow existing implementations, the Waksman network requires \( \mathcal{O}(n \log n) \) bits of client memory. For Oblivious Permutation, can the client memory overhead be reduced? Further, does this lead to performance improvements in practice?

\(^2\)Note that large hidden constants prevent their use in practice (e.g., 19600 for Zig-Zag sort [68]). Typically, the (asymptotically) non-optimal Bitonic network [112], which requires exactly \( n \log^2 n \) time, is used for small memory settings.
In response, we construct a non-trivial routing algorithm, WaksmanOP, for an oblivious execution of the Waksman network, which admits client memory allocation asymptotically smaller than prior (or naive) approaches.

1.7 Contributions

This thesis has produced the following algorithms and data structures with corresponding contributions.

- PrincessList (Chapter 3 and Research Question 1); the first succinct indexed list that achieves optimal run time. The properties are summarised in the following theorems.

**Theorem 1.** An ordered set of m items, drawn from the universe \([n]\), where \(n = \text{poly}(m)\), can be stored in \(m(\log n + o(\log m))\) bits with probability \(1 - 1/m^{O(1)}\) and support index and position in \(O(\log m / \log \log m)\) time and insert and delete in \(O(\log m / \log \log m)\) amortized time.

**Theorem 2.** An ordered set of m items, drawn from the universe \([n]\), can be stored in \(m(\log n + o(\log m))\) bits and support index in \(O(\log m / \log \log m)\) time, position in \(O(\log m)\) time with probability \(1 - 1/m^{O(1)}\) and any sequence of \(O(m)\) updates (insert or delete) takes \(O(m \log m)\) time with probability \(1 - 1/m^{O(1)}\).

- HistoricalMembership (Chapter 4 and Research Question 2); a solution to time-recency that achieves optimal memory allocation and fast update and query times. The properties are summarised in the following theorem.

**Theorem 3.** On a sequence of \(S = \langle s_1, s_2, \ldots \rangle\), where \(s_i \in [n]\), for parameters \(W\) and \(\varepsilon > 0\), at each timestamp \(t \geq 1\), HistoricalMembership solves the Recency problem in

\[
(1 + o(1))(1 + \varepsilon)(B + W \log(\varepsilon^{-1}))
\]

bits of memory, admitting query and update times of item \(x\) in \(O(\log(\varepsilon \cdot r(x,t)))\). This bound is with high probability worst case for queries and expected amortized for updates. Here, \(B\) is the information-theoretic lower bound for storing a subset of size \(W\) from the universe \([n]\).

- RankORAM (Chapter 5 and Research Question 3); an ORAM protocol that achieves state-of-the-art bandwidth performance with client memory that is asymptotically smaller that prior low bandwidth approaches. Our experiments demonstrate the feasibility of our ideas in practice.

**Theorem 4.** RankORAM is an oblivious RAM that stores a database of \(n\) blocks of \(B\) bits, requires \(O(n + \sqrt{n} \cdot B)\) bits of private memory, performs accesses in a single round-trip and observes an amortized bandwidth overhead of \(4 \log n\) blocks.
• WaksmanOP (Chapter 6 and Research Question 4); a novel algorithm for routing on the Waksman network. With reference to TABLE 2.4, the algorithm provides a reasonable trade-off between client storage and I/O for large block sizes, \( B \), compared to existing methods. We implement and test WaksmanOP against recognized baselines. Our experiments show that WaksmanOP would be the algorithm of choice for a set of parameters that reflect many realistic scenarios. The properties of WaksmanOP are summarized in the following theorem.

**Theorem 5.** The WaksmanOP is an oblivious array permutation algorithm for an array of \( n \) \( B \)-sized elements. The algorithm requires \( 2n + o(n) + 2B \) bits of client storage, \( O(nB) \) server storage and completes in at most \( 4n \log n - 3.6n \) read/write operations.

The preceding list details algorithmic advances in the fields of succinct data structures and oblivious storage. All the results derive from mathematically proven performance characteristics and represent some measurable improvement over prior work. In addition, Chapters 5 and 6 supplement the theoretical material with proof-of-concept experimental work. This study represents the first formal treatment of the recency problem. Furthermore, it expands the practical scope of the recency query and demonstrates its relevance in field of oblivious storage.

The substantive work in this thesis is presented in manuscript form. Each substantive chapter contains a preamble and the manuscript of either the published or currently-under-review version of the work. We now present a literature review that establishes the research context for these manuscripts.
Chapter 2

Literature Review

Our literature review has two core components. In the first component, we detail some prior solutions to the recency query. This will provide the reader with a context sufficient for assessing the advances made within our own work. The solutions are drawn from two distinct problems: the identification of hot data and the calculation of stack distance. For the second component, we narrow the focus and situate each Research Question within its own literature. Here, we also present the material that we build on for our main contributions. We signpost the relevant subsections with their matching research questions and each can be read as a preamble to its corresponding chapter. We begin by covering some preliminary results.

2.1 Preliminaries

The preliminaries cover key concepts that are used throughout the literature review. It also serves as a warm up for later subsections that cover small memory data structures.

Succinct data structures

A data structure is succinct if it has a memory commitment close to the information-theoretic lower bound and admits the prescribed query operation efficiently. Although the notion of efficient here is not rigorous, a succinct data structure typically aims for runtimes that are comparable to the state-of-the-art for non-succinct representations. We formalize the memory requirement in the following definition.

**Definition 3.** If a minimum of $B$ bits are required, in the information-theoretic sense, to store the data, a succinct data structure occupies $B + o(B)$ bits of memory and supports query and updates efficiently.

For example, there are $\binom{m}{n}$ distinct subsets $G$, with $|G| = n$, of a universe $U = [m]$. Therefore, under a fixed coding, a representation of the set $G$ must occupy at least $B(m, n)$ bits on average, where:

$$B(m, n) = \left\lceil \log \binom{m}{n} \right\rceil \geq n \log \frac{m}{n}.$$ (2.1)
Following Definition 3, a succinct representation of the set $G$ uses $B(m,n) + o(B(m,n))$ bits. Succinct data structures have enjoyed application in memory scarce settings, such as the representation of large genomes [26, 41]. Examples of succinct representations include, but are not limited to, dynamic sequences [115, 128], dictionaries [53, 129] and trees [109, 116].

**Rank and Select**

The rank and select operations constitute one of the most important contributions on succinct data structures. They can be defined as follows.

**Definition 4.** For a string $C$ on alphabet $\Sigma$,

- $\text{rank}_b(C, i)$: given $i \in \{0, \ldots, n-1\}$ and $b \in \Sigma$, return $|\{ j \in \{0, \ldots, i\} \mid C[j] = b\}|$, i.e., the number of occurrences of character $b$ in the subsequence $C[0 \ldots i]$.

- $\text{select}_b(C, j)$: given $j \in \{0, \ldots, n-1\}$ and $b \in \Sigma$, return $\min \{x \mid \text{rank}_b(C, x) = j\}$, i.e., the index of the $j$th occurrence of $b$ in $C$.

As a warm up, we give an overview of a canonical solution to the rank query. Jacobson demonstrates how to support constant time rank operations over a plain-form bitmap [86]. They construct a two-level solution. For a bitmap of length $n$, the top level stores rank answers every $r = \log_2 n$ bits. The second level then stores rank answers relative to the start of each segment. The answers are spaced $\log(n/2)$ bits apart and thereby require $O(\log \log n)$ bits per sample. A universal lookup table is used to complete the query within a sub-sample. Note that each level requires $o(n)$ bits\(^1\). An image of the data structure is available in Figure 2.1. A similar approach can be applied for select queries\(^2\). These techniques can be used to support constant time rank and select queries for alphabets of size $\log \phi n$, where $\phi \in (0, 1)$. For such moderately-sized alphabets, Ferragina et al. [56] provide a plain-form representation with negligible redundancy and Goylynski et al. [65] demonstrate how to compress the string without sacrificing efficiency. Larger alphabets, of size $\omega(\log^2 n)$, require a more sophisticated data structure named the wavelet tree [113] and depart from plain-form representations.

**Approximate data structures**

The field of succinct data structures offers a range of techniques and tools for supporting various queries efficiently in small memory. An alternative approach to take is approximation\(^3\). By allowing the query to err with some probability, we circumvent the information theoretic lower bound. We illustrate this through the set membership query. By Equation 2.1, a set $S \subseteq [m]$, of size $n$, requires at least $n \log(m/n)$ bits of space (to support exact membership queries). By introducing a one-sided false-positive rate\(^4\) of $\delta \in (0, 1)$, the lower bound is reduced to

\[^1\]For example, the top level can be stored in a array of $\log(\log^2 n) \cdot n/\log^2 n = o(n)$ bits.

\[^2\]Instead of dividing the bitmap into segments of equal width, the segments are sliced such that they have the same number of 1 bits. Then, the length of the segment will determine what representation is used at the next level.

\[^3\]There are also succinct approximate data structures [53].

\[^4\]If $x \in S$, then a query on $x$ must report Yes. If $x \notin S$, then a query on $x$ with probability at least $1 - \delta$ reports No, but with probability at most $\delta$ reports Yes (i.e., a false positive).
The Recency Problem and its Applications

William L. Holland

The static rank data structure of Jacobson [86]. To evaluate the query \( \text{rank}_1(B, 9) \), the operation begins at the top level \( L_1 \). The latter provides global rank answers at 6-bit intervals. Therefore, we compute \( i_1 = \lfloor 9/6 \rfloor = 1 \) and we add \( L_1[i_1] = 4 \) to the query output. We then move to the second level \( L_2 \), which provides local (relative to the bucket) rank answers every two bits. We compute \( i_2 = \lfloor (9 \mod 6)/2 \rfloor = 1 \) and add \( L_2[i_1][i_2] = 1 \) to the query output. Now, the segment \( B[8...9] \) is small enough to evaluate rank queries in constant time with a universal lookup table. Thus, \( \text{rank}_1(B, 9) = 4 + 1 + 1 = 6 \).

\( n(\log(1/\delta) - O(1)) \) bits [53] and the dependence on the alphabet size is removed. The lower bound for approximate set membership expresses a trade-off between accuracy and memory; we can authorize more compact data structures, but at the cost of lost query accuracy.

The Bloom filter (BloomFilter) is a well-known solution to the approximate membership problem and is a structure that appears frequently throughout our literature review. Given a set \( S \subset U \) of items, the BloomFilter maintains a Boolean array \( B \) of width \( v \) and stores a hash function \( h: U \rightarrow [v] \). Using the hash function to map the set onto the array, the array bits are set in the following way:

\[
B[i] = \begin{cases} 
\text{TRUE}, & \text{if } (h(x) = i) \land (x \in S), \\
\text{FALSE}, & \text{otherwise}.
\end{cases}
\]

To query an item \( x \), we check if \( B[h(x)] \) is set to TRUE. A false positive occurs if there is a collision. That is, if \( h(x) = h(y) \) for \( x \notin S \) and \( y \in S \). To mitigate the impact of collisions we have two options: (1) increase the width of the array; (2) add more hash functions. Note, if we add too many hash functions the accuracy will degrade. The parameterization of the filter, that is, the choice of width \( v \) or the number of hash functions, is an optimization problem. For example, if at most \( n \) items are “stored” in the BloomFilter, then using \( (v/n) \ln 2 \) hash functions minimizes the false positive rate for fixed \( v \) [106]. An example of a BloomFilter with two hash functions is present in Figure 2.2.

**Hashing**

Typically, the analysis of randomized algorithms relies on the properties (or assumptions) of a particular hash function. For instance, our prior statement about the optimal number of hash

\[ n(\log(1/\delta) - O(1)) \] bits [53] and the dependence on the alphabet size is removed. The lower bound for approximate set membership expresses a trade-off between accuracy and memory; we can authorize more compact data structures, but at the cost of lost query accuracy.

The Bloom filter (BloomFilter) is a well-known solution to the approximate membership problem and is a structure that appears frequently throughout our literature review. Given a set \( S \subset U \) of items, the BloomFilter maintains a Boolean array \( B \) of width \( v \) and stores a hash function \( h: U \rightarrow [v] \). Using the hash function to map the set onto the array, the array bits are set in the following way:

\[
B[i] = \begin{cases} 
\text{TRUE}, & \text{if } (h(x) = i) \land (x \in S), \\
\text{FALSE}, & \text{otherwise}.
\end{cases}
\]

To query an item \( x \), we check if \( B[h(x)] \) is set to TRUE. A false positive occurs if there is a collision. That is, if \( h(x) = h(y) \) for \( x \notin S \) and \( y \in S \). To mitigate the impact of collisions we have two options: (1) increase the width of the array; (2) add more hash functions. Note, if we add too many hash functions the accuracy will degrade. The parameterization of the filter, that is, the choice of width \( v \) or the number of hash functions, is an optimization problem. For example, if at most \( n \) items are “stored” in the BloomFilter, then using \( (v/n) \ln 2 \) hash functions minimizes the false positive rate for fixed \( v \) [106]. An example of a BloomFilter with two hash functions is present in Figure 2.2.

\[ n(\log(1/\delta) - O(1)) \] bits [53] and the dependence on the alphabet size is removed. The lower bound for approximate set membership expresses a trade-off between accuracy and memory; we can authorize more compact data structures, but at the cost of lost query accuracy.

The Bloom filter (BloomFilter) is a well-known solution to the approximate membership problem and is a structure that appears frequently throughout our literature review. Given a set \( S \subset U \) of items, the BloomFilter maintains a Boolean array \( B \) of width \( v \) and stores a hash function \( h: U \rightarrow [v] \). Using the hash function to map the set onto the array, the array bits are set in the following way:

\[
B[i] = \begin{cases} 
\text{TRUE}, & \text{if } (h(x) = i) \land (x \in S), \\
\text{FALSE}, & \text{otherwise}.
\end{cases}
\]

To query an item \( x \), we check if \( B[h(x)] \) is set to TRUE. A false positive occurs if there is a collision. That is, if \( h(x) = h(y) \) for \( x \notin S \) and \( y \in S \). To mitigate the impact of collisions we have two options: (1) increase the width of the array; (2) add more hash functions. Note, if we add too many hash functions the accuracy will degrade. The parameterization of the filter, that is, the choice of width \( v \) or the number of hash functions, is an optimization problem. For example, if at most \( n \) items are “stored” in the BloomFilter, then using \( (v/n) \ln 2 \) hash functions minimizes the false positive rate for fixed \( v \) [106]. An example of a BloomFilter with two hash functions is present in Figure 2.2.

\[ n(\log(1/\delta) - O(1)) \] bits [53] and the dependence on the alphabet size is removed. The lower bound for approximate set membership expresses a trade-off between accuracy and memory; we can authorize more compact data structures, but at the cost of lost query accuracy.

The Bloom filter (BloomFilter) is a well-known solution to the approximate membership problem and is a structure that appears frequently throughout our literature review. Given a set \( S \subset U \) of items, the BloomFilter maintains a Boolean array \( B \) of width \( v \) and stores a hash function \( h: U \rightarrow [v] \). Using the hash function to map the set onto the array, the array bits are set in the following way:

\[
B[i] = \begin{cases} 
\text{TRUE}, & \text{if } (h(x) = i) \land (x \in S), \\
\text{FALSE}, & \text{otherwise}.
\end{cases}
\]

To query an item \( x \), we check if \( B[h(x)] \) is set to TRUE. A false positive occurs if there is a collision. That is, if \( h(x) = h(y) \) for \( x \notin S \) and \( y \in S \). To mitigate the impact of collisions we have two options: (1) increase the width of the array; (2) add more hash functions. Note, if we add too many hash functions the accuracy will degrade. The parameterization of the filter, that is, the choice of width \( v \) or the number of hash functions, is an optimization problem. For example, if at most \( n \) items are “stored” in the BloomFilter, then using \( (v/n) \ln 2 \) hash functions minimizes the false positive rate for fixed \( v \) [106]. An example of a BloomFilter with two hash functions is present in Figure 2.2.

**Hashing**

Typically, the analysis of randomized algorithms relies on the properties (or assumptions) of a particular hash function. For instance, our prior statement about the optimal number of hash
functions assumes that items are distributed according to a perfect hash function. That is, items are distributed uniformly and independently across the array. Although this certainly simplifies the analysis and improves the result\(^5\), the assumption on the distribution of the hash function expresses an ideal and perfect hash functions do not exist in any efficient form\(^6\). Thus, concrete hash functions that are small, fast, and random, are sought-after artefacts. Generally speaking, the desired properties are a small representation of the function, fast evaluation and strong independence properties.

Set membership is a key application of hashing. A hash table stores items (and possibly additional associated information) in an array with each location determined by a hash function (or multiple hash functions). A key hash table construction, introduced by Rasmus Pagh \([118]\), which appears throughout our review, is the cuckoo hash table (CuckooTable). The table comprises two arrays, \(T_1\) and \(T_2\), each with its own hash function. For \(i \in \{1, 2\}\), let \(h_i\) denote the hash function for array \(T_i\). An item \(x\) can be inserted at either of the two locations appointed by the two hash functions, \(T_1[h_1(x)]\) or \(T_2[h_2(x)]\), and nowhere else. The insertion procedure attempts to fill one of the two designated locations. If these are occupied, an item is evicted and the procedure attempts to place the evicted item at the alternate location. If \(y\) is evicted from \(T_1[h_1(x)]\), then we attempt to place it at \(T_2[h_2(y)]\). This may trigger a sequence of evictions that continues until all items are settled at a designated location. With some probability, an insertion will fail (the evictions continue in a cycle) and the table is rebuilt with fresh randomness by selecting a new hash function. Each insertion takes expected constant time; access and deletion are worst-case constant time. Variations include storing unallocated items in a supplementary stash to circumvent insertion failure \([91]\) and a two-level variant that supports a succinct representation with worst-case constant operations \([11]\).

\(^5\)With “less” independence, we would need to increase the size of the BloomFilter to achieve the same false positive rate when compared to an implementation with a perfect hash function.

\(^6\)For example, we could take the family \(\mathcal{H}\) of all functions \(h\) that map \(U\) to \(v\). Then selecting one of these functions uniformly at random would suffice. However, as there at \(|U|^v\) such functions, \(v \log |U|\) bits are required to represent a single function, which exceeds the allocation of a BloomFilter.
The Recency Problem and its Applications

William L. Holland

2.2 Prior solutions

We now consider some prior solutions and precedents to the recency query. The solutions cover both item-recency (see Definition 2) and time-recency (see Definition 1). The solutions are drawn from two distinct problems; the identification of hot data and the calculation of stack distance. A notable theme that recurs throughout this body of prior work is that solutions are heuristic in nature and lack a formal treatment. In addition to performance improvements, this is what differentiates our work from its precursors.

Identification of hot data

Hot data is a concept that arises for effective data placement in memory management applications. Informally, the temperature of a block of data corresponds to the likelihood that it will be accessed again in the near future. In other words, it expresses how relevant a data block is; a given logical block address is hot if it is relevant. For effective administration of the storage hierarchy, it is imperative that a memory management system places hot data in fast memory. Therefore, a key design issue in storage systems entails how we measure the temperature of a data block.

Shen et. al classified prior work into three categories [133]. The first group refers to time-based algorithms. Typically, these algorithms maintain the most recent access time, that is, the exact time-recency, of each block [105, 131]. This approach can achieve adequate hot data identification. However, storing a statistic for each block is memory intensive and limits the range of application scenarios.

The second group captures algorithms based on cache-replacement policies. As a simple example, a least recently used (LRU) list can be used to implement the hot data policy. The list is fixed width – the width may depend on the amount of memory available for the application – and contains all the hot data. At the arrival of a new logical block address; if the address is hot, and therefore resides in the list, it is moved to the head of the list. On the other hand, if the address is cold, it is placed at the head and the tail, the eldest hot block, is evicted. Notably, the distance of a block from the head of the list measures exact item-recency for hot data blocks. This approach has two drawbacks. The first is that any accessed block, regardless of how relevant it is, will be classified as hot and may demote a more relevant block. To remedy
The Recency Problem and its Applications

William L. Holland

this issue, Chang et al. introduce a two-level LRU list [33]. The top level contains all the hot blocks and the second level contains candidates. Both levels implement the LRU policy. On arrival, a block that does not reside in the data structure, at either level, is placed, in a probationary sense, in the candidate list and evicts the eldest candidate. In contrast, if the incoming block resides in the candidate list, it is promoted to the hot list and, subsequently, denotes the least recently used hot block to the candidate list. Thus, the structure privileges blocks that have arrived both recently and frequently. An illustration of the data structure is available in Figure 2.3. However, despite striking an improved balance between recency and frequency, the two-tier LRU cache requires linear scans of the component list structures and is considered computationally intensive. In addressing this drawback, Shen et al. partition the address space into groups with a hash function [133]. Subsequently, each group is assigned a small fixed-width LRU list. Thus, the expense of the linear scan is greatly reduced. Unfortunately, with this solution we return to the first drawback, wherein all accessed blocks are classified as hot. Further, this approach will likely amplify this weakness; if irrelevant blocks are hashed to the same group, they will take a longer time to be evicted.

The solution of Shen et al. illustrates how hashing can be used to reduce the cost of scanning a list by randomly partitioning it into disjoint sublists. The data structure we present in Section 3 adopts the same strategy. We provide additional innovations that allow for a succinct representation of the list and support for more complex queries.

Frequency based hot data identification

The first two groups, categorized as time- and cache-replacement based algorithms, equate block relevance with some measure of recency. In contrast, the third group of algorithms associate relevance with the access frequency of the block [77, 120, 121]. This design assumes that a data block that occurs frequently is more likely to occur than a data block that occurs infrequently. Significantly, this class of algorithms is notable for accommodating low-memory solutions. As such, the group is suitable for applications where the mechanisms for hot data identification operate in constrained resource environments. As noted by Hsieh et al., with reference to specific applications, “with potentially very limited computing power from a flash-memory controller or an embedded-system microprocessor, it is of paramount importance to have efficient designs for space management methods” [77]. Typically, frequency-based algorithms are sublinear not only in the size of the memory but also in the number of hot data blocks.

However, for many applications, the notion that frequency equates to relevance is insufficient and ignores changes to the distribution of the workload. Online access patterns have a strong impact on the performance of flash-memory storage systems due to garbage collection activities [89, 148] and many workloads exhibit high temporal localities [34]. Thus, the recency of a data block is an additional indicator of its relevance and can be utilized to improve classification accuracy. In turn, a number of hot data mechanisms from this group combine a measure of recency with their frequency estimate [120, 121]. Following the above observations, Park et al. identify three design requirements for hot data identification [120]: effective capture

---

7By setting the width to a constant, we observe constant time updates.
of both frequency and recency; small memory consumption; and low computational overhead. Many solutions that follow this blueprint contain a component that produces a recency estimate in low memory. We will now detail some of the key low memory solutions – all constructed in the context of flash memory management – and evaluate the proposed methods for recency estimation. This line of work demonstrates how to use Bloom filters to answer recency queries and we incorporate these insights into our own construction, HistoricalMembership (Section 4).

Low memory solutions

Heish et al. propose a count-min sketch type data structure [36] that performs classification based on time-decayed frequency information [77]. Similar to above, the address space is partitioned, via a hash function, and groups of addresses share a single counter. To privilege the more recent past, the counters are periodically decreased. In this way, the mechanism acknowledges that two data blocks with the same frequency may not be equally relevant and the more recent accesses are given a higher weight. Accuracy is increased by repeating the partitioning process and selecting the most accurate estimate. The construction offers superior memory and time efficiency over canonical solutions based on caching techniques [39] or those that support exact time-recency queries on large hash tables.

Understanding the importance of recency information, Park et al. were the first to incorporate (what they call) a more fine-grained measure of recency in their identification mechanism [119, 120]. The scheme employs multiple Bloom filters that are updated in a round-robin fashion to partially encode the order of the most recent accesses. The scheme contains \( l \) separate independent Bloom filters, \( V_0, \ldots, V_{l-1} \). At the arrival of the next logical block address \( x \), if \( V_i \) was updated at the previous access, then \( x \) is added to \( V_i' \), where \( i' = i + 1 \mod l \). Periodically, every \( T \) updates, a BloomFilter is erased (the sequence of erasures also operates in a round-robin). Thus, the filters have different ages and are used to generate a recency weighted frequency estimate\(^8\). This setup supports a simple procedure for an estimate of time-recency: find the youngest BloomFilter a block belongs to and count the number of updates since it was last erased. Due to the structured manner in which filters are updated and erased, the calculation is straightforward. If a BloomFilter has observed \( j \) erasures across the group since its last erasure, its age is at most \( (j + 1) \cdot T \). Thus, we have a low-memory data structure that supports time-recency queries with bounded absolute error.

The granularity\(^9\) of the estimates can be greatly improved with some adjustments to the update procedure. As the updates operate according to the round-robin agreement, an old BloomFilter contains updates from a recent round. With no method to distinguish between old and recent updates, Recency information is destroyed. To better preserve this information, in an alternative to the directive of the round-robin, individual Bloom filters can be updated in

---

\(^8\)To produce an estimate, each BloomFilter is assigned a weight corresponding to its age. Younger filters have larger weights. A measure of relevance is constructed by adding the weights of the filters that contain a given block. Note, a crude frequency estimate can be constructed by counting the number of filters a block belongs to.

\(^9\)It is interesting that the authors claim that, in comparison to the work of Heish et al. [77], their structure captures “fine-grained recency”. Here, granularity is measured by the number of Bloom filters. As, for a fixed-memory allocation, more (and therefore smaller) Bloom filters would provide for a tighter bound, there is a sense in which this is true. Nonetheless, the bound is still crude when measured against relative error.
The Recency Problem and its Applications

William L. Holland

Figure 2.4: Sliding (batched) Bloom filters for recency estimation. Each round has \( T = 5 \) updates and each BloomFilter is initialised with one hash function. The rightmost BloomFilter is the youngest filter. The image displays the state of the data structure at the completion of a round. Membership queries state that item \( x \) belongs to the second and third BloomFilter (from the right). Therefore, \( 5 \leq r_t(x) \leq 9 \). For hot data identification, a weight can be assigned to each BloomFilter and the mechanism can sum the filters with positive membership queries. At the start of the next query, the leftmost BloomFilter (in red) is erased and a new BloomFilter is initialised to absorb the next \( T \) updates.

ACM. Accordingly, each BloomFilter now corresponds to a single period (of length \( T \)) of consecutive updates and the filters are stored in a sequence, sorted according to age. At the end of a period of \( T \) updates the eldest BloomFilter is deleted and a new BloomFilter is initialised at the front of the sequence. The new BloomFilter absorbs the next \( T \) updates. As a result, each BloomFilter contains a batch of contiguous updates and we no longer leak recency information into older filters. This is identical to the naive solution for sliding membership [111] (see Section 2.4) and admits time-recency queries with bounded absolute error of at most \( T \). An image of the batched BloomFilter structure is available in Figure 2.4. This improvement does, however, come at a cost, and now frequency information is destroyed within a given period of \( T \) updates. Thus, in an inversion of the round-robin selection procedure, the batched selection procedure privileges recency information over frequency information.

To balance this trade-off, Ha et al. [72] employ a combination of the two selection procedures. The approach involves updating the filters in a round-robin of smaller batches. The size of the batches is determined by a *dynamic* selection algorithm that is conditioned by the workload. For example, roughly speaking, recent items are more relevant than frequent items for workloads with high temporal locality. In this instance, the selection algorithm should favour a batched approach. The authors utilize an approximation of *stack distance* to measure temporal locality\(^{10}\) and inform the dynamic selection algorithm. Stack distance is equivalent to item-recency and is an important measure in recency-sensitive applications. We expand on some methods for calculating stack distance below.

Through this subsection we have observed approaches that provide estimates for time-recency with bounded absolute error in small memory. The approaches encapsulate a key technique for time-recency estimation; storing collection of items in a set-membership structure and keeping track of the age of the structure. All items in the set-membership structure are given the same time-recency estimate. In Chapter 4, we build on this insight and construct a data structure, HistoricalMembership, that is both *succinct* and provides time-recency estimates with bounded *relative* error.

\(^{10}\)Workloads with low stack distance have high temporal locality.
Calculation of stack distance

Stack distance refers to the position of an item in a stack. It is determined by the number of distinct items placed on the stack since the item was last queried. Thus, it is equivalent to our definition of item-recency (see Definition 2). Stack distance is an important measure in a number of distribution sensitive applications, particularly those where the distribution of the workload is subject to change. It was originally designed for virtual page modeling [25, 101] and has since been used to model and estimate cache misses for least-recently-used (LRU) based cache policies [5, 29, 55]. Different variants of the policy are implemented in modern processors, such as the Tree-PLRU [138] or the Bit-PLRU [99]. As noted above, an estimates of average stack distance are used to measure the temporal locality of workloads [72].

A stack processing algorithm requires calculating the stack distance for each item in a given input workload and the literature has sought to construct time-efficient methods for computing this metric. We will cover some approaches below and detail the techniques that support run-time efficiency. At a high level, these techniques will appear again in this review when we look at string processing and list-indexing.

Almasi et al. decompose the LRU stack algorithm into three components [5]. For a input item, these components are defined as follows.

**Definition 5.** The components of an LRU stack are:

- **search:** find the location of the memory reference in the stack.
- **count:** compute the stack distance by counting the current location in the stack. If the item does not belong to the stack, then the distance is undefined.
- **update:** move the item to the top of the stack.

The canonical solution is to employ a linked list. The search and count procedures can be completed simultaneously with a linear scan from the head. The update component operates on the list pointers to remove the queried item from its place in the chain and place it at the head of the stack. This arrangement is simple, but computationally intensive due to the linear scans. To improve efficiency of the search and count routines there are two approaches. First, we can attach auxiliary data structures to the linked list to support sublinear search and count operations. Updating the auxiliary structures typically comes at the price of increasing the (amortized) cost of the update operation. Second, we can adopt a different representation of the list that removes the need for linear scans.

As an example of the first class of solution, a hash table can point to a specific node in the linked-list, thereby completing a search operation, in constant time. However, a hash table does alone does not support efficient calculation of the stack depth. To reduce the cost of a linear scan of the stack, a variation of the skip-list [127] can be employed. Here, for a parameter $s$, every $s^{th}$ item in the stack is copied and placed in an order-preserved list that sits on top of the stack. A link is maintained between each copy and its original node. The copied nodes are referred to as markers. Thus, after a search is completed, the count query scans to the next marker, jumps to the copy node and continues to scan the auxiliary list, increasing the count by $s$ at each node. If the queried item is at stack distance $y$, the operation requires $O(s + y/s)$ time.
Figure 2.5: Markers list algorithm with skips $s = 3$. To calculate the stack distance of item 35, a hash table is used to locate the correct node in bottom linked-list in constant time. The procedure then moves to the left and counts the nodes traversed until the first marker (nodes 25 and 23). At the marker, the procedure moves to the list of markers and then counts the markers traversed before reaching the head (skipping $s$ nodes at a time). The stack distance is $q_t(35) = 1 + 1 + s + s = 8$.

Note, that the update procedure involves modification of the auxiliary list and all markers between the queried item and the head need to be shifted up one position. An image of the markers list is provided in Figure 2.5. The markers list is a moderately efficient solution to item-recency. We are now going to look at some more efficient techniques and conclude the subsection with some limitations and areas for improvement.

An alternative approach is to construct a different representation of the stack. Bennett and Kruskal (BennettKruskal) [25] demonstrate that a stack can be implemented with a bitmap $B$, where, at time $t$,

$$B[i] = \begin{cases} 1 & \text{if } q_t(x) = i \text{ for some } x \text{ in the stack} \\ 0 & \text{otherwise} \end{cases}$$

and a map (implemented by a hash table), where,

$$P(x) = t - r_t(x).$$

The functions $r_t$ and $q_t$ are defined, respectively, in Definitions 1 and 2, and represent our measures of recency. Thus, the bitmap indexes the timestamps of the most recent arrival of each item in the stack and the hash table records the most recent timestamp of each item. For item $x$, the stack distance is calculated by deducing index the $P(x)$ that references the most recent arrival of $x$ and counting the number of 1s that proceed this point in the bitmap:

$$\text{count}(x) = \sum_{i=P(x)}^{i-1} B[i].$$

As this summation incurs the same asymptotic cost as a linear scan, Bennett and Kruskal introduce an auxiliary structure to facilitate faster counting. The structure can be interpreted as a tree with branching factor $\gamma$. Each leaf node represents a $\gamma$ width section of the bitmap. Each internal node stores a count of the number of 1 bits that reside in the leaf nodes of its subtree. To calculate the stack distance, the 1 bit that corresponds to a queried item $x$ belongs in the $[P(x)/\gamma]^{th}$ leaf node. First, the number of 1 bits that proceed $x$ within the leaf node is calculated in a linear scan. Then, the remainder of the sum is accumulated with a traversal up the tree to the root node by aggregating the counts of the right siblings of each internal node.
As each node requires $O(\gamma)$ time to complete the sum, the count query, assuming the hash table grants constant time access to the leaf node completes in $O(\gamma \log \gamma(t))$ time. Although the authors do not propose this, the branching cost can be reduced, by evaluating the sum of the counts of left siblings on a partial sums data structure [150].

**Definition 6 (Partial Sums).** A partial sums data structure stores an array of integers $A[1 \ldots \gamma]$ and admits the following procedures.

- **add** $(i, \delta)$: perform $A[i] \leftarrow A[i] + \delta$, where $\delta = \log \Theta(1) t$.
- **sum** $(j)$: return $\sum_{i \leq j} A[i]$, where $\sum_{i \leq j} A[i] = O(t)$.

On problem size $\gamma = O(\log^\phi t)$ (where $\varphi \in (0, 1)$), both operations can be performed in $O(1)$ amortized time [52]. Thus, the cost of a count query is bound by the height

$$O(\log \log \gamma(t)) = O\left(\frac{\log t}{\log \log t}\right)$$

of the tree. For item $x$, an update involves: (1) flipping the bit $B[P(x)] \leftarrow 0$; (2) inserting a new bit $B[t] \leftarrow 1$; (3) propagating the the changes to $B$ through the internal nodes of the auxiliary tree$^{11}$; and (4) updating the hash table $P(x) \leftarrow t$. This technique, of laying a tree over a list or string in the name of added efficiency, is standard within the literature and we cover its additional applications when we introduce dynamic strings and indexed lists in Section 2.3. An image and example of BennettKruskal is available in Figure 2.6.

Almasi et al. also adopt the BennettKruskal (bitmap and recency-map) representation of the stack [5]. However, under the observation that once a 0 bit is created it is never destroyed, they propose to count the number of 0 bits. Specifically, they store all the distinct intervals of consecutive 0 bits in a search tree, otherwise known as an interval tree. Asymptotically, for a stack of length $n$, as there are $O(n)$ non-overlapping intervals, this has the effect of removing the dependence on $t$ (the stream length) obtained in the Bennett and Kruskal approach. As the leaf nodes are now of non-uniform size, a leaf (that contains the searched index) cannot be accessed in constant time. Consequently, queries begin at the root and the count is aggregated during a search for the interval (leaf node) that contains the query index. By utilizing the same branching factor as above and a partial sums structure for summation in the internal nodes, a count query takes $O(\log n / (\log \log n))$ time. Nodes in the interval tree will be periodically merged as a result of the update operation and may cause the tree to become unbalanced. The weight balanced $B$-tree from Dietz [52] prevents the amortized cost of rebalancing from exceeding the height of the tree. Consequently, the $O(\log n / \log \log n)$ time cost for all operations is optimal [60].

The aforementioned data structures can support efficient update and query operations. However, with the heavy use of pointers and the addition of auxiliary structures of size proportional to the stack, the solutions are non-succinct. Our work in Section 3 is able to resolve this shortcoming. We now move our attention to indexed list data structures, which can be viewed as a more general representation of the stack.

$^{11}$Each internal node on the path from leaf $P(x)/b$ to the root is decremented by one and each internal node on the path from leaf $t/b$ to the root is incremented by one.
Figure 2.6: The BennettKruskal stack distance data structure for the sequence of accesses \(\langle 4, 8, 4, 7, 6, 5, 6, 2, 1, 6, 4, 2, 4, 4, 3, 2 \rangle\). To calculate the stack distance of item 7, we first retrieve the time stamp of the most recent arrival from an array (or hash table if the stack changes in size). The timestamp is able to index into the bit corresponding to 7 in the bitmap. Each bit represents a leaf in an auxiliary tree. We then traverse the tree on the leaf-to-root path and accumulate the counts of all right siblings on the path. In this instance, the stack distance is \(1 + 5 = 6\).

2.3 List indexing

This subsection can be viewed as a preamble to Section 3 and the core literature to which Research Question 1 belongs. The purpose of this subsection is to cover prior solutions to the list indexing problem (see Problem 1).

List indexing is a popular problem in the data structures literature from the previous century. A list that supports these operations\(^{12}\) is named an indexed list. Notably, the position query is an inbuilt function in programming languages such as Python and Java. For a list encoding an arbitrarily ordered set \(\mathcal{L} \subseteq [m]\) of size \(n\), a lower bound of (amortized) \(\Omega(\log n / \log \log n)\) time per operation is due to Fredman and Saks [60]. Note that the insert operation encodes an ordering over the set of items stored in the list.

The most straightforward solution is a linked-list. However, its linear-time query and update operations are inefficient. Of interest to us here are mechanisms for navigating a list efficiently. The BennettKruskal implementation of the stack gave us a preview of types of techniques that support efficient search. However, this solution is particular to a stack representation, as it takes advantage of the fact that queried items are moved to the head. List indexing articulates a more general approach to navigating through lists. The move-to-front

\(^{12}\)Traditionally, it is assumed that the update sequence does not admit any repetitions. We follow this precedent here. If a malicious input sequence is allowed, the insert algorithm can be modified to check for a prior occurrence. At this point, a rule regarding which element is kept would have to be agreed upon. Alternatively, the operations could be modified to allow for repetitions. For example, the position function could return the lowest index where an item appears.
The Recency Problem and its Applications

William L. Holland

Figure 2.7: List indexing with a binary tree for list \( L = [4,5,8,3,7,6,9,2,1,0] \). All leaf nodes are list items. The blue lines note links in the list. Dashed lines denote links to left siblings. To calculate position\((L,2)\), we first access leaf 2 via a hash table. We then traverse the leaf-to-root path accumulating the counts in all left siblings. In contrast, the operation index\((L,7)\) starts at the root node. We then check the count at the left sibling and infer that the index is in the right subtree. We then recurse on the right child with index\((L_r,(7 - 5))\) and continue until a leaf node is encountered.

rule can be applied to an indexed list to achieve a measure of stack distance. Similar to above, prior solutions are focused on time-efficiency. In contrast, a key contribution we make – the focus of Chapter 3 – is the first construction of a succinct indexed list.

Prior solutions

The literature on list indexing assumes that item pointers are readily available as arguments to the operations. Thus a typical data structure appears in three components; a hash-table to retrieve item pointers; the underlying linked-list; and the supporting structure to support (fast) indexing.

A non-optimal solution entails a straightforward application of a balanced (or height-bounded) binary tree. List items are stored, in order from left to right, in the leaves of the tree and internal nodes store a count of the number of leaves (or, the size of the corresponding sublist) in the subtree rooted at the internal node. An index\((i)\) query begins at the root and, following the logic of the counts stored at internal nodes, can branch into the sublist containing the correct index. Conversely, position\((x)\) begins at a leaf and accumulates a global index on a leaf-to-root path; the count of any left sibling at an internal node gets aggregated to the current index. Updates require recourse to a balancing criteria and may entail some restructuring\(^{13}\). All operations take \(O(\log n)\) time (worst-case or amortized depending on the choice of tree) and the tree requires \(\Theta(n \log m)\) bits to store.

This idea was extended by Dietz with what are now fairly standard techniques [52].

\(^{13}\)Possible constraint data structures include AVL trees [59] or red-black trees [21].
The Recency Problem and its Applications

William L. Holland

binary tree is replaced with a weight-balanced \( B\)-tree with branching factor \( \Theta(\log^\phi m)\), for \( \phi \in (0, 1) \). As before, internal nodes count the number of leaves in the subtree of which they are the root. On a leaf-to-root path, a position query accumulates the counts, at each internal node, of all left siblings. To remove dependence on the branching factor, the sum of the counts of left siblings can be evaluated in constant time using a partial sums data structure (see Definition 6).

For branching factor \( \gamma = \Theta(\log^\phi m)\), both operations can be performed in \( \Theta(1) \) amortized time. Thus the cost of a position query is bound by the height \( (\Theta(\log_{\log^\phi m} n)) \) of the tree. The index query, as before, proceeds on a root-to-leaf path and, at each node, identifies the branch that contains the correct index. Efficient navigation at the internal nodes is supported by the additional operation:

- **select_ps\((i)\)**: return the smallest \( j \) such that \( \text{sum}(j) \geq i \).

Although the partial sums structure of Dietz does not explicitly solve the latter query\(^{14}\), a constant time solution, on small problem sizes, is provided by Raman et al. [128]. Thus, the cost of both query operations is bound by the height of the tree and is thereby optimal at \( \Theta(\log n/\log \log n) \). Due to rebuilding requirements on both the partial sums structure and the weight balanced \( B\)-tree, update costs are amortized. The solution, which includes the hash table and the list\(^{15}\), requires \( \Theta(n \log m) \) bits.

Andersson and Petersson define an approximate version of the problem [8]. The approximate position query is permitted to err with a user-defined relative error and the approximate index query can retrieve any item from a neighborhood, of size proportional to the relative error, surrounding the correct index\(^{16}\). The added flexibility allows the authors to remove the dependence on the problem size from the update and query costs. For relative error parameter \( \varepsilon \in (0, 1) \), queries can be evaluated in constant worst-case time and updates in amortized \( \Theta(\varepsilon^{-2}) \) time.

The data structure contains a hierarchy of levels. The top level is a list of sublists and is representative of the full list of items. Constituting the second level, each sublist is itself a list of sublists, but now representative of a portion of the list. The level structure continues in this manner and the size of a sublist is logarithmic in the size its parent list (here size refers to the number of items and not the number of nodes in the list). Each level, a list of sublists, is implemented differently depending on its cardinality. For example, the top level is implemented with a packed array (of pointers to sublists in the second level) that supports slow updates but fast index queries. As the top level is updated rarely (only when child sublists are merged or split), the amortized update cost is small. Conversely, the sublists in the bottom level are small enough to be implemented by means of a global lookup table supporting constant time operations. Significantly, as lists in the bottom level have size \( \Theta(\log \log n) \), there are a constant

---

\(^{14}\)They augment the weight balanced \( B\)-tree with additional navigation pointers that implicitly implement the functionality.

\(^{15}\)Additional operations like successor, predecessor, return head, and return tail can easily be supported by augmenting any solution to the indexed list problem and by threading the list elements (in this case, linking the leaves together).

\(^{16}\)The queries are subject to some constraints, such as monotonicity which states that approximate position queries must respect the order of the list. In other words, for any pair of items \( x, y \) where \( \text{position}(x) < \text{position}(y) \), it follows that \( \text{approxPosition}(x) < \text{approxPosition}(y) \).
number of levels in the structure. Thus, the runtime of the operations does not depend on the height of the structure. With a small variation in the cardinality of the sublists at a given level, which is maintained through the merging and splitting of sublists, it is possible to estimate the number of list elements that precede a particular sublist and approximate queries can be executed during a single traversal.

A common theme among prior solutions is the focus on time-efficiency. The novelty of our work entails the construction of a succinct solution with time-efficiency that matches prior state-of-the-art. Before continuing, we need to acknowledge that the rank-select problem (introduced above in Definition 4) has some overlap with the list indexing problem. Further, it is one of the most studied problems in the succinct data structures literature and contains a long history of solutions. Thus, we differentiate these two problems below and, in doing so, draw out the significance of our work.

The rank-select problem

List indexing has proximity to the well known rank-select problem [66, 67]. Significantly, the respective query operations share some overlap. The select operation is equivalent to the position query and an access $C[i]$ is exactly the index query. Thus, a compressed rank-select data structure for dynamic sequences [75, 108, 114, 116] would act as a solution to the list indexing problem. The two problems are very close and, indeed, the rank-select problem inherits the time lower bounds of the list indexing problem. The main difference between the two problems is the assumption about the universe size. For the rank-select problem it is reasonable to focus on texts drawn from a small universe of characters. However, the solutions become intractable for problems on larger universes and, in particular, where repetitions do not occur. For example, with $m = |\Sigma|$, a state-of-the-art solution by Navarro and Nekrich has a memory allocation with a redundancy term of $O(m \log n)$ [114]. In the list indexing problem, where $m \geq n$, this leads to a non-succinct representation. Further, a solution by Munro and Nekrich [108] that supports “arbitrarily large alphabets” has a similar limitation. While the memory allocation is quoted as $H_k(C) + o(n \log m)$ bits\(^{17}\), it appears to require (implicitly) that $m \leq n$. The issue is that an auxiliary data structure is stored for each character in $\Sigma$. Resolving this issue is not as simple as assigning 0 bits for non-occurring characters as this introduces the need for a search structure\(^{18}\). Even a state-of-the-art dynamic succinct dictionary [11] would push the memory allocation over a succinct allowance. A succinct indexed list remedies this restriction on the universe size and is an open problem that we address here.

Our solution to list indexing, PrincessList, uses a rank-select data structure as a black box. For moderately sized alphabets, the standard solution is the wavelet tree [98, 113]. At a high level, the wavelet tree is a mechanism that recursively decomposes a string into disjoint substrings and uses small alphabet strings to indicate how the branching is organized. This concludes the background literature for Section 3.

---

\(^{17}\) $H_k(S)$ is the $k$th order empirical entropy of the string $S$. This term is somewhat redundant in our setting; as repetitions are not allowed, the empirical entropy is high.

\(^{18}\) Equivalently, we could hash the alphabet into a smaller alphabet.
2.4 Sliding Membership

The following exposition of the sliding membership literature can be viewed as a preamble to Section 4 and the core literature to which Research Question 2 belongs.

Did item x occur recently? This is the question posed by the sliding membership problem and one that is related to the time-recency query. Indeed, time-recency subsumes sliding membership; if you know when an item occurred you can infer whether it is recent. As a point of contrast, these two problems are qualitatively different to the list-indexing problem. The latter concerns the maintenance of the relative order of a collection of items. In comparison, sliding membership and time-recency are concerned with when particular items occur within a sequence of updates.

Time-recency can be solved trivially with a hash table by storing each item with its most recent timestamp\(^19\). This solution can also be used to solve sliding membership. However, as noted above, hash tables are space intensive in environments with resource constraints. In these settings, we are interested in low memory solutions. Thus, as an exemplar of small memory computing, we begin our exposition with an overview of the data stream literature. The overview will centre on particular models and techniques utilized to develop solutions to what we call temporal queries. Note that we have already seen some streaming data structures when we examined frequency-based hot data identification [77, 120]. Similarities to these solutions will be recognisable below.

The data stream

The data stream can be viewed as a sequence \( S = (s_1, s_2, \ldots) \) of data points. It can be summarised with three key principles: it is ordered, fast and large. These principles are what make data streaming problems challenging and interesting. We do not have the capacity to store the full stream in memory. Therefore, typically, we store a summary or sketch that enables approximate queries. Generally, the accuracy of these approximations is expressed in relative error. To support some query \( ? \), we have both an exact algorithm \( \Phi \) and a randomized approximate algorithm \( \hat{\Phi} \). Given two parameters \( \varepsilon, \delta > 0 \), a well designed \( \hat{\Phi} \) solves \( ? \) on input stream \( S \) with the following guarantee:

\[
(1 - \varepsilon)\hat{\Phi}(S) \leq \Phi(S) \leq (1 + \varepsilon)\hat{\Phi}(S), \quad \text{with probability } \geq 1 - \delta.
\] (2.2)

In terms of accuracy, \( \Phi \) outperforms \( \hat{\Phi} \). Thus, to justify this concession, \( \hat{\Phi} \) should offer some form of improved efficiency in either time or space.

The streaming paradigm emerged as an effective tool for approximating database queries [6]. It has since seen application in internetworking [28], functional approximation theory [142], computational geometry [82] and graph theory [17]. The preceding list of work registers a mutual concern and motivation; that of moving towards an approximate query framework to enable memory and time efficiency in the context of large volumes of data [42]. Many of the standard streaming solutions support time-indifferent aggregate queries (such as the frequency

\(^{19}\)Although, asymptotically, we may be concerned with the size of the timestamp growing too large (the stream could continue indefinitely). Below we will see techniques that reduce this cost.
The Recency Problem and its Applications

William L. Holland

Figure 2.8: Examples of the sliding window and time-decay streaming models for supporting recency sensitive queries. The rightmost item is the most recent occurrence. Let $f^W(x)$ denote the frequency of item $x$ on the window $W$ and $f^T(x)$ denote the time-decayed frequency of $x$ using a linear decay function. On the stream $⟨3, 2, 1, 2, 6, 7, 4, 6, 5, 4⟩$, $f^W(2) = 0$, $f^W(6) = 1$, $f^W(4) = 2$ and $f^T(2) = 0.6$, $f^T(6) = 1.3$, $f^T(4) = 1.7$.

of an item [36]). The latter are insensitive to the order of the stream. For example, permuting a stream will not change the frequency of an item. In turn, we define temporal queries as functions that express sensitivity to the order in which the data arrives. This applies to both the sliding membership and time-recency queries, where, now, a permutation of the input stream may alter the output of a query.

Two common approaches for constructing temporal queries involve either restricting queries to sliding windows [48] (what is the frequency of item $x$ over the last $W$ updates?) or in time-decaying an aggregate query [43] (more recent arrivals are given more weight). These models allow statistics that are time-indifferent to be embedded into a time-conscious algorithm. We provide examples for each model in Figure 2.8.

Of proximity to the recency problem is what one might call time-frequency queries, which return item frequencies for a specified time interval. Initial solutions are provided by Matusevych et al. [102] and involve sketching the stream as a sequence of time-indexed approximate frequency data structures. The sequence is subject to a time-decay model, wherein precision over old occurrences is permitted to atrophy under the assumption that recent information is more pertinent. With this decay, we can maintain a small memory allocation by routinely selling the accuracy of old datastructures for a smaller representation. Frequency estimates, for any specified interval, can be constructed from the sequence of data structures. Shrivastava et al. improve these accuracy and query efficiency with a more nuanced time-decay model [135]. Alternatively, Basat et al. aim to support time-frequency estimates on sliding windows, again under the assumption that “recent data items are more relevant than old ones” [19]. The authors apply similar tactics to above and partition the window into a sequence of time-indexed approximate frequency data structures.

The literature on temporal queries encapsulates two approaches for supporting recency-sensitive queries. First, recency can be prioritized through an appropriate streaming model, for example, by applying a sliding window or a time-decay model. Second, maintenance of
order in which items occur can be achieved by a divide-and-conquer approach that partitions
the stream into disjoint intervals, each with its own time-indifferent data structure. Notably,
both of these tactics are evident in solutions to the sliding membership problem.

Prior Solutions to sliding membership

Before proceeding, we provide a formal definition of sliding approximate membership.

Definition 7 (Sliding Approximate Membership). On parameters \( D, W \in \mathbb{Z}^+ \) and \( \delta \in (0, 1) \),
on a sequence of items \( S(t) = \langle s_1, s_2, \ldots, s_t \rangle \), from a universe \([n]\), the sliding membership of
item \( x \) in the length-\( W \) window of most recent items, \( S_W(t) = \langle s_{t-W+1}, \ldots, s_t \rangle \):

\[
x \in S_W(t) = \begin{cases} 
  \text{yes} & \text{if } x \in S_W(t), \\
  \text{no} & \text{if } x \notin S_{W+D}(t), \\
  \text{yes or no} & \text{otherwise.}
\end{cases}
\]

The answer is permitted to provide a false positive (return yes if \( x \notin S_{W+D}(t) \)) with probability
at most \( \delta \). There are no false negatives.

Significantly, we are not allowed to produce false negatives; if the item belongs to the
window, we must return yes. The parameter \( W \) denotes the window length; \( D \) denotes the slack
(the \( D \) arrivals that appear most recent to the window for which there are no restrictions on the
query answer); and \( \delta \) is the false positive rate. Some solutions solve the harder problem of
\( D = 0 \). In general, allowing some slack affords a more memory efficient solution [18].

As a benchmark, the simplest solution entails an array, of length \( n \), recording the most
recent timestamp of each item. This approach requires \( n \log t \) bits of memory. To reducing this
cost, Metwally et al. introduce a high-level solution – on the setting where \( D = 0 \) – where
one maintains a circular array (see Figure 2.9), of length \( W \), of item signatures [104]. A new
item arrival overwrites the tail of the (circular) array and the array head is incremented to point
at the new arrival. This reduces the problem to one of efficiently querying the array; an item
that is a member of the window is contained in the array. A data structure that supports set
multiplicity is used to monitor the circular array and is kept up to date as fingerprints enter
and depart the array. Membership is verified via a count greater than zero. To implement
set multiplicity, Metwally et al. introduce a counting Bloom filter, where the Boolean cells
of a standard BloomFilter are replaced by counts to permit deletions. A hash function can
be used to reduce the cost of the item signatures by mapping the key universe onto a smaller
domain. This mapping institutes collisions in the key domain and, consequently, introduces
a false positive rate. With a universal hash function [30], this domain has size \( W\delta \) and the
circular array, in turn, occupies \( W \log(W/\delta) \) bits. To support set multiplicity, the counting
Bloom filter requires \( W \log W \cdot \delta’(1/\delta) \) and operation costs of \( \log 1/\delta \) are proportional to the
number of hash functions.

Assaf et. al improve the previous solution by constructing a more efficient set-multiplicity
structure [54] named TinyTable. At a high level, the structure is a hash table that stores fing-
erprints and counts. This is a well-known approach for achieving BloomFilter functionality
in compact space and constant time [27]. TinyTable is based on rank-index hashing, a type of
dynamic hash-chaining that removes the space overhead of pointers [79]. The bit allocation is proportional to the counting Bloom filter, but has smaller update and query cost. Updates and queries take expected constant time but have poor worst-case behaviour\textsuperscript{20}.

In the context of detecting click fraud in online advertising networks, Zhang et al. [152] propose two solutions. The first, admitted as a naïve solution, involves dividing the window into fixed length sub-intervals, each of length $D$, and maintaining an individual BloomFilter on each interval (see Figure 2.4). A query is required to probe all $\lceil W/D \rceil + 1$ bloom filters. Thus, the additive error aggregating from the multiple instances needs to be accounted for. This leads to either a high false positive rate or a sub-optimal bit allocation. The second solution, named Timing Bloom Filter, stores time-stamps in the BloomFilter cells. Membership is verified if the oldest time-stamp returned belongs to the window. As the timestamps perpetually increase in size, the bit allocation exceeds that of prior approaches based on set multiplicity.

Liu et al. store (item-signature, time-stamp) pairs in a cuckoo hash table [96] (ExactCuckoo). As full item signatures are used, the solution provides exact solutions and satisfies the case $\delta = 0$. As with prior solutions, the memory allocation can be reduced by hashing onto a smaller address space and introducing approximate answers. Further, as the index in the hash table partly identifies the item, signatures require only $\log(1/\delta)$ bits with the use of universal hash functions. A stash, implemented as a linked-list, stores items that cannot be assigned a location in the table\textsuperscript{21}. After each update, a constant number of cells are scanned and expired items are deleted. Updates take expected constant time and queries worst-case constant as per cuckoo-hashing theory [118]. Memory utilization is almost 1/2, so the table contains a large proportion of empty cells.

The theoretical state-of-the-art solution belongs to Naor et al. [111] (OptimalSM). It is the only approach that accounts for slack and measures its benefit. A key contribution is the

---

\textsuperscript{20}The authors cite ‘constant time with high probability’, however this is not proven anywhere and ‘high probability’ is neither quantified nor parameterized

\textsuperscript{21}This happens in cuckoo hashing if and only if the hash graph contains a subgraph with more edges than nodes.
The Recency Problem and its Applications

William L. Holland

<table>
<thead>
<tr>
<th></th>
<th>Update time</th>
<th>Query time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExactCuckoo [96]</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(W \log n + \log W)$</td>
</tr>
<tr>
<td>OptimalSM [111]</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$(1 + o(1))(1 + D/W)(B + W \log(W/D))$</td>
</tr>
<tr>
<td>HistoricalMembership</td>
<td>$O(\log(eW))$</td>
<td>$O(\log(eW))$</td>
<td>$(1 + o(1))(1 + \varepsilon)(B + W \log(e^{-1}))$</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of HistoricalMembership with existing art. Term $B$ denotes the information-theoretic lower bound for storing a set of $W$ items from the universe $[n]$. ExactCuckoo solves exact Recency. OptimalSM solves approximate Recency with bounded absolute error $\leq D - 1$, with $D \leq W$. HistoricalMembership solves approximate Recency with bound relative error.

elaboration of a lower bound of

$$W \log \frac{1}{\varepsilon} + W \cdot \max \left\{ \log \frac{W}{D}, \log \log \frac{1}{\varepsilon} \right\} - O(W).$$

Their construction is tight up to the first two terms. Similar to the naïve solution outlined above (a sequence of width $D$ bloom filters: see Figure 2.4), OptimalSM entails partitioning the window into blocks of size $D$. Then (item-fingerprint, block-number) pairs are stored in a hash table. The block-numbers are circular and can be recycled to lower the bit allocation. This tactic can be adopted in ExactCuckoo to reduce the cost of the timestamp. The hash table employed is backyard cuckoo hashing [11], which, compared to the standard cuckoo design, achieves better memory utilization and worst-case, as opposed to expected, constant-time queries and updates. To identify expired items, a constant number of items are scanned at each update (those with an old block number are evicted). Backyard cuckoo hashing is largely a theoretical artefact, so no experimental evaluation is provided.

To summarize, we observe three types of solution: direct descendants of the Bloom filter [152]; circular array solutions [15, 104]; and the hash tables solutions [96, 111]. The hash table solutions, as well as the timing Bloom filter, provide answers to time-recency on a sliding window. A comparison with our solution, HistoricalMembership, is made in Table 2.1.

2.5 Oblivious RAM

We now pivot to an application of HistoricalMembership within the domain of oblivious storage. This subsection will develop the context for Research Question 3 and serve as a preamble for Section 5. In the latter, we illustrate how HistoricalMembership can be used to increase the performance of a whole class of oblivious storage solutions. This subsection is broken into three parts. First, we begin with a small overview of oblivious storage and its applications. Second, we discuss key assumptions and performance metrics as a way of articulating nuance within Research Question 3. Finally, we cover prior solutions and establish our contribution relative to these solutions.
Oblivious Storage

A fundamental problem in database security is how to outsource private storage requirements without leaking sensitive information. A first step is to encrypt the content of the data. However, encryption alone cannot protect against all vulnerabilities. For instance, the order in which the client accesses the outsourced data, known as the access pattern, can leak sensitive information. An illustrative example is given by Moataz et al. [107]:

observing that an investment bank has repeatedly accessed their files on a specific company may reveal that they plan to invest in that company. Crucially, there is no easy way to bound what information you might leak.

Access pattern vulnerabilities have been demonstrated in a number of domains. These include, but are not limited to, leakage through page fault patterns in secure enclaves [134, 144, 149], through SQL query patterns on encrypted databases [85, 90] and through search patterns, resulting in query recovery attacks, in searchable encryptions [31, 85].

Oblivious RAM (ORAM), pioneered by Goldreich and Ostrovsky [64], was introduced to mitigate access pattern leakage. It is a compiler that preserves the input-output behaviour of a program while producing an access pattern that is independent of the input. Essentially, an adversary observing the access pattern of a program compiled under ORAM should learn nothing (or at most a negligible amount) about the input. Thus, the access patterns produced by any two distinct inputs of the same length should be indistinguishable. By mitigating the vulnerability of access patterns, ORAM has been adopted in secure outsourced storage [139] and secure processors [1, 2]. It has also been applied in secure multi-party computation [62] and for privacy preserving group data access [70].

To maintain obliviousness, additional physical accesses are required to hide the identities (the logical addresses) of the accessed data blocks. The additional accesses occur either during a single access (online) or when the outsourced data is periodically shuffled (offline). In addition, dummy blocks are typically stored at the server to obscure sensitive information. For example, to hide the length of an array of real blocks, we can pad the array with dummies to its worst-case length. A trivial ORAM can be constructed by sequentially downloading the full database during each access. Obliviousness is achieved as the access pattern is identical for all input sequences of the same length. However, this approach incurs a linear blowup in bandwidth and is impractical on meaningfully sized databases. The aim within the literature is to minimize the bandwidth overhead without sacrificing the privacy guarantee. In contrast to the trivial solution, a bandwidth-efficient ORAM uses a combination of additional dummy accesses and periodic shuffling of the server’s contents to obfuscate the original access pattern.

Performance metrics

The bandwidth, or communication complexity, has various equivalent definitions. It is defined as either the number of data blocks retrieved per access, the relative blowup in communication vs. a non-oblivious RAM or the number of bits transferred across the channel. The latter measure is the most transparent. For example, Devedas et al., with a construction named Onion ORAM (OnionORAM), achieve a constant blowup in communication complexity, despite exchanging a large amount of metadata with the server [51]. The claim holds provided the block
size is sufficiently large and dominates the volume of communication. Thus, it might be more intuitive (or instructive) to measure communication in bits as constant communication does not imply that blocks are the only information exchanged. In addition, performance in latency sensitive applications can be impacted by the number of round-trips of communication, per single access, between the client and the server. This serves as a key metric in many practical approaches and a number of solutions have been designed specifically to reduce this cost [58, 61, 146].

Another important metric is the client memory allocation. A small client that can accommodate \( O(1) \) blocks has no information on where a targeted block is located at the server. Thus, a large amount of communication between the client and the server is required to locate the block without revealing this location to the server. Increasing private memory, to allow the client to store some metadata, helps reduce this overhead of information exchange with the server. Client memory is therefore engaged in a key trade-off with bandwidth and latency. The central tension is that we cannot increase the size of the client to a level that makes outsourcing redundant. Often, the size of the client is dictated by the application. On the one hand, secure enclaves, such as Intel SGX [7], have limited private memory, and on the other, a cloud computing service can rely on the private memory, either RAM or disk, of a desktop computer.

In the original formulation of the ORAM problem [64], the client is constrained to \( O(B) \) bits of private memory and the server is not permitted to perform computation on behalf of the client. Under this setting, a \( \Omega(\log n) \) lower bound exists for the bandwidth overhead [92]. This states that an access to a logical address requires at least a logarithmic number of physical accesses on average. The state-of-the-art solution in this setting, OptORAMa [13], matches the lower bound. However, the solution incurs a logarithmic number of round trips per access and admits an impractical concrete \(^{22} \) bandwidth cost. As we expand on below, this is largely due to the aforementioned problem constraints. Fortunately, they are unrealistic in contemporary settings and can be relaxed. In many applications, it is feasible to give the client \( \omega(B) \) bits of memory or allow the server to perform computation. For example, a number of approaches have employed private information retrieval techniques [49, 103], including fully homomorphic encryption [10, 62] and additively homomorphic encryption [37, 51], to outsource the client’s secret state and allow the server to do computation on behalf of the client and, in return, reduce the amount of communication.

Relaxing the problem constraints has led to more practical solutions, some that even circumvent the lower bound. The notion of practicality is key to our contribution.

**Definition 8** (Practical ORAM). An ORAM is called practical if it incurs a constant number of round trips of communication per access and admits a small concrete bandwidth overhead.

Our ORAM protocol, RankORAM, utilizes a larger client – we store a variation of HistoricalMembership in client memory – to achieve a practical construction. To contextualize the extent of our contribution, we now cover some prior solutions.

\(^{22}\)Here “concrete” refers to exact bandwidth cost without complexity notation.
Figure 2.10: A read(5) in a hierarchical ORAM. Level 2 (in red) is empty. The operation queries the hash table (here represented as an array padded with dummies) looking for item 5. The server returns dummy elements at levels 0 and 1 (note that a non-dummy could be returned at lower levels). The queried item is found and returned to the client at level 3. After the payload of the block is read, a rebuild occurs. T₂ is the smallest empty hash table. We then (obliviously) build T₂ on the input 5||T₀||T₁ and mark T₀ and T₁ as empty.

**ORAM solutions**

ORAM solutions can be categorized into two types: tree-based [141] and hierarchical [64]. At a high level the two approaches share some similarities. Both approaches involve distributing blocks across a sequence of levels that increase exponentially in size. Periodically the levels are rebuilt, either in portions (Tree variant) or as a whole (Hierarchical). More recently accessed data blocks are located in the lower levels (this is enforced strictly in the hierarchical variant) and less recently accessed blocks are gradually moved into the larger levels. An accessed block is always placed back in the top level and moves down the structure as it ages. Both approaches uphold the same two invariants. First, during an access the client cannot reveal which level the accessed block belonged to. In most cases, particularly without a large enough client to store relevant metadata, the client does not know, prior to the operation, which level the block resides in. Second, a block is never accessed twice in the same location between rebuilds at a given level. With these shared invariants, the fundamental differences between the two constructions include whether they impose restrictions on a block’s location within a level and how they treat the shuffling of data blocks. We cover the particularities of the two constructions, with a primary focus on the Hierarchical form, in more detail below.

Knowing where a block resides within the leveled structure can help reduce bandwidth and latency. For example, in the hierarchical ORAM, the Recency of a block will determine which level it resides in. As noted by Williams and Sion, “the main issue in constructing a single round trip ORAM is that a request for an item depends on how recently an item was accessed” [146]. However, this recency information is expensive, with a naïve implementation costing \(\omega(n)\) bits. In a plug-and-play, this cost can be reduced with HistoricalMembership and enable hierarchical ORAMs with a single round-trip.
Hierarchical ORAM

The hierarchical construction was originally proposed by Goldreich and Ostrovsky [64]. For a logical memory of $n$ blocks of size $B$ bits, the ORAM contains a hierarchy of oblivious hash tables $T_0, \ldots, T_L$, with $L = \log n$. In the words of Goldreich and Ostrovsky, the ORAM consists of “a hierarchy of buffers of different sizes, where essentially we are going to access and shuffle buffers with frequency inversely proportional to their sizes”. The hash table $T_l$ stores $2^l$ data blocks. Next to each table, a flag is stored to indicate whether the hash table is full or empty. When receiving a request to an address $x$ the ORAM operation involves both an access and rebuild phase:

1. access: Access all non-empty hash tables in order and perform a lookup for address $x$.
   If the item is found in some level $l$, perform a dummy look up in the tables $T_{l+1}, \ldots, T_L$.
   If the operation is a write, ignore the associated data and update the block with the fresh payload.

2. rebuild: Find the smallest empty hash table $T_l$ (if no such level exists then set $l = L$).
   Merge the accessed item and all of $\{T_j\}_{j \leq l}$ into $T_l$. Mark levels $T_0, \ldots, T_{l-1}$ as empty.

All the variations of Hierarchical ORAM (that do not utilise server computation) depend on the implementation of the oblivious hash table. The bandwidth overhead is determined by the two cost metrics of the underlying oblivious hashing scheme: the cost of building the hash table (offline bandwidth) and the cost of an access (online bandwidth). In most prior work this cost is dominated by the hash-table rebuilding phase.

In the original proposal by Goldreich and Ostrovsky, at level $l$, the hash table contains $2^l$ buckets of depth $\log n$ bits. When accessing a bucket obliviously, a linear scan is performed. As a table look up requires $O(\log n)$ accesses, to scan the bucket, the online bandwidth blowup is $O(\log^2 n)$. The rebuild phase makes use of a primitive, named oblivious sorting, that can sort an array without revealing, through the access pattern, the ordering of the sorted array. During the rebuild phase, all elements, including dummies, from levels $\{T_j\}_{j \leq l}$ are written into a temporary array. The temporary array is obliviously sorted and scanned to remove duplicates (an element may reside at multiple levels). Random tags are then assigned to elements. Through a clever use of oblivious sorts and scans, dummy elements are also tagged such that there are exactly $W$ copies of each tag. When sorted according to the tags the array is a hash table with buckets of equal depth. As oblivious sorting is expensive, rebuilding incurs an amortized bandwidth cost of $O(\log^3 n)$ blocks.

Subsequent improvements were achieved by changing the hashing primitive to an oblivious Cuckoo hashing scheme [32, 69, 70, 125]. To ensure privacy, an oblivious cuckoo hash table is built with a stash to store items that cannot be assigned a slot in the table [69]. However, the search cost of the cuckoo table, typically constant time, is dominated by the size of the stash, which is $O(\log n)$ with high probability. The online bandwidth overhead obtained by searching stashes can be reduced by combining the stashes of all levels into a single stash that

---

23 A security vulnerability is introduced by rebuilding the table with fresh randomness in the event that not all items are assigned a unique location by the hash functions. Recall (see Subsection 2.1) that, in this instance, non-oblivious hashing would repeatedly rebuild until a successful allocation.
is logarithmic in size \([70]\). Consequently, the online bandwidth cost, composed of a scan of the combined stash and \(L = \Theta(\log n)\) cuckoo probes, is \(\Theta(\log n)\) blocks. Due to the super-linear cost of constructing an oblivious hash table, owing largely to the use of oblivious sorting, the amortized bandwidth overhead (dominated by offline bandwidth) is reduced to the non-optimal \(\Theta(\log^2 n / \log \log n)\) blocks.

Chan et al. [32] present a simple two-tier hashing scheme that establishes an ORAM with the same bandwidth overhead as previous constructions based on (the more complex) oblivious Cuckoo hashing. Under the observation that, in traditional approaches, the bandwidth overhead is dominated by the rebuild phase, the two-tier hashing scheme permits a more expensive lookup query in exchange for a reduced rebuild cost.

A return to oblivious Cuckoo hashing, with a new oblivious construction algorithm, was made by Patel et al. in their scheme, PanORAMa [122]. In contrast to prior methods, the construction algorithm assumes that the input is randomly shuffled. This assumption is justified by the observation that the untouched elements in the lower levels (which are to be merged) appear to be distributed uniformly at random. Consequently, the construction algorithm can dispense with expensive oblivious sorting and reduce the offline bandwidth cost when compared to prior approaches. This idea is extended by Asharov et. al, with OptORAMa [13], to achieve optimality. In subsequent work, OptORAMa was modified to give worst-case logarithmic bandwidth overhead [14]. Similar to Goodrich et al. [70], PanORAMa and OptORAMa combine the cuckoo stashes from each level into a single hash table. More recently, Hemenway et al. demonstrate that combining stashes introduces an additional security vulnerability [76]. As the authors provide a patch that applies to all Hierarchical ORAMs with a combined stash, their result does not undermine the preceding contributions.

None of the preceding solutions satisfy our definition of practicality. They incur high concrete bandwidth costs and observe a logarithmic number of round-trips per access. Notably, the solutions are constrained by \(\Theta(B)\) clients and the foreclosure of server side computation. We now cover some solutions that relax these constraints towards more practical outcomes.

<table>
<thead>
<tr>
<th></th>
<th>client storage</th>
<th>bandwidth</th>
<th>hash table</th>
<th>single round-trip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square Root</td>
<td>(\Theta(B))</td>
<td>(\Theta(\sqrt{n}))</td>
<td>permuted array</td>
<td></td>
</tr>
<tr>
<td>Hierarchical</td>
<td>(\Theta(B))</td>
<td>(\Theta(\log^3 n))</td>
<td>balls-in-bins</td>
<td></td>
</tr>
<tr>
<td>Goodriech</td>
<td>(\Theta(B))</td>
<td>(\Theta(\log^2 n / \log \log n))</td>
<td>Cuckoo</td>
<td></td>
</tr>
<tr>
<td>Chan [32]</td>
<td>(\Theta(B))</td>
<td>(\Theta(\log^2 n / \log \log n))</td>
<td>Two tier</td>
<td></td>
</tr>
<tr>
<td>PanORAMa [122]</td>
<td>(\Theta(B))</td>
<td>(\Theta(\log n \cdot \log \log n))</td>
<td>Cuckoo</td>
<td></td>
</tr>
<tr>
<td>OptORAMa [13]</td>
<td>(\Theta(B))</td>
<td>(\Theta(\log n))</td>
<td>Cuckoo</td>
<td></td>
</tr>
<tr>
<td>RankORAM</td>
<td>(\Theta(n))</td>
<td>(4 \log n)</td>
<td>permuted array</td>
<td>✓</td>
</tr>
</tbody>
</table>
Practical ORAM

Initial solutions to ORAM are considered to be theoretical artefacts. Since 2011, a portion of the literature has aimed at constructions suitable for industry [130, 139], by reducing the concrete bandwidth and the number of round-trips between the client and the server. This is achieved, as suggested above, by allowing for larger clients or admitting server-side computation to circumvent the lowerbound.

Before we proceed with an overview of this line of work, we provide some additional details regarding the tree-based scheme PathORAM, introduced by Stefanov et al. [141], as it constitutes the foundation of a number of practical solutions. PathORAM places data blocks in a binary tree of depth $\lceil \log n \rceil$. Similar to Hierarchical ORAM, blocks are therefore distributed across a sequence of levels, increasing exponentially in size. The nodes in the tree are buckets that can store a constant number of blocks. Each block is randomly assigned to a leaf and the leaf assignments are stored in a position map. The construction enforces the invariant that a block must reside on the root-to-leaf path prescribed by the leaf assignment. The position map requires $\Theta(n \log n)$ bits of private memory. To achieve a low-memory solution, for a large enough block size ($B = \Omega(\log^2 n)$ bits), the position map is stored in a smaller ORAM (by packing the blocks). This can be repeated recursively until the position map for the smallest ORAM meets the client memory requirements. We omit details of the access and rebuild phases as they are not required for our purpose.

Constant number of round-trips With their contribution SRORAM, Williams and Sion were the first to construct a single-round trip ORAM [146]. Following the hierarchical framework, the authors observed that an access is interactive. As the client does not know which level a block belongs to, the levels are queried sequentially until the target block is found. This requires a round trip per level. Consequently, the authors aimed to remove the interactive component of an access. To achieve this, they built on a previous approach [147] that separates membership testing from block storage. Each oblivious dictionary (or level) is coupled with an encrypted BloomFilter, which represents the membership at each level. Utilizing this separate mechanism for membership testing, during an access, SRORAM builds a layered branching program with paths that depend on the location of the accessed item. The server queries each BloomFilter and the output is used to unlock the next step in the correct path through the query object. The path reveals which blocks to return to the client. Thus, the branching program constitutes a non-interactive query object.

SRORAM is a small memory solution; the client can store up to a constant number of blocks and the query object. Further, due to the cost of rebuilding the oblivious hash tables, the offline round-trips and concrete bandwidth are still high. The key innovation is the idea of constructing a non-interactive query by allowing server-side computation.

Fletcher et al. improve on this result by constructing an ORAM with constant communication complexity on a single round trip [58]. The design, named BucketORAM, entails a hybrid scheme that works by splitting each level of a hierarchical ORAM into buckets of fixed width, in effect augmenting the structure with a tree shape. The hybrid approach has two advantages. First, the scheme inherits the efficiency of tree-based algorithms, where rebuilds now occur on buckets and not whole levels. Second, following the hierarchical structure, the authors can
Table 2.3: Performance of the class of practical ORAMs. Of the listed solutions, only PathORAM fails to achieve a constant number of round-trips per access, observing a cost of $\mathcal{O}(\log n)$ round-trips. The low-memory solutions require server-side computation to achieve a single online round-trip of communication.

Larger clients and low concrete bandwidth The single-round trip ORAMs succeed by allowing server-side computation. However, SRORAM suffers from expensive oblivious sorts on small clients and TWORAM requires recursion to store the position maps, which increases the bandwidth overhead. These bandwidth costs can be reduced by permitting a larger client.

Stefanov et al. were the first to consider larger clients under the observation that for contemporary settings there are many applications in which clients can afford more than $\mathcal{O}(B)$ bits of private memory [140]. The authors, with the PartitionORAM, expand the client to $\mathcal{O}(\sqrt{n} \cdot B + n \log n)$ bits in a bid to reduce the cost of oblivious sorts. The key innovation is the partitioning technique, which partitions a single ORAM into multiple (Hierarchical) ORAMs of size $\sqrt{n}$. Therefore, given the budget of client memory, hash table rebuilds can be performed without interaction with the server in $\mathcal{O}(\sqrt{n} \cdot B)$ bits, removing the need for expensive oblivious sorting to perform the shuffles. Items are randomly assigned to a point of

---

24We cover this in more detail below, but oblivious sorting in $\mathcal{O}(B)$ bits is very expensive in practice. The state-of-the-art techniques (which were created after the release of PartitionORAM) leverage $\mathcal{O}(\sqrt{n} \cdot B)$ bit clients. In Chapter 6, we construct an oblivious permutation that requires $2n + 2B + o(n)$ bits and reduces the dependence on the block size.
the partition and, avoiding recursion, the position map is stored at the client. To demonstrate security, the authors verify that the sequence of partitions accessed does not leak any sensitive information.

Dautrich et al., with the aptly named BurstORAM, apply some optimizations and novel techniques to PartitionORAM to minimize online bandwidth and delay offline block shuffling to idle periods [50]. We detail both optimizations as both can be adopted in our own construction. The first is level caching. As the lower (and smaller) levels of an ORAM are accessed and rebuilt frequently, continual communication is required to maintain them. Thus, the authors, leveraging the available client space, place the smaller levels at the client. The second optimization requires the server to XOR batches of blocks before sending them to the client. This reduces the bandwidth overhead of each batch to that of a single block. To support this practice, additional metadata is required and the client stores, in the position map, not only the partition number for each block but also its level number within the partition. Thus, if a targeted block resides at some level \( l' \), the client can retrieve dummy data blocks for all levels below \( l' \) in addition to those above \( l' \). Recall that in the standard hierarchical ORAM the client has to perform membership tests at these levels and does not know the identity of the returned blocks. Consequently (ignoring for a simplicity that some levels might be empty), if the server returns the compressed batch \( B_S = B_l \oplus \cdots \oplus B_L \), the client can decode \( B_{l'} \) by first constructing

\[
B_C = B_l \oplus \cdots \oplus B_{l'-1} \oplus B_{l'+1} \oplus \cdots \oplus B_L
\]

and performing

\[
B_{l'} = B_S \oplus B_C.
\]

This technique can be applied in a number of ORAMs on the condition that the client stores the relevant metadata.

Ren et al., with RingORAM, apply the optimizations of Burst ORAM to a tree-based construction [130]. The position map now stores both the level and the leaf number of each data block. In addition, to retrieve a single block per tree-node, RingORAM permutes the buckets and stores the permutation metadata at the client. As a result, the client knows where each real and dummy data block resides within the tree and the XOR technique can be applied. With the position map stored at the client, RingORAM avoids the bandwidth overhead associated with recursion and provides significant improvement to the concrete bandwidth cost when compared to its predecessor PathORAM.

Chen et al. [37] combine RingORAM with another tree-based solution, Onion ORAM [51]. The latter utilizes additively homomorphic encryption\(^{25}\) and server-side computation to reduce the bandwidth overhead. To improve the rebuild efficiency, the authors introduce a homomorphic permutation algorithm to allow shuffling at the server without interaction with the client.

What these solutions have in common is a \( \Theta(n \log n) \)-bit client memory allocation for position maps. In contrast, RankORAM uses compressed Recency to capture the relevant metadata. Therefore, as demonstrated in Table 2.3, we are able to reduce the client memory allocation

\(^{25}\)Onion ORAM is an improvement on a prior work that relies on the more computationally intensive fully homomorphic encryption [103].
The Recency Problem and its Applications  William L. Holland

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Client Storage</th>
<th># of I/0s</th>
<th>Randomized</th>
<th>Oblivious</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge sort</td>
<td>(O(\log n) + 2B)</td>
<td>(4n\log n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitonic sort [112]</td>
<td>(O(\log n) + 2B)</td>
<td>(2n\log^2 n)</td>
<td>(\checkmark)</td>
<td></td>
</tr>
<tr>
<td>Melbourne Shuffle [117]</td>
<td>(O(B \cdot \sqrt{n}))</td>
<td>(O(n))</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>CacheShuffle [123]</td>
<td>(O(B \cdot S))</td>
<td>(O(n \log S n))</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>Original Waksman [22, 143]</td>
<td>(O(n \log n) + 2B)</td>
<td>(4n \log n - 3.6n)</td>
<td>(\checkmark)</td>
<td></td>
</tr>
<tr>
<td>Bucket ORP [12]</td>
<td>(O(Z \cdot B))</td>
<td>(8n \log (n/Z))†</td>
<td>(\checkmark)</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>Waksman</td>
<td>(2n + o(n) + 2B)</td>
<td>(4n \log n - 3.6n)</td>
<td></td>
<td>(\checkmark)</td>
</tr>
</tbody>
</table>

Table 2.4: Summary of properties of client-server algorithms that permute an array of \(n\) elements each comprising \(B\) bits. Randomized algorithms [117, 123] may fail to perform the required functionality. Merge sort is referenced as a non-secure benchmark. For CacheShuffle, \(S = \omega(\log n)\). For BucketORP, the authors recommend \(Z = 512\). † BucketORP incurs \(4n \log n\) I/Os when \(n/Z\) is a power of 2.

against state-of-the-art practical solutions. This is a significant step towards the use of practical solutions within memory constrained environments such as secure enclaves.

2.6 Oblivious Permutation

Oblivious permutation is a key primitive in almost all ORAM schemes. RankORAM uses oblivious permutation as a black box and, to supplement this work, we developed a new oblivious permutation algorithm, WaksmanOP, that explores a similar trade-off between client-memory and bandwidth. Although we do not adopt the solution in RankORAM, it obtains the same client memory bound and has scope, by allowing server-side computation, to be incorporated to reduce the bandwidth overhead. We expand on this possibility in Chapter 7 and it constitutes one of the key directions of future work.

This subsection can be viewed as a preamble to Section 6 and the core literature to which Research Question 4 belongs. The purpose of this subsection is to cover prior solutions to oblivious permutation. In the previous subsection, we encountered the notion of oblivious algorithms through the sorting primitive. Oblivious algorithms satisfy two objectives; (1) they implement a specified functionality; and (2) they generate an access pattern that does not reveal anything about the algorithm’s private input. For example, the permutation functionality is used to permute an array according to some input permutation. Thus, oblivious permutation, by definition, permutes an array without revealing the input permutation through its access pattern. This ability to shuffle the contents of the server without revealing the order of the shuffling to an adversary allows the client to obscure the locations of data blocks at the server. Therefore, oblivious permutation is a key primitive in many ORAM schemes.

There are two families of state-of-the-art oblivious permutation: those based on sorting networks [4, 20, 68, 112], and those based on shuffling [117, 123]. Our exposition begins with an overview of these two families of solution.

40
Sorting and switching networks for oblivious permutation

Early solutions to oblivious permutation involve assigning distinct labels to elements and applying a sorting network. The labels are assigned by the input permutation, $\pi$. Sorting networks are composed of several levels of carefully laid out comparators that gradually transform the input to a sorted output. The network layout depends only on the input size. Thus, sorting networks are oblivious. A comparator takes two elements and rearranges them based on their relative order. The output of each comparator serves as an input into comparators of the next level. Since they can be evaluated in a sequence, client memory is small: the algorithm reads the pair of elements that correspond to the current comparator and writes them back to the server according to their relative order. The bandwidth overhead depends on the number of comparators, which we refer to as the network size. With optimal theoretical sorting networks, such as AKS [4] or Zig-Zag [68], the permutations can complete in $O(n \log n)$ I/Os (the number of blocks exchanged). However, the asymptotic bounds hide large constant factors (e.g., 19,600 for Zig-Zag [68]) and the non-optimal bitonic sort [112], with $n \log^2 n$ comparators, is considered a practical alternative. For example, AKS or Zig-Zag become feasible alternatives only for $n \geq 2^{1900}$.

Additionally, oblivious random permutations can be achieved through randomized switching networks [47], but are not considered practical due to their size. The latter work of Czumaj is significant as small deterministic switching networks, which perform all possible permutations, fail when the switches are set uniformly and independently at random: the output permutation of the randomly configured network is not uniform. This observation has largely prevented their adoption in both theory and practice.

Oblivious Shuffles

The Melbourne shuffle is an oblivious permutation that does not rely on oblivious sort [117]. The authors were the first to exploit larger client memory, relative to prior work, in pursuit of lower I/O and bandwidth costs. In contrast to algorithms based on sorting networks, the Melbourne Shuffle is randomized and may fail with small probability without affecting its security. In terms of performance, it is the state-of-the-art for I/O efficiency in settings where the client is permitted $o(nB)$ bits of memory. As the algorithm requires $\Omega(\sqrt{nB})$ bits of client memory, it is sensitive to the block size and can become untenable for large values of $B$. Patel et al. [123] show that the Melbourne shuffle can be adapted to client memory of size $O(BS)$ while incurring an $O(n \log S)$ I/O cost, for $S = o(\log n)$.

A recent work, named Bucket ORP [12] and based on the bucket ORAM [12], provides a slightly different functionality: it randomly permutes the array to some unspecified permutation. The algorithm assigns elements to random bins and routes elements into the bins through a butterfly network. Subsequently, the bins are individually permuted and then concatenated together. Bucket ORP achieves a competitive balance between bandwidth and client memory efficiency and constitutes a key benchmark. Bucket ORP does not achieve perfect oblivious-

\(^{26}\)The notion of optimal here assumes that the keys are sufficiently large. This is always the case during oblivious random permutation as we are sorting the key set $\{0, 1, \ldots, n - 1\}$. In contrast, one-bit keys can be obliviously sorted in $o(n \log n)$ time [95].
ness. Notably, when it shuffles the input towards some unspecified permutation, not all \( n! \) permutations are achievable. Though the set of impossible permutations is negligible and is governed by a security parameter, it consists of permutations that follow a specific structure such as the identity permutation or the reverse permutation. As a result it cannot be used if perfect obliviousness is required.

The trade-off between client memory and bandwidth

A number of the large client ORAM solutions require \( \Omega(\sqrt{n} \cdot B + n \log n) \) bits of private memory. The justification for storing a position map – the \( \Theta(n \log n) \) bit component of the bound – stems from the observation that, for some applications, the block size is large, that is, \( \omega(\log n) \) bits. Indeed, some authors [50] note that, particularly for cloud applications, the size of the position map is negligible when compared to \( \Theta(\sqrt{n} \cdot B) \) bits. This has implications for the shuffle family of oblivious permutations. The shuffle-based solutions, extend the client to \( \Theta(\sqrt{n} \cdot B) \) bits to reduce the bandwidth cost when compared to small memory solutions based on sorting networks. However, for applications where \( n \log n = o(\sqrt{n} \cdot B) \), the memory of allocation of the shuffle family is highly sensitive to the block size. This can prevent the application of efficient oblivious permutation in memory constrained settings. In Section 6, we address this shortcoming and construct an efficient oblivious permutation that is not sensitive to the block size. Thus, our contribution, WaksmanOP, provides a new point in the trade-off between client memory and bandwidth within this two-dimensional parameter space.

Our key innovation is a routing algorithm for the Waksman network [143] that requires only \( 2n + o(n) \) bits of client memory, a logarithmic improvement over prior approaches. Significantly, the Waksman network supports efficient concrete bandwidth performance, but had previously been ignored due to its large memory requirement. Table 2.4 summarizes the asymptotic performance of approaches for oblivious permutation, including our algorithm, WaksmanOP.

Applications of the Waksman network

Prior to this work, the Waksman network has been adopted in the setting of multi-party secure computation (MPC). Zahur et al. [151] revisit the square-root oblivious RAM of Goldreich and Ostrovsky [64] in the setting of MPC and obtain performance improvements by replacing the oblivious sorting primitive with a Waksman switching network. The authors demonstrate performance improvements across a concrete parameter space. Zahur et al. [151] use the Waksman network in a model that is significantly different from ours. They do not have the same constraints on client storage and can configure the Waksman network with the algorithm outlined in the original paper [143], which is to say, with more client memory than is permitted here, that is, with \( \omega(n) \) bits.

The Waksman network has seen application ORAM constructions. The key innovation of Onion Ring ORAM entails an improved homomorphic permutation algorithm [37]. Homomorphic permutations were introduced in Onion ORAM to reduce the cost of the eviction phase [51]. The algorithm improved algorithm homomorphically evaluates a Waksman permutation network. Although similar constraints on client memory pertain here, the construction only
ever operates on small blocks. Moreover, the algorithm is executed at the server.

Finally, routing networks, including the Waksman network, have been used to verify the integrity of memory accesses in outsourced computation [23, 24]. This includes an implementation of the Waksman network [126] for verifiable computation, where the server employs the network to prove the correctness of the computation to the client.

**Waksman for Oblivious permutation**

The Waksman network can route an input of $n$ elements to any permutation using $n \log n - n$ switches. Hence, it provides superior bandwidth efficiency against oblivious permutation algorithms based on practical sorting networks with $n \log^2 n$ comparators. However, despite this promise, the Waksman network has not previously been adopted for oblivious permutation. This is largely due to the fact that one requires a *global view of the network* in order to set the switches according to an input permutation. The burden of configuring and storing the switch settings, along with the consequences for algorithm performance, need to be managed by the client. Note that sorting networks do not suffer from this restriction since the “setting” of each comparator depends only on the relative order of the input elements (e.g., determined using the elements’ permutation labels). The goal of our work is to rescue the Waksman network from this fate.

This concludes our literature review. The first component of the review covered prior solutions to recency queries and identified precedents for constructing efficient recency data structures. To recapitulate, recency can be prioritized through an appropriate streaming model. Further, temporal queries can be supported by dividing the stream into disjoint intervals and installing time-indifferent data structures on each of the intervals. We build on these ideas in Chapter 4 to support time-recency queries. The second component addresses the research questions more directly. We covered the core background to the substantive chapters and situated each research question within its own literature.
Chapter 3

Succinct List Indexing

In chronological fashion, we begin with a response to Research Question 1. The work is presented as a manuscript under review\textsuperscript{1} at ISAAC 2022. It is the only sole-authored paper in the thesis. The data structure that we exhibit supports exact item-recency queries (Definition 2) in small memory.

Prior solutions to item-recency are covered in Subsection 2.2 under the heading “Calculation of stack distance”. A common approach to achieving fast updates and queries involves augmenting a linked-list with some auxiliary structure that supports fast navigation. The Markers algorithm (Figure 2.5) is a good example. Further improvements are enabled by adding a hash table to allow nodes in the list to be accessed in constant time. The BennettKruskal algorithm (Figure 2.6) testifies to these runtime improvements. However, these structures are space-inefficient, with linked-lists requiring an abundance of pointers and additional hash tables multiplying the memory cost. This can prevent their range of application. For example, Ha et al., in the setting of flash storage, preference an alternate structure that provides crude approximations with no formal accuracy guarantee [72].

Our solution, the Princess List, adopts the more general abstraction of the indexed list. Standard indexed list solutions (Subsection 2.3) adopt a similar form to BennettKruskal – a linked-list, an auxiliary tree structure and a hash table – and, therefore, commit to a comparable memory allocation. In contrast, the Princess List is the first succinct representation of an indexed list with optimal update and query times. To provide some perspective on the significance of the result, a memory optimal encoding of an ordered set $S \subseteq [m]$, of size $n$, requires

$$\left\lceil \log \left(\frac{m}{n}\right) \right\rceil = n(\log m - O(1))$$

bits. Through a trivial encoding, by placing the elements in the correct order in an array, we can get close to this bound with $n \log m$ bits. However, updates take linear time. The Princess List, encodes the ordered set in $n(\log m + o(\log n))$ bits and supports all operations optimally in $O(\log n / \log \log n)$ time. Thus, in contrast to the array, we require an additional sub-logarithmic number of bits per item while supporting optimal time operations.

Additionally, our work overlaps with the literature on succinct dynamic strings. A key challenge with dynamic strings is dealing with large alphabets and state-of-the-art solutions

\textsuperscript{1}Submitted June 27 2022.
admit a memory allocation that is super-linear in the alphabet size. Our result illustrates a method to maintain a succinct representation in a setting, such as those that require stacks, where the alphabet size exceeds that of the list (or string).

The strength of this paper is the result. The work supporting the latter entails algorithmic components and analysis which are, by now, fairly standard. Further, a number of (more complicated) prior results are used out-of-the-box. However, they are integrated together in a novel way and this doesn’t detract from the outcome. Part of this simplicity has further implications. The high level algorithm, that forms the basis for the construction that produces Theorem 1, is simple and can be easily implemented. We have an implementation, along with some baselines, and some additional experimental work would add value to this piece. The presented manuscript does not report these results as the work is targeted at a theoretical venue.
**Succinct List Indexing in Optimal Time**

**Abstract**

An indexed list supports (efficient) access to both the offsets and the items of an arbitrarily ordered set under the effect of insertions and deletions. Existing solutions are engaged in a space-time trade-off. On the one hand, time efficient solutions are composed as a package of data structures: a linked-list, a hash table and a tree-type structure to support indexing. This arrangement observes a memory commitment that is outside the information theoretic lower bound (for ordered sets) by a factor of 12. On the other hand, the memory lower bound can be satisfied, up to an additive lower order term, trivially with an array. However, operations incur time costs proportional to the length of the array.

We revisit the list indexing problem by attempting to balance the competing demands of space and time efficiency. We prepare the first succinct indexed list that supports efficient query and update operations. To implement an ordered set of size \( n \), drawn from the universe \( \{1, \ldots, m\} \), the solution occupies \( n(\log m + o(\log n)) \) bits (with high probability) and admits all operations optimally in \( \mathcal{O}(\log n / \log \log n) \) time.

**2012 ACM Subject Classification** Theory of computation → Data structures design and analysis

**Keywords and phrases** Succinct Data Structures, Lists, Dynamic Data Structures

**Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

**Acknowledgements** This work was supported by
1 Introduction

List indexing is a popular problem in the data structures literature from the previous century. The problem demands a representation of a list \( L \) that supports the following operations:

- **insert**\( (L, x, y) \): insert element \( y \) at the index succeeding element \( x \).
- **delete**\( (L, x) \): delete element \( x \) from the list.
- **index**\( (L, i) \): return the element at index \( i \) in the list.
- **position**\( (L, x) \): return the index of element \( x \) in the list.

A list that supports these operations is named an **indexed list**. Notably, the **position** query is a key inbuilt function in programming languages such as Python and Java. For a list encoding an arbitrarily ordered set \( L \subseteq [m] \) of size \( n \), a lower bound of (amortized) \( \Omega(\log n / \log \log n) \) time per operation is due to Fredman and Saks [4]. The bound is assembled in the cell probe model. Matching upper bounds, in the word RAM model, are provided by Dietz [3]. As is noted by Andersson [1], the solution requires item pointers be provided to the operations. Thus, the data structure of Dietz appears in three components; a hash-table to retrieve item pointers; the underlying linked-list; and the structure to support (fast) indexing. This configuration requires \( \mathcal{O}(n \log m) \) bits of space with a constant factor around 12.

A data structure, supporting a particular query (or set of queries), is **succinct** if it accommodates a memory commitment “close to” the information-theoretic lower bound and admits the prescribed query operation(s) “efficiently”. Formally, if a minimum of \( B \) bits are required, in the information-theoretic sense, to store the data, a succinct data structure occupies \( B + o(B) \) bits of memory. Despite the affluent and diverse state of the field of succinct data structures [5, 9, 11, 14, 18], a succinct representation of an indexed list does not exist and we pursue the problem of finding such a representation.

The information-theoretic lower bound for encoding an ordered set is

\[
B(m, n) = \log \left( \frac{m!}{(m-n)!} \right) = n(\log m - O(1))
\]  

bits. A low-memory solution can be constructed by positioning the items consecutively in an array that has \( \log m \) bit cells. The per-item cost of the encoding is \( O(1) \) bits above the information-theoretic lower bound. However, updates to the list and **position** queries are slow, requiring \( O(n) \) time. This observation leads to an interesting question: can a list be encoded at close to \( \log m \) bits per item and support dynamic indexing operations efficiently?

In response to this question, we first present a simple data structure, named the Princess List (PL), that significantly improves the update and query efficiency of the canonical linked-list at a small space overhead. We then detail an optimized implementation of the Princess List (named PL+) that occupies \( n(\log m + o(\log n)) \) bits of space, with high probability, and admits **optimal** query and update times, matching the prior state-of-the-art. The structure pays, per-item, a sublogarithmic number of bits above the information-theoretic lower bound for ordered sets. In the appendix we demonstrate an alternate solution that achieves the memory commitment with probability 1, but with update times close to optimal with high probability.

---

1 We assume that the update sequence does not contain any repetitions. We elaborate on the implication of including repetitions in Appendix A.
The Princess List  At a high level, the PL is a divide-and-conquer approach, where the
list items are partitioned, via a hash function, into a collection of disjoint sublists. If the hash
function that performs the partition maps on to a small range \( [\sigma] = \{0, 1, \ldots, \sigma - 1\} \), the list
can be mapped to a random string on the alphabet \([\sigma]\). Each character \( c \in [\sigma] \) represents
an order-preserved sublist \( L_c \) on the set \( S_c = \{ x \in L \mid h(x) = c \} \) and each occurrence of a
character represents a unique item. The PL is comprised of the random string \( h(L) \) and the
sublists \( L_c \).

An operation entails identifying, through the random string, the correct (and unique)
sublist to update or query, followed by an execution of the operation on the small sublist.
The random string maintains the inter-order between the sublists and, consequently, allows
us to relate each subproblem to the superproblem. Thus, the dynamic string negotiates the
divide-and-conquer strategy and, simultaneously, maintains the order of the full list. In other
words, the string articulates both how we branch into subproblems and how we merge the
sublists back together. With this set-up, queries are fast, comprising a constant number of
queries on the string and a linear scan on a subproblem.

The high level structure, composed of the random string and the sublists, introduces
three key challenges. First, as each item is represented by both a string character and
an item identifier in the sublist, the representation of an item in each component must
be concise. Item identifiers are expensive, requiring \( \log m \) bits, and we exploit a random
permutation of the universe to partition each identifier into both its hash character and a
small unique identifier in the assigned sublist. This is a technique known as the quotient filter
[17]. Consequently, the total information needed for both the reduced small-alphabet string
and the sublists is concise. Second, we need to maintain all the component data structures
in a compact form. To store the small alphabet string we utilize an existing solution for
succinct dynamic strings [15]. To reduce the size of the bit allocation for the sublists, we use
a linked-list representation with \( \omega(1) \) items packed into each node. This allows the per-item
cost of the pointers to be sublogarithmic. Third, as we are required to perform linear scans
on the sublists, we don’t want the cardinality of any sublist to grow too large. To mitigate
the impact of a large sublist, we introduce a threshold and store any sublist that exceeds the
threshold under the non-succinct but time-optimal solution of Dietz [3].

1.1 Contribution and outline

We present the first succinct indexed list with update time and query time performance
identical to the state-of-the-art. We first introduce a simple and novel solution to list
indexing (the PL) that lowers the traversal cost on a linked-list (§3). We then curate an
optimized instance of our simple solution (the PL+) that achieves the attributes available in
the following theorem (§4).

Theorem 1. For any constant \( \gamma > 0 \), an ordered set of \( n \) items, drawn from the universe
\([m]\), where \( m = \text{poly}(n) \), can be stored in \( n(\log m + o(\log n)) \) bits with probability \( 1 - \Theta(1/n^\gamma) \)
and support index and position in \( \Theta(\log n / \log \log n) \) time and insert and delete in
\( \Theta(\log n / \log \log n) \) amortized time.

This constitutes our main result and contribution. The significance here is that our data
structure performs all operations optimal in run time – equal to the prior state-of-the-art –
while spending close to \( \log m \) bits per item. As a comparison, we achieve better asymptotic
performance than balanced binary search trees for updates and access queries with additional
support for indexing on ordered sets and a succinct representation (over ordered sets) with
high probability. In addition, to obtain a succinct representation with probability one, we
demonstrate a modified construction (detailed in the appendix) that secures the following properties.

Theorem 2. For any constant $\gamma > 0$, an ordered set of $n$ items, drawn from the universe $[m]$, where $m = \text{poly}(n)$, can be stored in $n(\log m + o(\log n))$ bits and support index in $O(\log n / \log \log n)$ time, position in $O(\log n)$ time with probability $1 - O(1/n^\gamma)$ and any sequence of $O(n)$ updates (insert or delete) takes $O(n \log n)$ time with probability $1 - O(1/n^\gamma)$.

One constraint for the data structure is that the universe size is a polynomial in the problem size. Therefore, as currently stated, our result will not hold on small problem instances. However, as small problem sizes are less interesting, this is only a minor concern. In these instances we can afford to store a non-optimal representation and, indeed, this is likely preferable. The constraint is a condition of our hash family [20]. Other succinct data structures, such as Backyard Cuckoo Hashing [2], which utilize the latter result, also inherit this condition.

1.2 The rank-select problem

List indexing has proximity to the well known rank-select problem [6, 7]. For the latter, a solution constitutes a succinct representation of a sequence $C$ (where repetitions are allowed), drawn from a universe (or alphabet) $\Sigma$, that supports the following operations.

- $C[i]$: return the character at index $i$.
- $\text{rank}_b(C, i)$: given $i \in \{0, \ldots, n-1\}$ and $b \in \Sigma$ return $|\{j \in \{0, \ldots, i\} \mid C[j] = b\}|$, i.e., the number of occurrences of character $b$ in the subsequence $C[0 \ldots i]$.
- $\text{select}_b(C, j)$: given $j \in \{0, \ldots, n-1\}$ and $b \in \Sigma$ return $\min\{x \mid \text{rank}_b(C, x) = j\}$, i.e., the index of the $j^\text{th}$ occurrence of $b$ in $C$.

The select operation is equivalent to the position query and an access $C[i]$ is exactly the index query. Thus, a compressed rank-select data structure for dynamic sequences [10, 13, 15, 16] would act as a solution to the list indexing problem. The two problems are very close and, indeed, the rank-select problem inherits the time lower bounds of the list indexing problem. The main difference between the two problems is the assumption about the universe size. For the rank-select problem it is reasonable to focus on texts drawn from a small universe of characters. However, the solutions become intractable for problems on larger universes and, in particular, where repetitions do not occur. For example, with $m = |\Sigma|$, a state-of-the-art solution by Navarro and Nekrich has a memory allocation with a redundancy term of $O(m \log n)$ [15]. In the list indexing problem, where $m \geq n$, this leads to a non-succinct representation. Further, a solution by Munro and Nekrich [13] that supports “arbitrarily large alphabets” meets a similar fate. While the memory allocation is quoted as $H_k(C) + o(n \log m)$ bits$^2$, it appears to require (implicitly) that $m \leq n$. The issue is that an auxiliary data structure is stored for each character in $\Sigma$. Resolving this issue is not as simple as assigning 0 bits for non-occurring characters as this introduces the need for a search structure. Even a state-of-the-art dynamic succinct dictionary [2] would push the memory allocation over a succinct allowance. A succinct indexed list remedies this restriction on the universe size and is an open problem that we address here.

To help distinguish our problem from this influential strain of prior work, it is best to think of an indexed list as a dictionary. In this context, it would make little sense to use a

$^2$ $H_k(S)$ is the $k^\text{th}$ order empirical entropy of the string $S$. This term is somewhat redundant in our setting; as repetitions are not allowed, the empirical entropy is high.
compressed sequence as a solution to the problem. Before we progress with the exposition, we introduce some background around strings, the hashing scheme we engage and the existing solutions to list indexing.

2 Background

We proceed in the (unit cost) word RAM model of computation, with word size $w = \Theta(\log m)$. Consequently, items from the universe can be stored in $\mathcal{O}(1)$ machine words and bitwise operations on words can be performed in constant time. We use the superscript notation $\text{index}^M$, $\text{position}^M$, $\text{insert}^M$, $\text{delete}^M$ to refer (unambiguously) to query and update algorithms on a list under the representation of the data structure $M$. We drop the superscript $M$ and the argument $\mathcal{L}$ when both are obvious from the context.

2.1 Strings

In addition to the $\text{rank}$ and $\text{select}$ queries, a dynamic string supports the following update operations:

- $\text{insertSt}(C, b, i)$: insert character $b \in \Sigma$ at index $i$ in string $C$.
- $\text{deleteSt}(C, i)$: delete character at index $i$ in string $C$.

The fundamental dynamic string implementation is the wavelet tree. It emerged in the context of text compression [8], as a tool for compressing suffix arrays, and has since seen application in a diverse range of problems. The power of the wavelet tree rests in its capacity to support both the compression of strings and fast query and update operations. The state-of-the-art wavelet tree is advanced by Navarro and Nekrich and acknowledges the following properties.

$\blacktriangleright$ Lemma 3 ([15]). For $\varphi \in (0, 1)$, a dynamic string of length $n$ on an alphabet of size $\sigma$ can be stored in $n \log \sigma + \mathcal{O}(n \log \sigma/\log^{1-\varphi} n) + \mathcal{O}(\sigma \log n)$ bits and support the queries $\text{rank}$ and $\text{select}$ in $\mathcal{O}(\varphi^{-2} \log n/\log \log n)$ time and $\text{insertSt}$ and $\text{deleteSt}$ in $\mathcal{O}(\varphi^{-2} \log n/\log \log n)$ amortized time.

2.2 $k$-wise independent hashing

The division of the list into disjoint sublists is coordinated by a random source. Our compact representation utilizes existing hash families with limited independence, small description and constant time evaluation.

A family of functions is $k$-wise independent if, for a function $f[m] \rightarrow [r]$ selected uniformly at random from the family, the image of any $k$-tuple, $(f(x_1), \ldots, f(x_k))$, is uniformly distributed in $[r]^k$. There exists no family of $k$-wise independent hash functions that meets our requirements – that is, small description and constant time evaluation. However, the construction of Siegel [20] is a good approximation and is sufficient for our purposes.

$\blacktriangleright$ Theorem 4 ([20]). Let $S \subseteq U = [m]$ be a set of $n = k^{O(1)}$ elements. For any constants $\varepsilon, c > 0$ there is a RAM algorithm constructing a random family $\mathcal{H}$ of functions in $o(n)$ time (provided $m = \text{poly}(n)$) and $o(k)$ words of space, such that:

- with probability $1 - O(1/n^c)$, $\mathcal{H}$ is $k$-wise independent;
- there is a RAM data structure of $O(k^{1+\varepsilon})$ words representing its functions such that a function can be evaluated in constant time. The data structure can be initialized to a random function in $O(k^{1+\varepsilon})$ time.
Significantly, the tail probabilities of \( k \)-wise independent random variables can be bound with a Chernoff-like result. The result we employ comes from Schmidt et. al [19].

**Theorem 5** ([19] Theorem 5.III.b). If \( X \) is the sum of \( k \)-wise independent random variables, each confined to the interval \([0,1] \), with \( \mu = \mathbb{E}[X] \) and \( k = \lceil \delta \mu e^{-1/3} \rceil \), then \( \Pr[|X - \mu| \geq \delta \mu] \leq e^{-\delta \mu / 3} \).

Negatively related random variables

Furthermore, in our analysis, we encounter instances of sums of indicator random variables that are not \( k \)-wise independent, but do hold the property of being negatively related. Happily, Jansen established that Chernoff bounds apply to sums of negatively related random variables [12] and we utilize this result in our analysis.

### 2.3 Prior work

The literature on list indexing assumes that item pointers are readily available (doubtless through a hash table) as arguments to the operations. A non-optimal or naive solution entails a straightforward application of a balanced (or height-bounded) binary tree. List items are stored, in order from left to right, in the leaves of the tree and internal nodes store a count of the number of leaves (or, the size of the sublist) in the subtree rooted at the internal node. An \( \text{index}(i) \) query begins at the root and, following the logic of the counts stored at internal nodes, can branch into the sublist containing the correct index. Conversely, \( \text{position}(x) \) begins at a leaf and accumulates a global index on a leaf-to-root path; the count of any left sibling at an internal node gets aggregated to the current index. Updates require recourse to a balancing criteria and may entail some restructuring. All operations take \( O(\log n) \) time (worst-case or amortized depending on the choice of tree) and the tree requires \( \Theta(n \log m) \) bits to store.

This idea was extended by Dietz with what are now fairly standard techniques [3]. The binary tree is replaced with a weight balanced \( B \)-tree with branching factor \( \Theta(log^2 m) \), for \( \varphi \in (0,1) \). As before, internal nodes count the number of leaves in the subtree they root. On a leaf-to-root path, a \( \text{position} \) query accumulates the counts, at each internal node, of all left siblings. To remove dependence on the branching factor, the sum of the counts of left siblings is evaluated on a partial sums data structure. The latter stores an array of integers \( A[1...b] \) and admits the following procedures.

- \( \text{add}(i, \delta) \): perform \( A[i] \leftarrow A[i] + \delta \), where \( \delta = \log \Theta(1) m \).
- \( \text{sum}(j) \): return \( \sum_{i<j} A[i] \).

On problem size \( b = O(\log^2 m) \), both operations can be performed in \( O(1) \) amortized time.

Thus, the cost of a \( \text{position} \) query is bound by the height \( (O(\log \log m n)) \) of the tree. Efficient navigation for \( \text{index}(i) \) queries is provided by the additional operation:

- \( \text{select_ps}(i) \): return the smallest \( j \) such that \( \text{sum}(j) \geq i \).

Although the partial sums structure of Dietz does not explicitly solve the abstraction of \( \text{select} \) queries of prefix sums\(^3\), a constant time solution, on small problem sizes, is provided by Raman et al. [18]. Thus, the cost of both query operations is bound by the height of the tree and is thereby optimal at \( O(\log n / \log \log n) \). Due to rebuilding requirements on both the partial sums structure and the weight balanced \( B \)-tree, update costs are amortized. The solution requires \( \Theta(n \log m) \) bits.

---

\(^3\) They augment the weight balanced \( B \)-tree with additional pointers that implicitly implement the functionality.
Figure 1 The Princess List. The list \( \mathcal{L} \) of items from \( U \supseteq \{1, \ldots, 16\} \) is hashed, via the function \( h : U \rightarrow \{0, \ldots, 3\} \), into the string \( h(\mathcal{L}) \). Items mapping to character \( c \) are stored in the (linear) sublist \( \mathcal{L}_c \). To evaluate \( \text{index}(9) \): identify \( h(\mathcal{L})[9] = 2 \); calculate \( \text{rank}_2(h(\mathcal{L}), 9) = 2 \); and perform \( \text{index}(\mathcal{L}_2, 2) = 14 \) with a linear scan.

Lemma 6 ([3]). The list indexing problem can be solved by a data structure, occupying \( O(n \log m) \) bits, that supports \( \text{index} \) and \( \text{position} \) in \( O(\log n/\log \log n) \) time and \( \text{insert} \) and \( \text{delete} \) in \( O(\log n/\log \log n) \) amortized time.

Andersson and Petersson define an approximate version of the problem [1]. The \( \text{position} \) query is permitted to err with a user defined relative error and the \( \text{index} \) query can retrieve any item from a neighborhood, of size proportional to the relative error, surrounding the correct index. The added flexibility allows the authors to remove the dependence on the problem size from the update and query costs. For relative error parameter \( \varepsilon \in (0, 1) \), queries can be evaluated in constant worst-case time and updates in amortized \( O(\varepsilon^{-2}) \).

3 The Princess List: a simple solution to list indexing

The linked-list, obliged to traverse all preceding nodes in the ordered set, performs the operation \( \text{index}(i) \) in \( O(i) \) time. To reduce this expense, the \( \text{PL} \) employs a divide-and-conquer strategy that partitions the list into a collection of disjoint sublists. A search proceeds on a single sublist and thereby reduces the expense of the linear scan. To allocate items to sublists, we map the list, via a hash function \( h \), onto a small alphabet \( \Sigma = [\sigma] \) and represent the mapped list

\[
h(\mathcal{L}) = \langle h(\mathcal{L}[0]), \ldots, h(\mathcal{L}[|\mathcal{L}| - 1]) \rangle
\]

with a dynamic string. Under this regime, multiple items may map to the same character and we store such a set of items under a (order-preserved) list representation. The random string \( h(\mathcal{L}) \) organises the branching into subproblems and, simultaneously, preserves the order of the full list. The range of the hash function permits control over the size of the sublists. An image of the \( \text{PL} \) arrangement, which outlines, at a high level, the division of the problem into subproblems and the support of the string \( h(\mathcal{L}) \) towards managing the relation between subproblems, is available in Figure 1.

Fixing notation, let \( M \) denote the list data structure used to implement the sublists \( \mathcal{L}_\Sigma \) and \( \text{PL}^M(\mathcal{L}, \sigma) \) name the subsequent Princess List representation. To evaluate \( \text{index}(i) \), the program begins by identifying the relevant sublist through the operation \( c = h(\mathcal{L})[i] \). Therefore, the item at location \( i \) in \( \mathcal{L} \) belongs to sublist \( \mathcal{L}_c \). Subsequently, the global location...
Succinct List Indexing

**Procedure** \( \text{index}^{PL}(i) \)
- \( c \leftarrow h(L)[i] \)
- \( i' \leftarrow \text{rank}_c(h(L), i) - 1 \)
- return \( \text{index}^M(L_c, i') \)

**Procedure** \( \text{position}^{PL}(x) \)
- \( c \leftarrow h(x) \)
- \( j \leftarrow \text{position}^M(L_c, x) + 1 \)
- return \( \text{select}_c(h(L), j) \)

**Procedure** \( \text{insert}^{PL}(x, y) \)
- \( i \leftarrow \text{position}^{PL}(x) \)
- \( c \leftarrow h(y) \)
- \( i' \leftarrow \text{rank}_c(h(L), i) \)
- \( z \leftarrow \text{position}^M(L_c, i') \)
- \( \text{insert}^M(L_c, z, y) \)
- \( \text{insertSt}(h(L), c, i + 1) \)
- return

**Figure 2** Query operations for the \( PL^M(L, \sigma) \) representation of the list \( L \). The data structure \( M \) is used to implement the sublists.

**Figure 3** Update operations for the \( PL^M(L, \sigma) \) representation of the list \( L \)

\( i \) needs to be translated to a local location on the sublist. This is achieved by a \( \text{rank} \) query; \( i' = \text{rank}_c(h(L), i) - 1 \) denotes the position of \( L[i] \) in the sublist \( L_c \). The query is completed by the operation \( \text{index}^M(L_c, i') \), which, under a linked-list representation, stipulates a linear scan and requires \( O(i') \) time. Conversely, to evaluate \( \text{position}(x) \), the program starts at the sublist \( L_{h(x)} \). The value \( j = \text{position}^M(L_{h(x)}, x) + 1 \) provides the rank of the character occurrence \( h(x) \) that corresponds to the item \( x \). The query is completed by determining the global index of the \( j \)th occurrence of \( h(x) \) in \( h(L) \) through the query \( \text{position}(x) = \text{select}_{h(x)}(h(L), j) \). Pseudo-code for the query operations is available in Figure 2.

The update \( \text{insert}(x, y) \) begins by identifying the index of insertion in both the string \( i = \text{position}^{PL}(L, x) \) and the relevant subproblem \( i' = \text{rank}_c(h(L), i) \). Character \( h(y) \) is then inserted at index \( i + 1 \) in the dynamic string and placed after item \( \text{position}^M(L_{h(y)}, i') \) in sublist \( L_{h(y)} \). The \( \text{delete}(x) \) operation proceeds in a similar fashion; it locates the index of the occurrence of \( x \) in \( h(L) \) and identifies the relevant subproblem \( c = h(x) \). The character \( c \) is then removed from \( h(L) \) and \( x \) is removed from \( L_c \). Pseudo-code for the update operations is available in Figure 3.

The efficiencies of the operations depend on the efficiency of the dynamic string \( h(L) \) and the size of the sublists. By Lemma 3, all string queries can be supported in \( O(\log n / \log \log n) \) time and string updates in \( O(\log n / \log \log n) \) amortized time. Each sublist has expected size \( n/\sigma \). The amount by which sublist sizes deviate from their expectation depends on both \( \sigma \) and the randomness of the underlying hashing scheme.

### 3.1 Towards a compact representation

In itself, the PL does not articulate a succinct form. Thus, before moving to the main result, we need to construct the bridge between the general form and an optimal encoding. This is achieved in three parts. First, we need to parametrize the PL by choosing an alphabet size. The verdict on the parameter choice is provided by the \( \Omega(\log n / \log \log n) \) time lower bound for the operations of an indexed list [4]. We require linear scans when
operating on sublists. Thus, it is desirable that subproblems have size $O(\log n / \log \log n)$. As sublists have expected cardinality $n/\sigma$, the latter implies an alphabet size of

$$\sigma = \Omega \left( \frac{n \cdot \log \log n}{\log n} \right). \quad (2)$$

Second, when following a standard linked-list implementation, the memory allocation of the subproblems trespasses over $n(\log m + o(\log n))$ bits. Therefore, an alternate list representation is required at the sublists. Multiple directions are available and we opt for packing a super-constant number of items in the nodes of a linked-list to reduce the per-item cost of the pointers. Third, in light of the space allocated to the dynamic string – which is $\omega(1)$ bits per character by equation (2) – we cannot afford to store the full key of each item. The key length can be reduced with a technique, standard in succinct dictionaries [17], named the quotient filter. A random permutation $\pi : [m] \rightarrow [m]$ is employed. For item $x$, the leftmost $\log \sigma$ bits of $\pi(x)$ specify the subproblem to which $x$ is a member and the rightmost $(\log m - \log \sigma)$ bits of $\pi(x)$ are stored in the list. In this manner, the representation of the item is split between a character in the dynamic string and an identifier in the allocated sublist. Permutations are necessary to avoid collisions in the sublists. The remainder of the paper illustrates the details of the bridge outlined above. We refer to this construction, an optimal instance of the PL, as PL+.

### 4 PL+: state-of-the-art wavelet trees and packed linked-lists

Items are allocated to sublists with Siegel’s hash family [20]. Therefore, by Theorem 4, with probability $1 - O(1/n^\gamma)$, item allocations are $k$-wise independent, for $k^O(1) = n$ and any constant $\gamma > 0$. Further, the hash function is evaluated in constant time and is stored in $o(n)$ bits. For our construction, we require that $\log n = \Theta(\log m) = \Theta(w)$. In other words, it is assumed that the universe size is a polynomial in the problem size.

Our structure, PL+, is an instantiation of the $\text{PL}^{\text{PackUnpack}}(L, \Theta(n \log n / \log n))$ form, where $\text{PackUnpack}$ refers to a type of linked-list that we detail below. To implement the dynamic string, PL+ appoints the wavelet tree of Navarro and Nekrich [15]. As the ideal alphabet size changes with $n$, the structure is periodically rebuilt to conform with the configuration of asymptotic constants of the alphabet. We set $\sigma$ to a power-of-two (for efficiency) such that

$$\sigma \in \left[ \frac{1}{2} n \cdot \frac{\log \log n}{\log n}, 2n \cdot \frac{\log \log n}{\log n} \right]$$

Therefore, a rebuild happens, in the worst-case, every $\Theta(n)$ updates. A simple rebuild can be performed by constructing a new string under the updated alphabet. This takes $O(n \log n / \log \log n)$ time by Theorem 3. In addition, selecting a new hash function from the constructed hash family takes $o(n)$ time. Periodically, but not with every rebuild, we will also have to reconstruct the hash family. The latter occurrence takes $o(n)$ time. Thus, rebuilding contributes $O(\log n / \log \log n)$ amortized time to each update.

#### 4.1 A random permutation

Without loss of generality, and for ease of demonstration, we assume that $m$ is a power-of-two. The permutation is generated from a collection of one-round Feistal permutations. We use them in a manner similar to Arbitman et al. [2], where existing hash functions, and the properties they inhabit, are recruited to generate the randomness. Let $H_{\sigma}$ be a class of
Succinct List Indexing

Figure 4 One-round Feistal permutation [2].

\[ h : [m/\sigma] \rightarrow [\sigma] \] functions. Let \( x_l \) denote the leftmost \( \log \sigma \) bits of a key \( x \) and \( x_r \) denote the rightmost \( \log(m/\sigma) \) bits. A permutation \( \pi_h : [m] \rightarrow [m] \) is defined for each instance \( h \in H_\sigma \):

\[ \pi_h(x) = (\pi_l(x), \pi_r(x)) = (x_l \oplus h(x_r), x_r). \]

This is a permutation as every pair of keys \( x \) and \( y \), where \( x_r = y_r \), receive the same hash value, but map to a different bucket under the \( \oplus \) operation. An image of the permutation is provided in Figure 4. With the permutation \( \pi \), an item \( x \) is allocated to a subproblem with the random character \( c = x_l \oplus h(x_r) \) and is stored under the representation \( x_r \) in \( L_c \). In this manner, items are allocated to subproblems according to the randomness prescribed by our hashing scheme.

4.2 The sublists

From Lemma 3 (with the parameter \( \varphi \) set to some constant), the dynamic string \( h(L) \) occupies

\[ n \log \sigma + o(n \log n) + O(\sigma \log n) = n(\log \sigma + o(\log n)) \]

bits. Thus, for a list implementation that is asymptotically close to \( n \log m \) bits, the remaining budget available to the sublists is \( n(\log m - \log \sigma + o(\log n)) \). As stated above, an item is stored in sublist \( L_{\pi_l(x)} \) with the \( (\log m - \log \sigma) \) (permuted) bits of \( \pi_r(x) \). This representation is in line with the budget stated above. It remains to demonstrate how the collection of sublists can be stored efficiently, with low redundancy.

A sublist is stored in a packed linked-list. A node in the linked-list contains at most \( \log \log n \) items stored in a packed array. If the keys are represented in \( K \) bits, the contents of each node occupy \( K \cdot \log \log n \) bits. Only the last node in the list is permitted to have less than \( \log \log n \) items. Consequently, the nodes are packed as tight as possible. Updates and queries can be performed on linear traversals. Note that the update operations require the shifting of items prior to insertion and after deletion.

As we perform linear scans (in the tradition of the linked-list), the runtime of operations depends on the cardinality of the sublists. Thus, for optimality, sublist cardinalities cannot exceed \( O(\log n / \log \log n) \). Unfortunately, in all likelihood, some sublists will contain \( \omega(\log n / \log \log n) \) items. To mitigate this outcome, we track the cardinalities of each sublist and, for a large enough constant \( C > 0 \), if a sublist cardinality exceeds \( C \log n \cdot \log \log n \), we implement the sublist with an uncompressed solution. The latter could be the time optimal solution put forth by Dietz [3] (see Lemma 6). With this set-up all operations require \( O(\log n / \log \log n) \) time to complete. We refer to this list data structure, which resorts to an uncompressed solution when its cardinality exceeds a specified threshold \( C \cdot \log n / \log \log n \), as a pack-unpack list with threshold parameter \( C \) (\text{PackUnpack}(C)).
4.3 Memory allocation for sublists

The proofs for the remainder of this section are located in Appendix B.2. The key with PL+ is to ensure that the number of items that belong to uncompressed sublists does not grow too large. To provide an upper bound on the latter, we compute both the maximum cardinality of the sublists and bound the number of sublists with cardinalities that exceed our threshold. We begin with the former. The upcoming pair of lemmas rely on the properties of $k$-wise independence. Recall that the construction of the hash family fails with probability $O(1/n^\gamma)$, for any constant $\gamma > 0$, and the analysis of the probabilities must account for this occurrence. We use the notation $\xi_{\gamma} = O(1/n^\gamma)$ to refer to this error term incurred by our choice of hash family.

**Lemma 7.** For $\gamma > 0$,

$$\max_{c \in \sigma} |L_c| \leq C_1 \log n$$

(5)

with probability at least $1 - e^{-O(C_1) \log n - \xi_{\gamma}}$.

The proof is a standard rehearsal of bounding the maximum load in a balls-in-bins problem with a Chernoff bound. We now bound the number of sublists that exceed the threshold.

**Lemma 8.** For $C_1 = O(\log n)$, $C$ set to a sufficiently large constant and any $\gamma > 0$,

$$|\{c \mid |L_c| \geq C \cdot \log n / \log \log n\}| \leq \frac{n}{C_1 \log n \cdot \log \log n}$$

(6)

with probability at least $1 - e^{-n/(O(C_1) \log n \log \log n) - \xi_{\gamma}}$.

Let $Q_C$ be the set of items that belong to unpacked lists when using threshold parameter $C$. The cardinality $|Q_C|$ is less than the product of the max load and the number of sublist cardinalities that exceed the threshold. Therefore, with the combination of Lemmas 7 and 8, and an appropriate choice of $C_1$, we arrive at the following result.

**Lemma 9.** For a sufficiently large constant $C$ and $\gamma > 0$, $|Q_C| \leq n / \log \log n$ with probability at least $1 - \xi_{\gamma}$.

Each item in an unpacked list requires $O(\log n)$ bits to store. Therefore, this result implies that, with high probability, items belonging to unpacked lists occupy a total of $O(n \log n / \log \log n)$ bits. This allocation is dominated by the space occupied by the packed lists and leads to the following result.

**Lemma 10.** For any $\gamma > 0$, if each item occupies $K$ bits, the memory allocation of the sublists, with sufficiently large threshold parameter $C$, aggregates to $n(K + o(\log n)) + o(n)K$ bits with probability at least $1 - O(1/n^\gamma)$.

5 Space and time efficiency of PL+

The memory allocation for PL+ is determined by the implementations of the string $h(L)$ and the sublists $\{L_c\}$. Therefore, with the combination of Lemma 3 and Lemma 10, we arrive at a bound on the memory allocation of PL+.

**Lemma 11.** For any $\gamma > 0$, to store a list on $n$ items, PL+ occupies $n(\log m + o(\log n))$ bits of space with probability at least $1 - O(1/n^\gamma)$. 

CVIT 2016
As the alphabet size is a function of the number of items, \( \text{PL}^+ \) needs to be rebuilt periodically. By relation (3), this occurs every \( \omega(n) \) updates. The change in alphabet also affects the permutation and we periodically require a different initialization of the hash family to support the scheme. As discussed above, the string can be rebuilt in \( O(n \log n / \log \log n) \) time. For a crude bound – which assumes that all items are uncompressed – the sublists can be rebuilt in \( O(n \log n / \log \log n) \) time. Therefore, the overall cost of the rebuild is \( o(\log n / \log \log n) \) amortized time per update.

\[ \text{Lemma 12.} \quad \text{On a list of } n \text{ items, } \text{PL}^+ \text{ performs index and position in } O(\log n / \log \log n) \text{ time and updates (insert or delete) complete in } O(\log n / \log \log n) \text{ amortized time.} \]

Combined, Lemmas 11 and 12 curate our main result, which we restate below.

\[ \text{Theorem 1.} \quad \text{For any constant } \gamma > 0, \text{ an ordered set of } n \text{ items, drawn from the universe } [m], \text{ where } m = \text{poly}(n), \text{ can be stored in } n(\log m + o(\log n)) \text{ bits with probability } 1 - O(1/n^\gamma) \text{ and support index and position in } O(\log n / \log \log n) \text{ time and insert and delete in } O(\log n / \log \log n) \text{ amortized time.} \]

The error term in Theorem 1 is dominated by the likelihood that the hash family fails. On the condition that the construction of the random hash family is successful, the desired bit allocation holds with probability \( 1 - (1/n)^{o(1)} \). The fail rate applies to each update. Therefore our result holds, with high probability, on any sequence of operations that is polynomial in length. Notably, the structure incurs, per-item, a sublogarithmic number of bits above the information theoretic bound and, taken with the efficient query and update operations, constitutes a succinct representation: by equation (1),

\[
n(\log m + o(\log n)) = B(m, n) + O(n) + n \cdot o(n) = (1 + o(1))B(m, n).
\]

The significance of the result is that the runtime of all operations is optimal, matching the previous best, and we reduce the memory commitment to a succinct representation with high probability.

To remove the probabilistic expression surrounding the memory allocation, we can just commit to using packed linked-lists for all sublists irrespective of their cardinalities. The outcome of this set-up is described in Theorem 2 and we provide the necessary detail and analysis in Appendix C.

\section{6 Conclusion}

The \( \text{PL} \) is an uncomplicated approach to list indexing. The algorithms are simple and offer immediate improvements over solutions that employ linear traversals. The data structure admits an instance that is close to optimal in space with respect to ordered sets. This is the first succinct indexed list with optimal time query and update operations. There are two directions for further work. In all prior work, update operations are amortized. Thus, even in the non-succinct case, designing worst-case update operations is an open problem. Further, achieving a non-probabilistic memory allocation with optimal time operations would conclude this line of work.

\section{References}


A Allowing for Repetitions

If we allow for repetitions, there are some precedents (from Python or Java) on how to modify the behavior of the operations. For example, the position function could return the lowest index of an item occurrence. The question is; how does this affect the theoretical performance?

Repetitions reduce the entropy of the list and the size of the lower bound. Certainly there are mechanisms that can leverage the reduction in entropy in the sublists (i.e. we can apply compression techniques). However, there is a point (in terms of the number of repetitions) at which this becomes a dynamic sequences problem and we would shift towards using the dynamic string of Munro and Nekrich [13].

The key difference between our version of the problem and that of the original problem proposed by Dietz [3], is that the updates and the position query take pointers (or "records") as inputs. It would make less sense to have repeated records in the list if we access them via pointers. Thus it is sensible to think of these records – each associated with a key in the list – as distinct from each other. Given that the index query returns a key but not a record, we think, even under the Dietz version of the problem, it is natural to think of the keys as distinct.

Further, we think our formalization of indexing an ordered set is what makes this an interesting problem.

B Relegated Proofs

B.1 Memory Allocation for Sublists

Lemma 7. For $\gamma > 0$,

$$\max_{c \in [\sigma]} |L_c| \leq C_1 \log n$$

with probability at least $1 - e^{-\Theta(C_1) \log n} - \xi \gamma$.

Proof. Fix a sublist $c$. Let $X_i$ be an indicator for the event that the $i^{th}$ item added to $L$ is placed in sublist $c$. It holds that $E[X_i] = 1/\sigma$. The cardinality $X = \sum_{i=1}^{n} X_i$ of sublist $c$ is the sum of $k$-wise independent random variables (for $k = n^\alpha, \alpha \in (0, 1)$). Therefore, by Theorem 5, with $\mu = E[X] \in [\log n/(2 \log \log n), 2 \log n/ \log \log n]$ by Equation (3),

$$Pr[X \geq C_1 \log n] \leq Pr[X \geq C_1 \log \log n/2 \cdot \mu]$$

$$\leq e^{-\Theta(C_1) \cdot \log n},$$

by our choice of $k$ and Theorem 4. The third line holds on the condition that $C_1 \geq 3 \log \log n$. A union bound absorbs the probability that the construction of the hash family fails and we arrive at the stated result.

Lemma 8. For $C_1 = O(\log n)$, $C$ set to a sufficiently large constant and any $\gamma > 0$,

$$|\{c \mid |L_c| \geq C \cdot \log n/ \log \log n\}| \leq \frac{n}{C_1 \log n \cdot \log \log n}$$

with probability at least $1 - e^{-n/(O(C_1) \log n \log \log n)} - \xi \gamma$.  


Before we proceed with the proof, we need an additional result. Let $X_{i,c}$ indicate that item $i$ is allocated to sublist $c$ and $X_c = \sum_{i=1}^n X_{i,c}$ denote the sublist cardinalities for $c \in [\sigma]$.

Further, we define

$$Y_c = \begin{cases} 1 & \text{if } X_c > C \cdot \log n / \log \log n \\ 0 & \text{otherwise} \end{cases}$$

to indicate the event that sublist $c$ exceeds the threshold. The task is to bound the sum $\sum_{c=1}^\sigma Y_c$. However, the random variables $\{Y_c\}$ are not independent. Fortunately, the $\{Y_c\}$ are negatively related and a Chernoff bound can be applied to acquire a concentration result on the sum of the indicators [12]. Negatively related random variables are defined as follows.

**Definition 13** ([12]). The indicator random variables $\{Y_i\}_{i \in [\sigma]}$ (defined on some probability space) are negatively related if for each $j \leq \sigma$ there exists further random variables $\{J_{i,j}\}_{i \in [\sigma]}$ defined on the same probability space such that:

- the definition of the random vector $\{J_{i,j}\}_{i \in [\sigma]}$ equals the conditional distribution of $\{Y_i\}_{i \in [\sigma]}$ given $Y_j = 1$;

- $J_{i,j} \leq Y_j$, $\forall i \neq j$.

**Lemma 14.** The random variables $\{Y_c\}_{c \in [\sigma]}$ are negatively related.

**Proof.** We can imagine this as an experiment where $n$ balls are thrown into $\sigma$ bins where the bin locations for the balls are $n^\alpha$-wise independent for $\alpha \in (0,1)$. Let $X_c$ denote the load of bin $c$ and let $L = C \cdot \log n / \log \log n$. The indicator $Y_c$ is 1 if $X_c > L$. To demonstrate that the indicators $\{Y_c\}$ are negatively related we construct the distributions $\{J_{c,j}\}_{c \in [\sigma]}$ for $j \in [n]$ that satisfy the two properties of Definition 13. Fix a $j \in [\sigma]$. The random variables $\{J_{c,j}\}$ take the following form. First throw the $n$ balls into $\sigma$ bins as above. If $X_j > L$, then set $J_{c,j} = Y_c$ for all $c \in [\sigma]$. Otherwise, pick $(L - X_j + 1)$ balls uniformly at random from bins in $[\sigma] \setminus \{j\}$ and place them in bin $j$. Let $X_c^*$ denote the modified bin loads. Now, $J_{c,j} = 1$ if $X_c^* > L$. As the bin loads $X_c^*$ are conditioned on $X_j > L$, the indicators have the correct distribution. Further, $J_{c,j} \leq Y_c$ as we only remove items from bins. Thus, the $\{Y_c\}$ are negatively related.

Now that we can apply a Chernoff bound, we can complete the proof of Lemma 8.

**Proof of Lemma 8.** As $X$ is the sum of $k$-wise independent random variables, by Theorem 5, our choice of $k$ and the observation that $\mu = E[X] \geq \log n / (2 \log \log n)$, the following holds:

$$\Pr[Y_c = 1] = \Pr[X_c > C \cdot \log n / \log \log n]$$

$$\leq \Pr[|X_c - \mu| > (C/2 - 1) \cdot \mu]$$

$$\leq e^{-O(C)^\mu}.$$  

Let $Y = \sum_{c=1}^\sigma Y_c$ denote the number of sublists that exceed the cardinality threshold.

$$E[Y] \leq e^{-O(C)^\mu} \cdot \sigma \leq e^{-O(C) \log n / \log \log n} \cdot \frac{n \log \log n}{\log n} \leq \frac{n}{(\log n)^{O(C)}}.$$  \hspace{1cm} (7)

The indicators $\{Y_c\}$ are not independent. However, by Lemma 14, they are negatively related and we are permitted to apply a Chernoff bound on their sum. Therefore, for $\delta = n/(C_1 \cdot 2E[Y] \log n \log \log n)$,

$$\Pr[Y > n/(C_1 \cdot \log n \log \log n)] = \Pr[|Y - E[Y]| > \delta E[Y]] \leq e^{-n/(O(C_1) \log n \log \log n)}$$
on the condition that $\delta \geq 1$. The latter occurs if

$$1 \leq \delta = \frac{n}{C_1 \cdot 2E[Y] \log n \log \log n} \leq \frac{(\log n)^{O(C)}}{2C_1 \log \log n}$$

The second inequality comes from Equation (7) and holds for $C_1 = O(\log n)$ and sufficiently large $C$. Adding the error rate of the hash family, via the union bound, completes the proof.

Lemma 9. For a sufficiently large constant $C$ and $\gamma > 0$, $|Q_C| \leq n/\log \log n$ with probability at least $1 - \xi_{\gamma}$.

Proof. By design, $|Q_C| \leq n/\log \log n$ if inequalities (5) & (6) both hold. By Lemmas 7 and 8, with $C_1 = \Theta(\log n)$, in conjunction with a union bound, both the inequalities hold with probability $1 - (1/n)^{(1)} - \xi_{\gamma}$. This completes the proof.

Lemma 10. For any $\gamma > 0$, if each item occupies $K$ bits, the memory allocation of the sublists, with sufficiently large threshold parameter $C$, aggregates to $n(K + o(\log n)) + o(n)K$ bits with probability at least $1 - O(1/n^{\gamma})$.

Proof. The structure is comprised of at most $\sigma$ packed linked-lists. These linked-lists contain at most $n/\log \log n + \sigma$ nodes. Each node contains $\log \log n$ items of $K$ bits and two $O(\log n)$ bit pointers. This accumulates to a bit commitment of size

$$\left(\frac{n}{\log \log n} + \sigma\right)(K \cdot \log \log n + O(\log n))$$

$$= nK + n \cdot o(\log n) + \sigma K \cdot \log \log n + \sigma O(\log n)$$

$$= n(K + o(\log n)) + K \Theta\left(\frac{n \log^2 \log n}{\log n}\right) + \Theta(n \log \log n)$$

$$= n(K + o(\log n)) + o(n)K. \quad (8)$$

By Lemma 9, the structure contains at most $n/\log \log n$ uncompressed items with probability $1 - \xi_{\gamma} = 1 - O(1/n^{\gamma})$. Each uncompressed item occupies $O(\log n)$ bits. Thus, combined, uncompressed items occupy

$$n/\log \log n \cdot O(\log n) = n \cdot o(\log n)$$

bits. This allocation is absorbed by the allocation for compressed items (Equation (8)).

B.2 Performance for PL+

Lemma 11. For any $\gamma > 0$, to store a list on $n$ items, PL+ occupies $n(\log m + o(\log n))$ bits of space with probability at least $1 - O(1/n^{\gamma})$.

Proof. For the permutation $\pi : [m] \to [m]$, the leftmost $\log \sigma$ bits specify the subproblem and the rightmost $[\log m] - \log \sigma$ bits are stored in the sublist. Thus, the space occupied by the sublists is determined by Lemma 10 with key size $K = [\log m] - \log \sigma$. By equation (4), the cost of the dynamic string is $n(\log \sigma + o(\log n))$ bits. Aggregated with the sublists, we get a memory commitment (measured in bits) of

$$n(\log \sigma + o(\log n)) + nK + n \cdot o(\log n) + o(n)K$$

$$= n(\log \sigma + o(\log n)) + n \log(m/\sigma) + o(n) \log(m/\sigma)$$

$$= n(\log m + o(\log n)) + o(n)$$

$$= n(\log m + o(\log n)).$$
Lemma 12. On a list of \( n \) items, \( PL^+ \) performs \textit{index} and \textit{position} in \( O(\log n / \log \log n) \) time and updates (\texttt{insert} or \texttt{delete}) complete in \( O(\log n / \log \log n) \) amortized time.

Proof. By Theorem 3, taking parameter \( \varphi \) as constant, all queries to the random string take \( O(\log n / \log \log n) \) time and update operations require \( O(\log n / \log \log n) \) amortized time. By Lemma 6, all operations on the uncompressed implementation for \texttt{PackUnpack} take \( O(\log n / \log \log n) \) time. The query \texttt{index}^\texttt{PL+}(i) demands two operations on the random string and an \texttt{index}^\texttt{PackUnpack} query on a sublist. The local \texttt{index}^\texttt{PackUnpack}(L, i') requires either a linear scan of length \( O(\log n / \log \log n) \) or an index operation on a fast uncompressed solution. The \texttt{position}^\texttt{PL+} query requires one constant time hash evaluation, a \texttt{select} query and \texttt{position}^\texttt{PackUnpack} query on the subproblem. Similar to the case of the \texttt{index} query, all subroutines take \( O(\log n / \log \log n) \) time and the claim for \texttt{position} holds.

An update is obtained by a combination of an update to the dynamic string and and update to the subproblems. The former operation has an amortized cost of \( O(\log n / \log \log n) \) and the runtime of the latter, similar to the query operations, requires a constant number of (possibly amortized) \( O(\log n / \log \log n) \) time subroutines. Periodic rebuilding adds amortized \( o(\log n / \log \log n) \) time to each update. This completes the proof.

C Alternative implementation for \( PL^+ \)

In our main result we achieve a succinct representation with high probability. The structure can be modified such that a succinct representation occurs with probability one: we commit to using packed linked lists (\texttt{pack}) and tolerate sublists with cardinality \( \omega(\log n / \log \log n) \). Further, the local \texttt{index}^\texttt{Pack}(L, i) query can take advantage of knowing which node the offset \( i \) belongs to. For every packed linked-list, we store a dynamic array of pointers, which we name \texttt{skip} pointers, where the \( j^{\text{th}} \) pointer points to the \( j^{\text{th}} \) node in the chain. This permits constant time access to a specified node. Consequently, the \texttt{index}^\texttt{Pack}(L, i) performs small \( O(\log n) \) traversals on a portion of the sublist \( L \). The \texttt{skip} pointers add a negligible \( \sigma \cdot O(\log n) \) bits to the memory allocation of the structure. The runtimes of the other operations depend on the cardinality of the engaged sublist. The latter is at most \( O(\log n) \) with probability \( 1 - O(1/n^\gamma) \). Note that Lemma 10 accounts for \( \sigma \) packed lists. Thus, we arrive at the following result.

Theorem 2. For any constant \( \gamma > 0 \), an ordered set of \( n \) items, drawn from the universe \([m]\), where \( m = \text{poly}(n) \), can be stored in \( n(\log m + o(\log n)) \) bits and support \texttt{index} in \( O(\log n / \log \log n) \) time, \texttt{position} in \( O(\log n) \) time with probability \( 1 - O(1/n^\gamma) \) and any sequence of \( O(n) \) updates (\texttt{insert} or \texttt{delete}) takes \( O(n \log n) \) time with probability \( 1 - O(1/n^\gamma) \).

As the expected cardinality of a sublist is \( n/\sigma = O(\log n / \log \log n) \), the expected runtime of \texttt{position} and the expected amortized runtime of the updates is \( O(\log n / \log \log n) \).
Chapter 4

Succinct Recency

In continuing our work on solutions to recency queries, we present a paper published\(^1\) in ISAAC 2020 as a response to Research Question 2. The data structure that constitutes the paper, HistoricalMembership, accommodates approximate time-recency queries, over a sliding window, in small-memory. The sliding window introduces a variation of Definition 1, but is critical for providing a small-memory representation.

The study was conducted three years ago and was the starting point for the thesis. We began with the informal query “when was the last time we saw item \(x\)?”. The first step was formalizing this notion, which is available in Problem 2. Other works had presented solutions to the recency query, mostly through the use of hash tables \([25, 105, 131]\), but this is the first study to consider its formal boundaries. For example, we ask: what is the minimum amount of memory we require to support recency queries?

Prior solutions to our formalization of the problem can be found in the sliding-membership literature (Subsection 2.4). In this setting, we are interested in answering membership queries over the most recent \(W\) occurrences on a data stream (the sliding window). This query is related to recency as knowledge of when an item occurs discloses whether the item occurred on the window. A simple solution, with bounded absolute error, can be constructed by dividing the window into segments of equal width and storing the segments with a set-membership structure. This approach is exemplified in Figure 2.4. Membership of a segment denotes a contiguous interval, in time, in which the item occurred. With this strategy of dividing the window into segments, Naor et al. provide a state-of-the-art solution with a succinct representation [114]. With HistoricalMembership, we refine this approach. We maintain the succinct representation of the window, but improve the accuracy to support recency queries with user-defined relative error. The cost of this accuracy improvement is a logarithmic increase in the runtime of the operations.

An earlier version of HistoricalMembership did not build on sliding membership components. The high-level structure, with the use of equivalence classes to denote epochs of time on the stream, is more-or-less the same. However, the components were made up of ad-hoc compression techniques and achieved only a compact\(^2\) representation. This non-optimal vari-


\(^2\)If the information theoretic lower bound for the query is \(\mathcal{O}\) bits, a compact representation encodes the infor-
The Recency Problem and its Applications

William L. Holland

ant of HistoricalMembership is the version we use in Chapter 5 when applied in oblivious storage solutions. We elaborate on this in more detail in the preamble to Chapter 5, but, essentially, we required some additional functionality and could afford the shift to the non-succinct representation.

The biggest limitation of the paper is the lack of a concrete application. The formal treatment of the problem is certainly novel, but, at the time, we didn’t furnish the result with any plug-and-play application. The second half of the thesis addresses this shortcoming. Chapter 5 demonstrates how to use recency queries in the domain of Hierarchical ORAMs.

One remaining question is whether the query and update times can be reduced further. My expectation is that the logarithmic bound could be a limit. But a lower bound still needs to be established.
Recency queries with succinct representation

William L. Holland
School of Computing and Information Systems, The University of Melbourne, Parkville, Australia
w.holland@student.unimelb.edu.au

Anthony Wirth
School of Computing and Information Systems, The University of Melbourne, Parkville, Australia
awirth@unimelb.edu.au

Justin Zobel
School of Computing and Information Systems, The University of Melbourne, Parkville, Australia
jzobel@unimelb.edu.au

Abstract

In the context of the sliding-window set membership problem, and caching policies that require knowledge of item recency, we formalize the problem of Recency on a stream. Informally, the query asks, ‘when was the last time I saw item \(x\)?’ Existing structures, such as hash tables, can support a recency query by augmenting item occurrences with timestamps. To support recency queries on a window of \(W\) items, this might require \(\Theta(W \log W)\) bits.

We propose a succinct data structure for Recency. By combining sliding-window dictionaries in a hierarchical structure, and careful design of the underlying hash tables, we achieve a data structure that returns a \(1 + \varepsilon\) approximation to the recency of every item in \(O(\log(\varepsilon W))\) time, in only \((1 + o(1))(1 + \varepsilon)(B + W \log(e^{-1}))\) bits. Here, \(B\) is the information-theoretic lower bound on the number of bits for a set of size \(W\), in a universe of cardinality \(N\).

2012 ACM Subject Classification  Theory of computation → Data structures design and analysis

Keywords and phrases  Succinct Data Structures, Data Streams, Sliding Dictionary

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2020.54

Acknowledgements  This work was supported by the Australian Research Council, grant number DP190102078, and an Australian Government RTP Scholarship. We acknowledge the Wurundjeri People of the Kulin Nations as traditional owners of the land on which we live and work.

1 Introduction

In evolving data streams, applications are often interested in recent history. This is evident in recency-sensitive applications such as in-network caching [5], web-crawling and the detection of duplicates [7]. Data structures that provide summaries of item-histories in support of these applications can be found in the sliding membership\(^1\) literature [2, 6, 8, 11]. The summaries are set membership structures fused with estimates of item recency; a statistic identical to the question ‘when was the last time I saw item \(x\)?’. Item recency can be supported by a hash table by augmenting item occurrences with timestamps or some identifier of a point in history. However, this strategy has not been fulfilled in both “small” space and with sufficient accuracy. Thus, we look at the problem of balancing these demands in the design of a data structure that supports a recency query.

What makes the recency and the sliding membership problems compelling, particularly in the context of the broader dictionary literature, is decay: the data structure must forget old items. Thus, a mechanism is required to determine the age of an item and, in turn, identify candidates for omission. If we want to keep the representation of the set of items

\(^1\) The sliding membership problem asks: ‘has item \(x\) occurred among the last \(W\) items?’
succinct (a representation of a set that occupies an amount of space close to the theoretic information lower bound), the question of how to determine item recency is non-trivial. Additional information must be associated with each item. So we may ask whether the size of this information is dependent on the number of items present and, in turn, whether this precludes a succinct representation. In this paper, we present a space-efficient data structure to support recency queries with relative accuracy, maintaining succinct representations of the occurred items.

1.1 Formalizing the problem

In the sliding membership problem, on parameters $D, W \in \mathbb{Z}^+$, a process observes a sequence of items $S(t) = \langle s_1, s_2, \ldots, s_t \rangle$, from a universe $[N]$, we seek to answer membership queries on item $x$ in the length-$W$ window of most recent items, $S_W(t) = \langle s_{t-W+1}, \ldots, s_t \rangle$:

$$x \in S_W(t) = \begin{cases} 
\text{yes} & \text{if } x \in S_W(t), \\
\text{no} & \text{if } x \notin S_{W+D}(t), \\
\text{yes or no} & \text{otherwise.}
\end{cases}$$

Including the slack parameter $D$ allows for more efficient solutions [8].

Recency

A recency query, $r(x,t)$, takes sliding membership a step further, and returns a measure of the age of an item $x$:

$$r(x,t) = t - \max\{j \mid s_j = x, j \leq t\}.$$  

Trivially, sliding membership reduces to recency, as $(r(x,t) \in [0,W])$ answers $x \in S_W(t)$.

The commonly cited naïve solution for sliding membership, with slack parameter $D > 0$, entails dividing the window into blocks of width $D$. Each block is stored in a static dictionary. To evaluate item membership on the sliding window it suffices to query each block, in turn, and return the logical disjunction of the results. Similarly, the recency of an item $x$, given that $x \in S_W(t)$, can be approximated by returning the recency of the (youngest) block it belongs to. This approach returns approximations with absolute error at most $D$. However, when viewed from the perspective of relative error, estimates are less accurate for items with low recency. Accurate estimates may be required throughout the window and are arguably more valuable for more recent items, motivating our formalization of the Recency problem.

Problem 1 (Recency). Given $W, D \in \mathbb{Z}^+$ and $\varepsilon \in (0,1)$, and the sequence $S(t) = \langle s_1, s_2, \ldots, s_t \rangle$ from universe $[N]$, when presented with some item $x \in [N]$, return an estimate $\hat{r}$ for $r(x,t)$ where

$$\hat{r} \in \begin{cases} 
(1+\varepsilon)r(x,t), & \text{if } x \in S_W(t), \\
-1, & \text{if } x \notin S_{W+D}(t), \\
(1+\varepsilon)r(x,t) \cup \{-1\}, & \text{otherwise.}
\end{cases}$$

To set the context of our contributions, we briefly consider approaches to Recency that are nearly immediately at hand. The solutions in the sliding membership literature take a
general form: insert (item-signature, timestamp) pairs into a hash table. This approach either incurs a large memory overhead [6] or does not admit bounded relative error [8]. As an alternative approach, one could store items in a circular array of length $O(W)$. However, queries are linear in the length of the array. This cost could be reduced to the cardinality of the window with a move-to-front list [10]. Both approaches are non-succinct in memory.

1.2 Contribution

We introduce a data structure named (HistoricalMembership) that achieves tight memory allocation and bounded relative error in return for logarithmic update and query times. Our solution builds on existing approaches, particularly the tactic of dividing the window into blocks of items of equivalent age. However, to achieve both relative $(1 \pm \varepsilon)$ accuracy in item recency and also space efficiency, the structure is hierarchical, comprising levels of geometrically increasing size. Level $l$ of HistoricalMembership is a sliding dictionary with a window of $\varepsilon^{-1}2^l$ items and slack of $2^l$, divided into blocks of size $2^l$.

We illustrate the formal validity of our approach and conjecture whether improvements are possible. Our main result is captured in the following theorem.

▶ Theorem 1. On a sequence of $S = \langle s_1, s_2, \ldots \rangle$, where $s_i \in [N]$, for parameters $W$ and $\varepsilon > 0$, at each timestamp $t \geq 1$, HistoricalMembership solves the Recency problem in

$$(1 + o(1))(1 + \varepsilon)(B + W \log(\varepsilon^{-1}))$$

bits of memory, admitting query and update times of item $x$ in $O(\log(\varepsilon \cdot r(x, t)))$. This bound is with high probability worst case for queries and expected amortized for updates. Here, $B$ is the information-theoretic lower bound for storing a subset of size $W$ from the universe $[N]$.

Importantly, HistoricalMembership achieves $\varepsilon$-approximation to item recency in space asymptotically identical to the state-of-the-art data structure for sliding membership [8]. The auxiliary information is not dependent on $W$ and the representation of the items, including the slack, is succinct.

2 Background

Succinct data structures

We adhere to the word RAM model of computation with word size $w = \Theta(\log N)$. Elements from the universe $N$ can be stored in $O(1)$ machine words and bitwise operations such as arithmetic require constant time. There exist $\binom{N}{m}$ distinct size-$m$ subsets of the universe $[N]$ so an encoding of such a subset requires, on average, at least

$$B(N, m) = \log \left( \binom{N}{m} \right) = m \log \frac{N}{m} + O(m)$$

bits. If the information theoretic memory lower bound for a data structure that supports a particular query is $B$ bits, then a succinct data structure is one that requires $(1 + o(1))B$ bits.

We refer to a data structure that solves (sliding) set membership as a (sliding) dictionary.

2 The type of item-signature depends on the context. For exact set membership the key $x \in [N]$ is sufficient. For approximate set membership, where collisions are allowed, the hash value $h(x)$ may be used to identify the item. Many solutions to approximate set membership are simply reductions to exact set membership via a hash function that reduces the key space.
Recency queries with succinct representation

Table 1 Comparison of HistoricalMembership with existing art. Term $B$ denotes the information-theoretic lower bound for storing a set of $W$ items from the universe $[N]$. ExactCuckoo solves exact recency. OptimalSM solves approximate recency with bounded absolute error $\leq D - 1$, with $D \leq W$. HistoricalMembership solves approximate recency with bound relative error.

<table>
<thead>
<tr>
<th></th>
<th>Update time</th>
<th>Query time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExactCuckoo [6]</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(W(\log N + \log W))$</td>
</tr>
<tr>
<td>OptimalSM [8]</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$(1 + o(1))(1 + D/W)(B + W \log(W/D))$</td>
</tr>
<tr>
<td>HistoricalMembership</td>
<td>$O(\log(\varepsilon W))$</td>
<td>$O(\log(\varepsilon W))$</td>
<td>$(1 + o(1))(1 + \varepsilon)(B + W \log(\varepsilon^{-1}))$</td>
</tr>
</tbody>
</table>

BackyardCuckoo

The foremost primitive in HistoricalMembership is the backyard cuckoo hash table (BackyardCuckoo) [1], a two-level variant of cuckoo hashing [9] that achieves improved memory utilization$^3$ and (with high probability) worst-case constant update and query times. The attributes of BackyardCuckoo are summarized in the following theorem.

Theorem 2 ([1]). A dynamic set of items, drawn from the universe $[N]$, of size at most $m$, can be stored in $(1 + o(1))B(N, m)$ bits. With probability at least $1 - 1/poly(m)$, insert, delete and query are performed in worst-case constant time.

In addition, backyard cuckoo hashing allows auxiliary information to be stored with each item. When a query is lodged, via an item-signature, the auxiliary information is (also) returned. With this extension, for auxiliary information of at most $K$ bits, the hash table uses $(1 + o(1))(B(N, m) + mK)$ bits of memory.

Sliding approximate membership

Sliding dictionaries are an established class of data structure and some of them (indirectly) solve the recency problem. We seek sliding dictionaries that can return their internal measure of recency. One class of solutions fails to do this [2, 7]. However, the solutions based on hash tables [5, 8], which store items with an amount of auxiliary information indicative of the age of the item, do provide recency estimates and are a starting point in our inquiry.

As a baseline for the exact recency problem, Liu et al. [6] propose storing (item-signature, timestamp) pairs in a cuckoo hash table. Expired items are identified as those with timestamps sat outside the window. For time efficiency, the deletion of expired items is performed lazily. This is done in one of two ways. First, if an expired item is in encountered during an insert or query, it is deleted. Second, at the completion of an insertion, a constant number of cells are scanned and expired items are deleted. The process records the finishing point of the scan and resumes at this location during the next update. These measures ensure that an expired item is deleted within a time frame of $W$ updates and to save space, timestamps are assigned modulo $2W$. Insertions take expected amortized constant time, while queries are worst-case constant as per cuckoo hashing theory [9]. Memory utilization is almost $1/2$, so the table contains a large proportion of empty cells. In total, the table requires $O(W(A + \log W))$ bits, where $A$ is the size of the item-signature.$^4$ Notably, as timestamps are stored, the data structure solves the exact recency problem. We refer to this solution as ExactCuckoo.

---

$^3$ Utilization arbitrarily close to 1, as opposed to arbitrarily close to 1/2 for standard cuckoo hashing.

$^4$ $\lceil \log 1/\delta \rceil$ for approximate set membership and $\lceil \log N \rceil$ for exact set membership.
The theoretical state-of-the-art solution for absolute-error Recency is by Naor and Yogev [8]. It is the only approach that accounts for slack, and moreover measures its benefit. We refer to the data structure as Optimal Sliding Membership (OptimalSM). The solution entails partitioning the window into blocks of size $D$. Then (item-signature, block-number) pairs are stored in a hash table. As above, block IDs are assigned on a circular field and evictions are lazy. With this approach, and by introducing slack, OptimalSM reduces the cost of the timestamp compared with ExactCuckoo.

Blocks that overlap with the window are called active and those that sit outside the window as expired. During an insertion, the procedure assigns the item to the youngest active block, which we call the contemporary block. At each timestamp, there are $W/D + 1$ active blocks and the circular field of block IDs is modulo $2(W/D + 1)$. The hash table is BackyardCuckoo and a succinct representation of the items is obtained. A key contribution is a lower bound on the sliding approximate membership problem.

▶ Theorem 3 ([8]). For parameters $W, D \in \mathbb{Z}^+$ and failure probability $\delta \in (0, 1)$, a data structure that returns approximate set membership queries on the length-$W$ sliding window, with slack $D$, requires at least the following number of bits

$$W \log \frac{1}{\delta} + W \cdot \max\{\log \frac{W}{D}, \log \log \frac{1}{\delta}\} - O(W).$$

Naor and Yogev’s construction is tight up to the first two terms, and OptimalSM solves Recency with bounded absolute error.

The ExactCuckoo and OptimalSM share this approach: store an item-signature with a time indicator, such as a timestamp or block ID in a hash table. In other words, we can think of the hash table as a black box, and so we collectively refer to these two approaches as Hash Sliding Membership (HashSM). HistoricalMembership employs multiple instances of HashSM to construct a Recency data structure with bounded relative error. A comparison between ExactCuckoo, OptimalSM and HistoricalMembership is available in Table 1.

3 HistoricalMembership

With ExactCuckoo, the sliding membership literature provides a solution for exact Recency. However, the structure does not suggest an approximate solution, nor how to trade (relative) accuracy for space, or rather, to reduce the $O(\log W)$ bit allocation for timestamps. The approach of Noar and Yogev [8], which involves dividing the window into blocks of fixed length, provides an approximation of bounded absolute error. This approach is sufficient for identifying old items, with large recency values, and is therefore ideal for sliding membership. Through the notion of a recency equivalence class, we refine their approach towards a structure that is sensitive to relative error.

3.1 Equivalence classes

Following OptimalSM, suppose we partition the window into blocks, and assign each item a block ID. The most recent item in the fourth block has a recency value of $3D$ and we can approximate the recency of all items in the block by assigning them the same value of $3D$. As the oldest item in the block has actual recency $3D + D - 1$, the absolute error of the estimate is at most $D - 1$ and hence incurs relative error approximately $1/4$. To provide a recency value with (parameterized) bounded relative error, pertinent for items with
Recency queries with succinct representation

![Figure 1](image_url)

A sequence \( \langle s_1, \ldots, s_{12} \rangle \) is partitioned into blocks of non-decreasing size. The timestamp of the ‘front’ item of each block becomes an implicit timestamp for all items in the block. For example, the token \( s_1 \) is assigned a recency estimate of 7. Since its actual recency is 11, it incurs a relative error of \( \left( \frac{11}{7} - 1 \right) \). Storing each item in the set \( S = \{ s_1, \ldots, s_{12} \} \) with its corresponding block number gives a static solution to approximate recency.

For low recency values, \textit{HistoricalMembership} partitions the window into a sequence of blocks of non-decreasing size. Every item in a block is deemed to have recency equivalent to the recency of the most recent item (the ‘front’ item) in that block. An example is shown in Figure 1. As a byproduct, we lose order and granularity within the blocks themselves, with each block now a homogeneous zone of recency with a single representative timestamp. In other words, \textit{HistoricalMembership} maintains a partial order of the underlying sequence. We want the front item to be a good representative for the block. We hence refer to a block that is unified by a \((1 + \varepsilon)\)-approximation as an equivalence class. To make this notion more rigorous, a pair of timestamps, \((t_a, t_b)\), satisfying the two conditions

\[
t_a < t_b \quad \text{and} \quad (t - t_a) \leq (1 + \varepsilon)(t - t_b)
\]

defines an equivalence class. The difference, \( t_b - t_a \), is the width of the equivalence class.

As an equivalence class demarcates a contiguous neighbourhood of items from the sequence, the approach of \textit{HistoricalMembership} is to dynamically re-organize the sequence into an appropriate collection of equivalence classes, as each item arrives in the stream. This constitutes a high-level view of \textit{HistoricalMembership}, but the efficiency of the structure depends on how the classes are stored and accessed.

### 3.2 Coordinating the equivalence classes

To perform efficient maintenance of the dynamic collection of equivalence classes, we propose merging adjacent classes. This tactic follows from the observation that older classes can be wider. As a class ages, and moves further away from the present, it implicitly becomes more (relatively) accurate. Suppose the two intervals \([t_a, t_b)\) and \([t_b, t_c)\) represent blocks within the division of the window. If the pair \((t_a, t_c)\) satisfies the conditions (2), the structure has permission to merge the blocks.

\textbf{Invariant}

Accordingly, within the framework of merge-type equivalence class maintenance, \textit{HistoricalMembership} organizes the classes into \( L = \lceil \log(\varepsilon W) \rceil - 1 \) levels. At level \( l \), the width of each equivalence class is \( 2^l \). Therefore, merging two classes at level \( l \), of width \( 2^l \), creates a new class of width \( 2^{l+1} \) and a resident of level \( (l + 1) \). To maintain the equivalence class constraints of relation (2), we insist that at least \( \varepsilon^{-1} \) and at most \( \varepsilon^{-1} + 1 \) classes at reside at...
each level. This ensures that each item at level \( l \) has recency at least
\[
\varepsilon^{-1} \sum_{i=0}^{l-1} 2^i = \varepsilon^{-1}(2^l - 1),
\]
which, in turn, justifies the width of the classes (refer to Lemma 4 below). Consequently, we require that \( \varepsilon^{-1} \) is an integer. Reducing the \( \varepsilon \) term in the approximation factor increases the number of classes at each level. Observe that \( \sum_{l=0}^{L} \varepsilon^{-1}2^l \leq W \); we refer to the difference
\[
E = W - \sum_{l=0}^{L} \varepsilon^{-1}2^l
\]
as the excess. We can either extend level \( L \) to include the excess or create a new level for it. \texttt{HistoricalMembership} opts for the former approach, as it is more space effective to extend the top level than to create a new level that operates below its conceptual cardinality.

**Updates**

As each item arrives in the stream, it is prepended to level 0, whose items are stored in `equivalence classes` of width 1. When level 0 becomes full, which is to say it contains \((2 + \varepsilon^{-1})\) items, the two oldest items become an equivalence class of width 2, which is promoted to level 1. Similarly, when level \( l \) acquires \((2 + \varepsilon^{-1})\) equivalence classes, its two oldest equivalence classes are merged, and become the newest equivalence class at level \( l + 1 \). At the top level, when an equivalence class `falls off the window`, it is (conceptually) deleted.

**4 Upper bound: level = sliding dictionary**

The preceding section presents an overview of the structure of \texttt{HistoricalMembership}. We now turn to the question of how to store and maintain this equivalence class partition of the window. An initial temptation would be to store each equivalence class as a (succinct) static dictionary. Periodically, the dictionary structures can be merged, and techniques are available to do this [3]. However, this leads to expensive worst-case queries, taking \( \Omega(\varepsilon^{-1}L) \) time, in which every dictionary in each level is queried, level by level.

**Levels as sliding windows**

To reduce the number of internal queries to the dictionary primitive, we observe that every level in fact constitutes a window partitioned into blocks of fixed length. Thus, we can engage a \texttt{HashSM} sliding dictionary to store an entire level of \texttt{HistoricalMembership}. The problem, for levels \( l \in \{0, 1, \ldots, (L-1)\} \), reduces to sliding membership with window length \( W_l = \varepsilon^{-1}2^l \) and slack \( D_l = 2^l \). Following the approach of \texttt{OptimalSM}, it suffices to divide the window into \( \varepsilon^{-1} \) blocks of width \( 2^l \) and store (item-signature, block ID) pairs in a hash table. At level \( L \), extended to contain the excess of the level structure, reduced to sliding membership with window length \( W_L = E + \varepsilon^{-1}2^L \) and slack \( D_L = 2^L \), where \( E \) is defined in relation (3). Block IDs can be assigned in a circular fashion, and, in our context, are identical to an equivalence class ID. Items that belong to an expired equivalence class are promoted lazily; the item may sit expired at a level \( l \), but it is understood to conceptually belong at level \( l + 1 \). item-signatures must be stored under a representation that is invertible, as is the case for \texttt{BackyardCuckoo} hashing.
Recency queries with succinct representation

![Figure 2 Image of levels 0 & 1 of HistoricalMembership, for $\varepsilon^{-1} = 2$ and $L \geq 2$, on the sequence $S = (\ldots, \bullet, \bullet, \bullet, \bullet, \bullet, \bullet, \bullet, \bullet)$. The global clock is at $t = 18$. Each level is a hash table that stores (item-signature, equivalence class ID) pairs and implements a sliding dictionary. Each level has $\varepsilon^{-1} + 1 = 3$ equivalence classes. The circular field of block IDs is $2(\varepsilon^{-1} + 1) = 6$. The contemporary block ID at level 0 is 0 and the contemporary block ID at level 1 is 3. The equivalence class width at level 1 is $2^1 = 2$. Note, $\bullet$ occurs twice at level 1 and does not appear in block 1. The item $\bullet$ has expired at level 1 and awaits (a lazy) promotion to level 2.

Insertion operation

An insertion into a level proceeds according to the logic of HashSM. Following a protocol of lazy promotions, if, in the process of allocating a free cell to an (item-signature, equivalence class) pair, the procedure encounters an item with a lapsed equivalence class, that is, a class that has been conceptually merged and placed in the succeeding level, it evicts the expired item and inserts it into the subsequent level. After the insertion pair has been allocated a free cell, the process scans a constant number of cells and promotes expired items. For levels $l \in [L - 1]$ block IDs come from the range $\{0, 1, 2, \ldots, 2(\varepsilon^{-1} + 1)\}$, treated as a circular field. For level $L$, block IDs come from

$$2(W_L/2^L + 1) = 2(\varepsilon^{-1} + E/2^L + 1).$$

(4)

In the HashSM approach of lazy deletions there is no control over when an expired item is evicted from a level. Rather, this is an outcome of the randomness that distributes items within the hash table. Therefore, care needs to be taken, during a promotion, when inserting into the succeeding level such that the item is placed in the correct equivalence class. To accomplish this, we make modify the approach of HashSM, specifically, installing an external assignment of block IDs.

Block ID assignment

In HashSM, during an insertion, the procedure assigns the item to the ‘contemporary’ block. With the combination of the levelled structure of HistoricalMembership and lazy promotions, an item evicted from level $l$ may not belong to the contemporary equivalence class (block) of level $l + 1$. Therefore, equivalence class IDs are assigned externally. For example, if, at level $l$, an evicted item returns the (expired) equivalence class ID $e$ from the circular field $2(\varepsilon^{-1} + 1)$, the corresponding class ID in level $l + 1$ can be calculated as follows. Letting $e_j$ name the contemporary class ID at level $j$, the number of equivalence classes between class $e$ and the local (level-$l$) window, assuming $l + 1 \neq L$, is

$$d = e_j - e - (\varepsilon^{-1} + 1) \mod (2\varepsilon^{-1} + 2).$$

The difference is equal to the contemporary class ID minus the expired ID minus the number of active classes modulo the circular field. The difference $d$ determines the equivalence class.

---

5 Hash tables such as Cuckoo and BackyardCuckoo offer multiple choices for each item allocation.
ID $e^*$ that $e$ would be merged into in level $l+1$. Recall that adjacent equivalence classes are (conceptually) merged prior to a promotion to a succeeding level. Therefore, the number of classes between $e^*$ and $e_{l+1}$ is $d/2$, and hence $e^* = e_{l+1} + d/2 \mod (2^{-1} + 2)$.

Each sliding dictionary is synchronized by a global clock, thus, the contemporary class IDs are also updated externally. For this reason, if there are no items to insert at a level, HistoricalMembership still scans for evictions as if an item were being inserted. A formal summary of the insertion procedure is available in Algorithm 2. The initialization algorithm is present in Algorithm 1. Together, they instruct the external assignment of block IDs for the sliding dictionaries. An image of HistoricalMembership is in Figure 2.

**Correctness/queries**

To evaluate the recency of an item $x$, HistoricalMembership probes the levels sequentially. Thus, assuming $x$ is in the window, query time is proportional to the logarithm of $x$’s recency. If $x$ is (first) retrieved from level $l$, its recency is estimated from its equivalence class ID, $e$. The level $l$ dictionary provides a local estimate $\hat{r}_l(x)$ with absolute error at most $2^l - 1$:

$$\hat{r}_l(x) = 2^l \cdot (e - e_l \mod (2^{-1} + 2)).$$

The local estimate can be interpreted as the recency of the item with respect to level $l$. To construct a global estimate $\hat{r}(x, t)$, with bounded relative error, we accumulate the widths of levels below level $l$ and append the sum to the local estimate. Due to the effect of slack, the width of each level varies. (The width of a level is the bound on the number of active items. Each level has $e^{-1}$ non-contemporary active classes.) At time $t$ in the stream, the contemporary class at level $i$ has at most $(t \mod 2^i)$ items, so the width of level $i$ is $(t \mod 2^i) + e^{-1}2^i$. Summing the widths of the preceding levels, we arrive at an estimate.

$$\hat{r}(x, t) = \hat{r}_l(x) + \sum_{i=0}^{l-1} (e^{-1}2^i + (t \mod 2^i)).$$

With the query algorithm in place, we can bound the relative error of the data structure.

**Lemma 4.** HistoricalMembership returns a $(1 + \varepsilon)$-approximation to item recency.

**Proof.** The second summand can be interpreted as the distance between level $l$ and the front of the sequence. The distance is exact. Thus, the error on the global estimate is bound by the error on the local estimate.

$$\frac{|r(x, t) - \hat{r}(x, t)|}{\hat{r}(x, t)} \leq \frac{2^l - 1}{\hat{r}_l(x) + \sum_{i=0}^{l-1} (e^{-1}2^i + (t \mod 2^i))} \leq \frac{2^l - 1}{e^{-1} \sum_{i=0}^{l-1} 2^i} = \varepsilon.$$

**5 Efficiency: selecting the hash tables**

The efficiency of HistoricalMembership hinges on the performance of the hash tables supporting the HashSM sliding dictionaries. BackyardCuckoo is the state-of-the-art, combining succinct representation with worst-case constant-time operations, except with probability proportional to $1/poly(s)$, where $s$ is the size of the set, in the size of the underlying set. Unfortunately, this is an issue for levels with low cardinality, where the failure probability is non-negligible, particularly across long sequences. Thus our choice tables for implementing HashSM depends on the size of the level. We split the level structure in half and only assign the BackyardCuckoo...
Recency queries with succinct representation

Table to the upper $\lceil L/2 \rceil$ levels. As the lower $\lceil L/2 \rceil - 1$ levels have an aggregated cardinality of $O((\varepsilon^{-1} + 1)W^{1/2})$, there is more flexibility in the hash table construction, in the sense that it does not need to be succinct. A number of options are available and we suggest the dynamic hash table of Dietzfelbinger et al. [4] (DynamicTable). The latter result allows a dynamic set of at most $m$ items to be stored in $O(m)$ words with constant worst-case query times and constant expected amortized insert and delete.

Lemma 5. In HistoricalMembership, a query can be evaluated in worst-case $O(\log(\varepsilon W))$ time and insertions completed in expected amortized $O(\log(\varepsilon W))$ time with probability $1 - O(1/W)$.

Proof. A query requires at most $L$ probes to the underlying hash tables. For DynamicTable, queries are worst-case constant. For BackyardCuckoo, when representing a set of size $m$, queries are non constant with arbitrarily small probability $1/poly(m)$. As the width of each level $l \geq \lceil L/2 \rceil$ is $\Omega(W^{1/2})$, we can initialize the BackyardCuckoo hash tables with sufficient randomness such that failures occur with probability $O(1/W^2)$. Taking a union bound over the event that each level reports sliding membership in constant time, queries are worst-case $O(\log(\varepsilon W))$ with probability $1 - O(1/W)$. Similarly, as an insertion causes at most $O(1)$ item insertions at every level, the same argument follows for the insertion time for HistoricalMembership. However, due to inheritance from the DynamicTable, insertion times are only expected amortized.

Further, as the levels are queried consecutively and in reverse chronological order, we can bound the query time as a function of recency.

Lemma 6. In HistoricalMembership, the query cost for an item $x \in [N]$, with recency $r(x,t) \leq W$, can be evaluated in worst-case $O(\log(\varepsilon \cdot r(x,t)))$ time with probability $1 - O(1/W)$.

To bound the memory allocation, we disaggregate the level structure into three components, and work bottom up: the lower levels, implemented by DynamicTable; the upper-middle levels, implemented by BackyardCuckoo; and the top level, which contains the excess.

Lemma 7. HistoricalMembership stores levels $\{0, 1, \ldots, \lceil L/2 \rceil - 1\}$ in $(1 + \varepsilon)\sqrt{W} \cdot O(w)$ bits.

Proof. Each lower level $l$ has cardinality at most $(\varepsilon^{-1} + 1)2^l$. Accumulated across all lower levels the cardinality is at most $(1 + \varepsilon)\sqrt{W}$. The DynamicTable stores $m$ via item-signatures in $O(m\log w)$ bits, which in total leads to a bound of $(1 + \varepsilon)\sqrt{W} \cdot O(w)$ bits.

Lemma 8. For levels $l \in \{\lceil L/2 \rceil, \ldots, (L-1)\}$, the memory allocation of the sliding dictionaries accumulates to $(1 + o(1))(\varepsilon^{-1} + 1)(2^L - 1)(\log (\frac{N}{W}) + \log \varepsilon^{-1} + O(1))$ bits.

Proof. By Theorem 2, BackyardCuckoo stores a set of size $m$, from the universe $[N]$, with each item containing auxiliary information of $K$ bits, in $(1 + o(1))(Bm + mK)$ bits. As level $l$ has cardinality at most $(\varepsilon^{-1} + 1)2^l$ and stores auxiliary information of $\log(2\varepsilon^{-1} + 2)$ bits for the equivalence class ID at that level, we can accumulate the memory commitment across the relevant levels. To set this up, we observe that

$$W \leq \varepsilon^{-1}2^{L+2},$$

(6)
and that
\[ \sum_{l=[L/2]}^{L-1} 2^l(L-l) \leq \sum_{l=0}^{L-1} 2^l = 2^{L+1} - 2. \tag{7} \]
Hence
\[ \sum_{l=[L/2]}^{L-1} (1 + o(1))(B(N, 2^l(\varepsilon^{-1} + 1)) + 2^l(\varepsilon^{-1} + 1) \log(2\varepsilon^{-1} + 2)) \]
\[ = (1 + o(1)) \sum_{l=[L/2]}^{L-1} 2^l(\varepsilon^{-1} + 1)(\log \left( \frac{N}{2^l(\varepsilon^{-1} + 1)} \right) + \log \varepsilon^{-1} + \mathcal{O}(1)) \]
\[ \leq (1 + o(1))(\varepsilon^{-1} + 1) \sum_{l=[L/2]}^{L-1} 2^l((\log \left( \frac{W}{2^l(\varepsilon^{-1} + 1)} \right) + \log \left( \frac{N}{W} \right)) + \log \varepsilon^{-1} + \mathcal{O}(1)) \]
\[ \leq (1 + o(1))(\varepsilon^{-1} + 1)(2^L - 1)(2 + \log \left( \frac{N}{W} \right)) + \log \varepsilon^{-1} + \mathcal{O}(1)) \] from (6),
\[ \leq (1 + o(1))(\varepsilon^{-1} + 1)(2^L - 1)(2 + \log \left( \frac{N}{W} \right)) + \log \varepsilon^{-1} + \mathcal{O}(1)) \] from (7).

The excess of the level structure has \( E = W - \varepsilon^{-1} \sum_{l=0}^{L} 2^l \) items. We extend the top level to contain the excess, becoming a sliding dictionary with \( W_L = E + \varepsilon^{-1}2^L \) and slack \( 2^L \).

Lemma 9. Level \( L \) occupies \( (1 + o(1))(W_L + 2^L)(\log \left( \frac{N}{W} \right)) + \log \varepsilon^{-1} + \mathcal{O}(1) \) bits.

Proof. The BackyardCuckoo table that implements the level-\( L \) sliding dictionary is initialised to contain \( W_L + 2^L \) items. We begin by providing a lower bound on this value.

\[ W_L + 2^L = (W - \varepsilon^{-1}(2^{L+1} - 1)) + \varepsilon^{-1}2^L + 2^L \]
\[ = W - \varepsilon^{-1}2^L + \varepsilon^{-1} + 2^L \]
\[ \geq W - \varepsilon^{-1}2^L \]
\[ \geq \frac{W}{2}, \] from inequality (6). We need to account for the cost of storing a block ID, and size of the level-L circular field is \( 2(\varepsilon^{-1} + E/2^L + 1) \) by Equation (4). Therefore, following Theorem 2 and Equation (1), the number of bit the level-L sliding dictionary requires is

\[ (1 + o(1))(B(N, W_L + 2^L) + (W_L + 2^L) \log(2(\varepsilon^{-1} + E/2^L + 1))) \]
\[ = (1 + o(1))(W_L + 2^L)(\log \left( \frac{N}{W_L + 2^L} \right) + \log \left( \varepsilon^{-1} + \frac{W - \varepsilon^{-1}(2^{L+1} - 1)}{2^L} \right)) + \mathcal{O}(1)) \]
\[ \leq (1 + o(1))(W_L + 2^L)(\log \left( \frac{N}{W} \right) + \log \left( \varepsilon^{-1} + \frac{\varepsilon^{-1}2^{L+2} - \varepsilon^{-1}2^{L+1}}{2^L} \right)) + \mathcal{O}(1)) \]
\[ = (1 + o(1))(W_L + 2^L)(\log \left( \frac{N}{W} \right) + \log \varepsilon^{-1} + \mathcal{O}(1)) , \]
where line three follows from inequalities (6) and (8).

We have now bounded the space consumption of each of the three components of HistoricalMembership. We now relate the overall space consumption to the information-theoretic lower bound.
Recency queries with succinct representation

Lemma 10. HistoricalMembership requires \((1 + o(1))(1 + \epsilon)(B + W \log \epsilon^{-1})\) bits of memory, where \(B\) is the information-theoretic lower bound to store a set of size \(W\).

Proof. The bound in Lemma 8 is at least \((W/2)(O(1) + \log \epsilon^{-1})\). This absorbs the bound of the lower levels, of Lemma 7, under the assumption \(w = o(\sqrt{W}(1 + \log \epsilon^{-1}))\). Therefore, it suffices to focus our attention on the upper levels. By Lemmas 8 and 9, levels \([L/2]\) to \(L\) can be stored in

\[
(W_L + 2^L + (\epsilon^{-1} + 1)(2^L - 1))(1 + o(1))(\log \left( \frac{N}{W} \right) + \log \epsilon^{-1} + O(1))
\]

bits. It suffices to simplify the leading factor.

\[
W_L + 2^L + (\epsilon^{-1} + 1)(2^L - 1) = W - \epsilon^{-1}(2^L+1 - 1) + \epsilon^{-1}2^L + 2^L + (\epsilon^{-1} + 1)(2^L - 1)
\]

\[
= W + 2^{L+1} - 1
\]

\[
\leq W + \epsilon W.
\]

As \(B = W(\log(N/W) + O(1))\), expression (9) simplifies to \((1 + o(1))(1 + \epsilon)(B + W \log \epsilon^{-1})\). ◀

Combining Lemmas 4, 5 and 10, we arrive at Theorem 1, in some sense, a remarkable result. HistoricalMembership achieves a representation of a window of a sequence of items with memory allocation that is tight, with respect to membership on the window, and supports Recency queries in time logarithmic in the recency value. This evolution from OptimalSM to HistoricalMembership represents a time-accuracy trade-off, where we refine our understanding of where items occur in the sequence paying for a small time overhead. Further, the movement between ExactCuckoo and HistoricalMembership represents a space-time trade-off and asks whether constant update and query times are possible in \(o(W \log W)\) space.

We finalize our theoretical development of Recency by observing that Theorem 1 can be applied to approximate set membership. By applying a universal hash function to the sequence of items, we can reduce the size of the universe at the expense of introducing collisions. The result is a space bound that is proportional to \(W\), and tight, up to the first two terms, by Theorem 3.

Corollary 11. For the universal hash function \(h : [N] \rightarrow [(1 + \epsilon)W/\delta]\), on input \(h(S(t)) = (h(s_1), h(s_2), \ldots, h(s_k))\), HistoricalMembership returns the approximate recency on an item with probability \(1 - \delta\). The data structure uses \((1 + \epsilon)W \log(\epsilon^{-1}\delta^{-1}) + O(W)\) bits.

6 Conclusion and future work

We have investigated and defined the notion of Recency through the concept of the sliding window. Existing work, from the sliding membership literature, realizes our definition but cannot accommodate a combination of accuracy and small memory. Our primary innovation is carried through the data structure HistoricalMembership, which supports Recency queries with bounded relative error on top of a succinct representation of the occurred items. The logic of HistoricalMembership is tied to the impression of an equivalence class, wherein items occurring at an equivalent moment in history can be assigned the same estimate. If we think of accuracy-space-time as a triangle, the data structures ExactCuckoo, OptimalSM and HistoricalMembership each occupy a unique face. This leads to the question as to whether operations for Recency data structures can be supported in constant time without sacrificing the memory and accuracy attributes of HistoricalMembership. Alternatively, the three data structures represent the contours and boundaries of the Recency problem.
References


7 Appendix

Algorithm 1 HistoricalMembership

1 Procedure initialise($W, \varepsilon$)
2 
3 $L \leftarrow \lfloor \log(\varepsilon W) \rfloor - 1$; \hspace{1em} // number of levels
4 for $l \in \{0, 1, \ldots, L\}$ do
5 
6 $\Lambda_l \leftarrow$ initialise a hash table that stores at most $2^l (\varepsilon^{-1} + 1)$ items; \hspace{1em} // contemporary equivalence class ID
7 $e_l \leftarrow 0$; \hspace{1em} // circular field of class IDs
8 $n_l \leftarrow 2(\varepsilon^{-1} + 1)$; \hspace{1em} // circular field of class IDs
9 $E \leftarrow W - \varepsilon^{-1} \sum_{l=0}^{L-1} 2^l$; \hspace{1em} // the excess
10 $\Lambda_L \leftarrow$ initialise a hash table that stores at most $2^L (\varepsilon^{-1} + 1) + E$ items;
11 $e_L \leftarrow 0$;
12 $n_l \leftarrow 2(\varepsilon^{-1} + E/2^L + 1)$; \hspace{1em} // circular field of class IDs at level $L$
13 $t \leftarrow 0$; \hspace{1em} // the global clock
14 return;

1 Procedure query($x$)
2 
3 $c \leftarrow 0$;
4 for $l \in \{0, 1, \ldots, L\}$ do
5 
6 $e^* \leftarrow$ retrieve $x$ from $\Lambda_l$;
7 if $e^* \neq -1$ then
8 return $c + 2^l \cdot (e_l - e^* \mod n_l)$; \hspace{1em} // item located at level $l$
9 $c \leftarrow c + \varepsilon^{-1} 2^l + (t \mod 2^l)$;
10 return $-1$;

Algorithm 2 HistoricalMembership

1 Procedure insert($x$)
2 
3 $t \leftarrow t + 1$; \hspace{1em} // update global clock
4 for $l \in \{0, 1, \ldots, L\}$ do
5 
6 if $t \mod 2^l = 0$ then
7 $e_l \leftarrow e_l + 1 \mod n_l$; \hspace{1em} // synchronize block IDs
8 $E \leftarrow$ insert $(x, e_0)$ into $\Lambda_0$; \hspace{1em} // insertion returns a set of evicted items
9 for $l \in \{0, 1, \ldots, L\}$ do
10 
11 $E' \leftarrow \emptyset$;
12 for $(y, e) \in E$ do
13 $e^* \leftarrow e_{l+1} + (e_l - e - (\varepsilon^{-1} + 1) \mod n_l)/2$;
14 $E' \leftarrow E' \cup$ insert $(y, e^*)$ into $\Lambda_l$;
15 $E \leftarrow E' \cup$ scan $\Lambda_l$;
16 return;
Chapter 5

Practical Hierarchical ORAM

We now shift to an application of HistoricalMembership; the data structure exhibited in Section 4. The following work is a manuscript under review¹ at VLDB 2023 and represents a solution to Research Question 3. We started work on this problem when we were looking for concrete applications for HistoricalMembership. An ORAM scheme provides access pattern privacy for remote storage. A sequence of (virtual) accesses to the database constitutes a data stream. An ORAM transforms this stream into a sequence of physical memory accesses that obfuscates any information associated with the input sequence. The process of obfuscation introduces an overhead of additional accesses. We realised that this overhead could be reduced with knowledge of how recently a (virtual) memory address was accessed. However, this metadata introduces a memory burden that may contradict the client’s need to outsource its storage. Thus, with a compact data structure that provides accurate recency queries, we found a compelling application of HistoricalMembership.

Building on the ideas established in Chapter 4, we accomplish two core contributions.

1. We introduce the first practical variant of the Hierarchical ORAM scheme, RankORAM.
2. We reduce client memory against other ORAM schemes with comparable bandwidth and latency performance.

An earlier construction of RankORAM, which is not presented in the submitted manuscript, used a different rebuilding schedule that is adopted in earlier Hierarchical ORAMs [64]. In this instance, we could use HistoricalMembership as a black box with $\varepsilon = 0.5$. However, for ease of presentation, the submitted manuscript adopts a modified, but equivalent, rebuilding schedule used in more recent publications on Hierarchical ORAM [13]. Thus, the client-side data structure presented below is a variation of HistoricalMembership, but still imports the core innovations around the merging of compressed data structures. In addition, we use a different encoding for the level dictionaries to enable support for rank queries.

The experimental component is limited to a proof-of-concept simulation. We do not test or provide a comprehensive evaluation of the full ORAM scheme against its competitors. The experiments could also be extended to the key target application: secure processors. Nonetheless, our evaluation does highlight the power of our result and includes comparison against

¹Submitted July 29 2022.
previously unimplemented approaches for compressing client-side metadata. This comparison, although favourable to RankORAM, additionally confirms the feasibility of these approaches in practice.

This is in my view the most rounded contribution of the thesis. It showcases and combines a medley of concepts from data compression and data security. The final manuscript includes; (1) robust theoretical results; (2) a detailed and nuanced discussion of the literature; and (3) meaningful experiments.
ABSTRACT

Accesses to data stored remotely create a side channel that is known to leak information even if the content is encrypted. Oblivious RAM is a cryptographic primitive that provides confidentiality of access patterns in remote storage settings. To outsource a database of $n$ blocks of $B$ bits, traditional solutions restrict the client to $O(B)$ bits of private memory. A class of solutions, known as Hierarchical ORAM, has achieved theoretically optimal bandwidth performance in this setting. Hierarchical ORAM distributes data blocks at the server across a hierarchy of hash tables, with the most recently accessed blocks in the lower levels of the hierarchy. Locating a block in the hierarchy requires a large number of round-trips of communication, with the server, per virtual access. Furthermore, rebuilding the hierarchy incurs a large concrete bandwidth overhead. Thus, Hierarchical ORAMs are seen as theoretical artefacts and have not been deployed in practice.

For many applications, such as cloud storage, the client can afford a larger, $\omega(B)$-bit, private memory allocation. With this premise, we introduce Rank ORAM, the first practical Hierarchical ORAM that takes advantage of a larger client. We construct a compact client-side data structure that keeps track of how recently data blocks have been accessed. Leveraging this information, Rank ORAM reduces both the number of round-trips of communication and the concrete bandwidth overhead of Hierarchical ORAM. In addition, Rank ORAM achieves a smaller client memory allocation than existing (non-Hierarchical) state-of-the-art practical ORAM schemes while maintaining comparable bandwidth performance. Our experiments on real network file-system traces demonstrate a reduction in client memory, against existing approaches, of a factor of 100. At the same time, client-memory is only 290MB when outsourced a database of 17.5TB.

1 INTRODUCTION

In remote storage settings, encryption cannot protect against all security vulnerabilities. For instance, the order in which a client accesses their outsourced data, known as the access pattern, can leak sensitive information. Access pattern vulnerabilities have been demonstrated in a number of domains. These include, but are not limited to, leakage through page fault patterns in secure processors [36, 39, 41], through SQL query patterns on encrypted outsourced databases [21, 22] and through search patterns, resulting in query recovery attacks, in searchable encryptions [7, 21].

To mitigate access pattern leakage, Goldreich and Ostrovsky introduced the notion of Oblivious RAM (ORAM) [15]. An ORAM scheme transforms a sequence of virtual accesses into a sequence of physical accesses that is independent of the input sequence. This transformation eliminates information leakage in the access trace. Ultimately, an adversary must not be able to distinguish between the access patterns produced by an ORAM on an arbitrary pair of input sequences of the same length.

A trivial ORAM can be constructed by sequentially downloading the full database during each access. Although the access pattern produced by the ORAM is identical for all input sequences of the same length, rendering this scheme oblivious, there is a linear blowup in bandwidth and it is impractical on meaningfully sized databases. Thus, the principal aim is to minimize the bandwidth overhead, that is, the number of additional physical accesses per virtual access, without sacrificing the privacy guarantee. In contrast to the trivial solution, a bandwidth-efficient ORAM, to obfuscate the original access pattern, combines additional dummy accesses and periodic shuffling of the server’s contents.

The original formulation of the ORAM problem [15] has two constraints: (1) client memory is restricted to $O(1)$ data blocks; and (2) server-side computation is forbidden. Under these constraints there exists a logarithmic lower bound for the bandwidth overhead [24] and optimality has been achieved by a class of solutions known as Hierarchical ORAMs (HierarchicalORAM) [2, 4]. The HierarchicalORAM was first introduced by Goldreich et al. [15] and has enjoyed a long line of improvements [8, 16, 17, 23, 25, 31, 33, 40] and variations [5, 12, 26] over the past two decades. However, these constructions have poor concrete bandwidth performance and have not been adopted in practice.

There are many scenarios where neither of the problem constraints are realistic, and a relaxation of the problem has led to the emergence of ORAMs with improved (concrete) bandwidth and latency performance. For example, a small client, that accommodates only $O(1)$ blocks, cannot store information on where a targeted

---

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment. Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097. doi:xx.xx/xxx.xx

1 Throughout this work "concrete" refers to the exact bandwidth cost, unobscured by complexity notation.
block is located at the server. Recall that blocks are periodically shuffled to help obfuscate the access pattern. Thus a large amount of communication between the client and the server is required to locate the block, without revealing this location to the adversary. Increasing the amount of metadata stored at the client can alleviate the interactive components of client-server communication and, consequently, reduce both the bandwidth overhead and the number of round-trips per access [10, 35]. Client memory is therefore engaged in a key trade-off with bandwidth and latency.

In a similar vein, our work seeks to extend HierarchicalORAM beyond its theoretical status while relaxing only the first constraint — allowing client memory larger than $O(1)$ blocks. We construct a novel HierarchicalORAM protocol, RankORAM, that stores additional metadata at the client. The cost of the metadata is small when compared to the size of the outsourced database (e.g., 290MB for a 17.5TB database). The protocol exploits this metadata to improve the concrete bandwidth performance and latency against existing HierarchicalORAM schemes (see Table 1). Importantly, RankORAM is the first HierarchicalORAM that performs single-round access without the help of server-side computation.

**RankORAM.** A HierarchicalORAM distributes the outsourced data across a sequence of oblivious hash tables that increase exponentially in size. More-recently accessed data blocks are located in the smaller levels and less-recently accessed blocks are gradually moved into the larger levels. The efficiency of a HierarchicalORAM rests on the implementation of the oblivious hash table. State-of-the-art oblivious cuckoo hash tables lead to optimality [3, 4].

In Hierarchical ORAM, the recency of a data block, that is, the number of accesses since it was last accessed, determines the level that it resides in. This is due to the deterministic scheduling of the rebuild phase that pushes the least-recently accessed blocks into the larger levels. Further, knowledge of the block recency supports non-interactive accesses to the server, which reduces the number of round-trips of communication [40]. However, storing the recency information for all data blocks is expensive and may be prohibitive in memory-constrained environments, such as secure enclaves. For example, the standard Intel SGX 2 [1] has only 96MB of private memory. This obstacle can be overcome with a recent advance by Holland et al., who demonstrate that approximate recency queries, with bounded relative error, can be supported under a succinct representation of the stored items [20].

Our scheme, RankORAM, utilises these techniques to compress recency information at the client. Consequently, we are able to support low latency, non-interactive queries with a client-memory allocation smaller, in theory and practice, than prior state-of-the-art practical ORAM schemes that utilise client-side metadata [35, 37] (see Table 2).

**Compressed Metadata.** Existing large client-memory ORAMs, such as Partition ORAM [37] (PartitionORAM) and Ring ORAM [35] (RingORAM), use arrays to store various metadata regarding block locations at the server. For a database of $n$ blocks of $B$ bits, an array occupies $O(n \log n)$ bits of client-memory. To reduce this cost it is standard to recursively store the array in a sequence of smaller ORAMs at the server. This comes at the expense of increased bandwidth and latency. In contrast, with RankORAM, we compress block metadata to achieve smaller client-memory without increasing bandwidth or round-trips with the server. Further, as our experiments demonstrate, it results in significant client memory reductions against approaches adopted in prior implementations. Though Stefanov et al. have mentioned the use of compression to reduce the cost of the metadata [37] (albeit not in the context of level information for HierarchicalORAM), they did not provide details of how this approach would work in terms of algorithms or data structures. To this end, we propose a detailed solution for [37], compressedCounters, that we use for comparison when evaluating our work (see Table 3).

**Our contributions.** We present a new ORAM scheme named RankORAM. It belongs to the class of HierarchicalORAM and introduces a new trade-off between client memory and bandwidth. The following theorem captures the performance of RankORAM.

**THEOREM 1.1.** RankORAM is an oblivious RAM that stores a database of $n$ blocks of $B$ bits, requires $O(n + \sqrt{n} \cdot B)$ bits of private memory, performs accesses in a single round-trip and observes an amortized bandwidth overhead of $4 \log n$ blocks.

With this result, our work offers the following contributions.

- RankORAM is the first HierarchicalORAM that achieves both a single round-trip per online access and a low concrete bandwidth overhead. Though, we utilize more client memory than prior approaches, our experiments demonstrate that the memory allocation is feasible even for resource-constrained settings (e.g., 290MB of client memory when outsourcing 17.5TB).
- RankORAM is supported by a novel client-side data structure, historicalMembership, that compresses the location metadata of the blocks at the server. historicalMembership can be used to reduce the number of round-trips per access for any HierarchicalORAM.
- The memory allocation of RankORAM is asymptotically smaller than prior state-of-the-art ORAMs with low concrete bandwidth overhead. Compared to PartitionORAM and RingORAM, we reduce the size of the client by a factor of $O(\log n)$.
- We supplement the work of Stefanov et al. [37] by providing a data structure, named compressedCounters, to compress client-side metadata. The structure works for both PartitionORAM and RingORAM.
- Our experiments, conducted on real network file system traces, demonstrate, for a 4KB block size, a reduction in memory for RankORAM by a factor of 100 against a non-compressed structure and a factor of 10 against compressedCounters. We also experimentally demonstrate the poor worst-case behaviour of compressedCounters.

## 2 RELATED WORK

All the variations of HierarchicalORAM depend on the implementation of the oblivious hash table. The amortized bandwidth overhead of HierarchicalORAM is determined by the cost of the rebuild (offline bandwidth) and the cost of an access (online bandwidth). In the original proposal by Goldreich and Ostrovsky, at level $l$, the hash table contains $2^l$ buckets of $O(\log n)$ depth. When accessing a bucket...
obliviously, a linear scan is performed. The scheme’s (amortized) bandwidth cost of $O(\log^2 n)$ is dominated by the rebuild phase.

Subsequent improvements were achieved by changing the hashing primitive to an oblivious cuckoo hash table [8, 16, 17, 33]. With cuckoo hashing, the lookup time is constant. These schemes incur an amortized $O(\log^2 n / \log \log n)$ bandwidth cost that is dominated by a rebuilding phase, which relies on expensive oblivious sorting\(^2\). Patel et al., with ParORAMa, provide a cuckoo construction algorithm that does not rely on oblivious sorting [31]. They assume that the input to the construction algorithm is randomly shuffled and the bandwidth overhead is reduced to $O(\log n \cdot \log \log n)$ blocks. With OptORAMa, this idea was extended by Asharov et. al to achieve optimal, $O(\log n)$, bandwidth overhead [2], matching the lower bound of Larsen et al. [24].

Chan et al. present a simple two-tier hash table, not based on cuckoo hashing [8]. Observing that, in work prior to theirs, bandwidth is dominated by rebuilds, the authors construct a simpler two-tier table that permits a more expensive lookup query in exchange for a reduced rebuild cost. However, the overall bandwidth cost is not asymptotically improved. All of the above ORAM protocols operate with $O(\log n)$ bits of client memory.

**Single round-trips.** The execution of an ORAM access depends of where the accessed block is located at the server. Locating the block, with no prior knowledge of where it resides, introduces client-server interaction. In small memory, this interactive component can be removed by allowing server-side computation. For example, SR-ORAM [40] and BucketORAM [12] place encrypted Bloom filters at the server to separate membership testing from block storage. The schemes can then build a layered branching program with paths that depend on the location of the accessed item. The server queries each Bloom filter and the output is used to unlock the next step in the correct path through the branching program. The path reveals which blocks to return to the client. In contrast to our work, which relies on a server performing only read and write requests, the server in the above schemes can perform more complex operations on data.

**Larger clients and low concrete bandwidth.** There are many applications in which clients can afford more than $O(B)$ bits of private memory. Under this observation, there are two constructions, PartitionORAM [37] and RingORAM [35], that store metadata, concerning server block locations, explicitly at the client. The metadata is stored in position maps, occupying $\Theta(n \log n)$ bits using an array, and allows accesses to be executed with a single round-trip of communication. Both PartitionORAM and RingORAM achieve state-of-the-art total bandwidth overhead of $3 \log n$, which can be reduced further following optimizations, using server-side computation, proposed by Dautrich et al. [10].

In the context of memory constrained environments, these schemes rely on a technique that recursively stores the metadata in a sequence of smaller ORAMs at the server. This comes at the cost of increased bandwidth and latency.

**Compressing metadata.** The position maps for both PartitionORAM and RingORAM, with multiple types of metadata per block, occupy $\Theta(n \log n)$ bits. As $n$ increases, this term begins to dominate the client memory allocation. To alleviate this burden, Stefanov et al. outline a method to compress the position map [37]. The compression method, compressedCounters, is designed for sequential workloads but has a worst-case memory allocation of $O(n \log n)$ bits. We expand on this work further in Section 8. A summary of the theoretical properties of different client-side data structures is provided in Table 3.

### 3 PRELIMINARIES

Fixing notation, we consider the setting where a client outsources $n$ blocks, each of $B$ bits, to untrusted storage.

#### 3.1 Performance Metrics

For measuring performance we consider three key parameters: **client memory, bandwidth overhead** and the **number of round-trips** (latency). The size of the client memory measures the amount of storage, both temporary and permanent, required to execute an ORAM scheme. The bandwidth overhead refers to the the number of blocks, possibly amortized, exchanged between the client and server per virtual access. It represents the multiplicative overhead of moving from a non-oblivious to an oblivious storage strategy. The number of round trips counts the rounds of communication between the client and server per virtual access.

---

\(^2\)Oblivious sorting in $O(B)$ bits is very expensive in practice and are requird in many ORAM schemes for the rebuilding phase. For a discussion on the trade-offs between client memory and bandwidth, see Holland et al. [19].
3.2 Definitions

Security. We adopt the standard security definition for ORAMs. Intuitively, it states that the adversary should not be able to distinguish between two access patterns of the same length. In other words, the adversary should learn nothing about the access pattern.

**Definition 3.1 (Oblivious RAM [37]).** Let

\[ y := \{ (op_1, a_1, data_1), \ldots, (op_m, a_m, data_m) \} \]

denote a sequence of length \( n \), where \( op_i \) denotes a read(a_i) or write(a_i, data_i). Specifically, a_i denotes the logical address being read or written and data_i denotes the data being written. Let \( A(y) \) denote the (possibly randomized) sequence of accesses to the remote storage given the sequence of data requests \( y \). An ORAM construction is deemed secure if for every two data-request sequences, \( y \) and \( z \), of the same length, their access patterns \( A(y) \) and \( A(z) \) are, except by the client, computationally indistinguishable.

**Oblivious Shuffle.** Oblivious shuffle is a key primitive of oblivious RAM solutions, including RankORAM. It implements the following functionality.

**Definition 3.2 (Functionality: Array Shuffle).** Let \( \mathcal{P} \) denote a set of permutations. On input array \( U \), of key-value pairs, and permutation \( \pi \in \mathcal{P} \), the Array Shuffle outputs the array \( V = shuffle(\pi, U) \), where \( V[i] = (k, v) \) and \( \pi(k) = i \).

We assume that the permutation function, \( \pi \), is given to the algorithm in a form that allows for its efficient evaluation. For example, it could be provided as a seed to a pseudo-random permutation. Oblivious algorithms preserve the input-output behaviour of a functionality and produce an access pattern that is independent of the input. We now define the notion of oblivious algorithm.

**Definition 3.3 (Oblivious Algorithm).** Let \( A(M^\mathcal{F}(x)) \) denote the access pattern produced by an algorithm \( M \) implementing the functionality \( \mathcal{F} \) on input \( x \). The algorithm \( M \) is oblivious if, for every two distinct inputs, \( x_1 \) and \( x_2 \), of the same length, except to the client, their access patterns, \( A(M^\mathcal{F}(x_1)) \) and \( A(M^\mathcal{F}(x_2)) \), respectively, are computationally indistinguishable.

Therefore, an oblivious shuffle implements functionality Definition 3.2 and does not reveal anything about the input permutation through its access pattern.

### 3.3 HierarchicalORAM

The Hierarchical ORAM contains a hierarchy of oblivious hash tables \( T_0, \ldots, T_L \), with \( L = \log n \). In the words of Goldreich and Ostrovsky [15], the ORAM consists of "a hierarchy of buffers of different sizes, where essentially we are going to access and shuffle buffers with frequency inversely proportional to their sizes". The hash table abstraction contains a look-up query and a construction algorithm. For the construction algorithm to be oblivious, by Definition 3.3, the input blocks must be placed in the table without leaking their locations through the access pattern.

A general HierarchicalORAM has the following structure. The hash table \( T_l \) stores \( 2^l \) data blocks. Next to each table, a flag is stored to indicate whether the hash table is full or empty. When receiving a request to an address, the ORAM operation involves both an access and rebuild phase:

1. **Access:** Access all non-empty hash tables in order and perform a lookup for address \( x \). If the item is found in some level \( l \), perform dummy lookups in the non-empty tables of \( T_{l+1}, \ldots, T_L \). If the operation is a read, then store the found data and place the block in \( T_0 \). If the operation is a write, ignore the associated data and update the block with the fresh value.

2. **Rebuild:** Find the smallest empty hash table \( T_l \) (if no such level exists, then set \( l = 0 \)). Merge the accessed item and all of \( \{T_j\}_{j \leq l} \) into \( T_l \). Mark levels \( T_0, \ldots, T_{l-1} \) as empty.

A block is never accessed twice in the same hash table in between rebuilds at a given level. This invariant is crucial to security of the scheme. As each block is always retrieved from a different location, the sequence of hash table probes produced by HierarchicalORAM appears random to an adversary.

Note that access is interactive since the client does not know which level a block belongs to. That is, the client has to query the levels sequentially until the target block is found. This requires a round trip per level and increases the latency of the protocol. Our client-side data structure is designed to remove this cost.

Our protocol adopts the HierarchicalORAM template. An instance depends on the choice of hash table and prior work has

---

<table>
<thead>
<tr>
<th>SR-ORAM [40]</th>
<th>( O(B) )</th>
<th>( O(\log n) )</th>
<th>✓</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWORAM [14]</td>
<td>( O(\log n) \cdot (1 + B) )</td>
<td>( O(\log n) )</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BucketORAM [12]</td>
<td>( O(\log n) \cdot (1 + B) )</td>
<td>( O(\log n) )</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PathORAM [38]</td>
<td>( O(\log n) \cdot (1 + B) )</td>
<td>8 ( \log n )</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RingORAM [35]</td>
<td>( O(n \log n + \sqrt{n} \cdot B) )</td>
<td>3 ( \log n )</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PartitionORAM [37]</td>
<td>( O(n \log n + \sqrt{n} \cdot B) )</td>
<td>3 ( \log n )</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RankORAM</td>
<td>( O(n + \sqrt{n} \cdot B) )</td>
<td>4 ( \log n )</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 2:** Comparison of ORAM schemes with low concrete bandwidth and/or a single round-trip of communication per access. Compared to state-of-the-art approaches, RankORAM reduces client memory by a factor of \( O(\log n) \).

<table>
<thead>
<tr>
<th>array [9, 10, 35, 37]</th>
<th>memory</th>
<th>update time</th>
</tr>
</thead>
<tbody>
<tr>
<td>compressedCounters [37]</td>
<td>( O(n \log n) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>historicalMembership</td>
<td>( O(n) )</td>
<td>( O(\log^2 n) )</td>
</tr>
</tbody>
</table>

**Table 3:** Comparison of client side data structures.
proposed a number of different constructions. The state-of-the-art HierarchicalORAM uses oblivious Cuckoo hash tables [2, 4]. They are theoretically optimal but the concrete bandwidth is too high to be used in practice.

3.4 Rank data structures
Our client-side data structure is built on set-membership structures that support the following operations on the set \( \Phi \):

\[
\Phi.\text{index}(r) = \text{return the } r^{\text{th}} \text{ smallest address in } \Phi \tag{1}
\]

\[
\Phi.\text{rank}(x) = \{ y \mid y \in \Phi, y < x \} \tag{2}
\]

A data structure that supports these operations, in addition to a membership query (\( x \in \Phi \)), is called an indexed dictionary [34]. For example, for the set \( \Phi = \{2, 5, 7, 9\} \), the functions evaluate as \( \Phi.\text{rank}(7) = 3 \) and \( \Phi.\text{index}(3) = 7 \).

4 HISTORICAL MEMBERSHIP
The key ingredient of our RankORAM solution is a client-side data structure, HistoricalMembership, that contains information about the location of the blocks at the server. This information helps us retrieve the required block from the server in a single round of communication. In this section we describe HistoricalMembership.

Let \( \Phi_l = \{ \sigma \mid (\sigma, \_ \_ ) \in T_l \} \) denote the set of logical addresses at level \( l \). HistoricalMembership maintains each set \( \Phi_l \) for \( l \in [L] \), in a compressed indexed dictionary (see Section 3.4). As access patterns typically have low entropy, for example, access patterns in file systems are highly sequential [30], the sets are compressible. Regardless, our choice of encoding has good worst-case behaviour, as we demonstrate below.

The collection of dictionaries, \( \Phi \), supports the following two functions:

\[
\text{level}(x) = \min\{ i \mid (x, \_ \_ ) \in T_i, i \in \{1, \ldots, L\} \}, \tag{3}
\]

\[
\text{position}(x) = \begin{cases} 
\Phi_{\text{level} (x) } . \text{rank}(x) & \text{if level}(x) < L \smallskip \\
x & \text{if level}(x) = L 
\end{cases} \tag{4}
\]

The function level denotes the level a block belongs to and is calculated in a sequence of membership queries starting from \( \Phi_0 \). The function position(\( x \)) denotes the rank of address \( x \) within its current level. The level functionality is observed in some prior work [35, 37], where implemented in an array (called the position map) mapping element addresses to a collection of auxiliary information. The position functionality is novel to this work and is used to map addresses into hash table locations.

When a merge happens at the server (\( T_l \leftarrow \bigcup_{i=1}^{l-1} T_i \)), the client-side structure is updated accordingly:

\[
\forall i \in \{1, \ldots, L - 1\} : \Phi_i \leftarrow \emptyset \tag{5}
\]

\[
\Phi_l \leftarrow \bigcup_{i=1}^{l-1} \Phi_i. \tag{6}
\]

Thus, one requirement of our encoding is that it supports efficient merging. Further, as each element belongs to exactly one level, we do not need to store \( \Phi_L \). To evaluate Equation (3), if the element is not a member of \( \{\Phi_L\}_{i=L-1} \), it must be a member of \( \Phi_L \). Further, we do not need rank information at level \( L \). This is covered in more detail in Subsection 5.2.

4.1 Run length encoding
A simple compact representation of a set is run length encoding [6]. This representation compiles the set \( S = \{x_1, x_2, \ldots, x_n\} \), where \( x_i < x_j \) for all pairs \( i < j \), as the string:

\[
\text{rle}(S) = (x_2 - x_1) \circ (x_3 - x_2) \circ \cdots \circ (x_n - x_{n-1}) \circ (x_{n+1} - x_n),
\]

where \( x_{n+1} = n \). Each sub-string can be encoded with a prefix-free Elias code [11]. As the latter encodes the integer \( x \) in \( O(\log x) \) bits, a level \( l \) dictionary requires:

\[
\sum_{i=2}^{n_l+1} O(\log(x_i - x_{i-1})) \leq \sum_{i=2}^{n_l+1} O(\log n/n_l)) \tag{5}
\]

\[
= O(n_l \log(n/n_l))
\]

bits. The first inequality comes from the observation that \( \log \) is a convex function. There are dictionaries that are more (space) efficient than this, such as those referred to as “succinct” [18]. However, run-length codes have the advantage of being easily mergeable: given encodings of the sets \( S_1 \) and \( S_2 \), one can enumerate \( S_1 \cup S_2 \), in order, in \( O(|S_1| + |S_2|) \) time and with a working space of \( O(1) \) words.

4.2 Auxiliary data structure
The efficiency of the level (Equation (3)) and position (Equation (4)) functions depend, respectively, on the efficiencies of the membership and rank queries on the component dictionaries. To support the latter efficiently, we supplement each code with an auxiliary structure of forward pointers that is constructed as follows. We divide the run-length code \( \text{rle}(\Phi_l) \) into \( O(|\Phi_l|/\log n) \) segments of equal cardinality of order \( \Theta(\log n) \). The auxiliary structure provides forward access to the beginning of each segment. Each pointer is stored with the starting address of the segment. This allows for a fast identification of the correct segment.

**Query algorithms.** To execute a fast membership query, the procedure performs a binary search on the element keys in the auxiliary structure to realize the correct neighborhood, then jumps to the corresponding segment of the run-length code in a single probe. The query is completed with a linear scan of the segment. As the segments have equal cardinality, rank queries can be computed in a similar fashion if we keep track of the index of the segment during the binary search on the auxiliary structure.

Index queries are computed as follows. Let \( W = \Theta(\log n) \) denote the width of the segments. For \( \Phi_l.\text{index}(i) \), the procedure first identifies the correct segment as \( j = [i/W] \). It then performs a local index query on segment \( j \) for input \( i' = i - j \cdot W \). The latter can be executed on a linear scan of the segment.

**Performance.** With the auxiliary structure established, we conclude by evaluating the performance of historicalMembership. All proofs are in the full version of the paper.

**Lemma 4.1.** historicalMembership, compressed with run-length codes, takes up \( O(n) \) bits.

\begin{footnote}{That is, we do need to uncompress the sets into client memory.} \end{footnote}
Lemma 4.2. To support a database of size \( n \), historicalMembership admits position and level queries in \( O(\log^2 n) \) time in the worst-case and provides updates in \( O(\log^2 n) \) amortized time.

The times are fast when compared to standard network latencies.

5 RANKORAM

Our goal is to construct a Hierarchical ORAM that (1) incurs a single round-trip per online access and (2) obtains a low concrete bandwidth overhead comparable to prior state-of-the-art. Both of these are achieved with the support of our historicalMembership data structure introduced in Section 4. RankORAM follows the HierarchicalORAM template outlined in Section 3.3. Invoking historicalMembership membership reduces communication and our oblivious hash table used to instantiate the template innovates thus:

1. To achieve a single online round-trip, we need to build a non-interactive query object to send to the server. historicalMembership encodes the level information, which tells us, a priori, where to retrieve the accessed block and where to retrieve dummies. Thus, all the hash table probes can be batched in a single request to the server.

2. To build an efficient oblivious hash table, we leverage the \( \text{rank} \) information provided by historicalMembership. This allows for a simple construction, with table locations generated by the position function (see Equation (4)), that dispenses with the (expensive) cuckoo hash table.

5.1 Setup

The client stores historicalMembership in private memory and requires \( O(\sqrt{n} \cdot B) \) bits of temporary storage to perform the oblivious shuffling required for rebuilds. With historicalMembership, the client can compute the level and position functions in private memory. The server stores the data blocks in a sequence of levels as specified by the HierarchicalORAM template. Similar to the client, the server requires temporary storage, of size proportional to the database, to support rebuilds.

The server is initialized with all data in level \( L \). During rebuilds, fresh permutation functions are generated using a fresh seed to a family of pseudorandom permutation functions. A fresh dummy block is constructed by encrypting an empty payload with information regarding the location of the dummy at the server. For example, a dummy at level \( l \) could be constructed as \( \text{enc}(i + t_l) \), where \( i \) denotes the hash table location and \( t_l \) denotes how many rebuilds have occurred at level \( l \). Thus all dummies appear unique and are indistinguishable, to the adversary, from real blocks.

5.2 Permuted Array as Oblivious Hash Table

RankORAM implements the level-wise hash table using a permuted array. Fixing notation, let \( T_l \) refer to the array, located at the server, that stores the elements of level \( l \) and let \( n_l := 2^l \). The array has length \( 2n_l \) and stores at most \( n_l \) real elements and, consequently, at least \( n_l \) dummy elements. For levels \( l < L \), we utilize the \( \text{rank} \) information to assign each element in level \( l \) to a unique index in the domain \( [n_l] \). The \( \text{rank} \) information is available through the indexed dictionaries that form historicalMembership. Elements are then permuted, according to the permutation \( \pi_l \), using their ranks:

\[
T_l[\pi_l(\Phi_l, \text{rank}(a))] = (a, \_).
\] (6)

For level \( L \), we do not need to worry about mapping the address space onto a smaller domain and we can proceed by simply permuting the address space:

\[
T_L[\pi_L(a)] = (a, \_).
\] (7)

Thus, each block can be retrieved with the \text{position} function. When a level is rebuilt, a new permutation function is generated with fresh random bits.

PartitionORAM places blocks in \( \sqrt{n} \) small Hierarchical ORAMs of size \( \approx \sqrt{n} \). There are some similarities here to the component ORAMs of the PartitionORAM, which also utilize permuted arrays. The key difference between our setup and that of partitioning is the size of the levels. Partitioning produces smaller levels. This allows shuffling to be performed exclusively at the client (the whole level is downloaded and permuted locally). In contrast, as the largest level of RankORAM has size \( n \), we require an interactive shuffling algorithm. The advantage of our approach is that, with the rank information compactly encoded in \( \Phi \), we can access blocks at the server directly through \( \pi_l \) (see Equation (6)). In other words, we are able to map the set \( \Phi_l \) onto the domain of \( \pi_l \) without collisions. Without rank information, PartitionORAM is required to store the offsets explicitly. Through our experiments, we demonstrate that our approach leads to significant savings in client memory.

5.3 Access Phase

The access algorithm, formalized in Algorithm 1, is similar to that of prior Hierarchical ORAMs. The primary difference is that the procedure begins by determining the block metadata (the level and \text{position} of the data block) stored at the client. The \text{position} function is used to find the index of the block within its level (line 7). With the level information, we know, a priori, which level the
Algorithm 1: Rank ORAM access. Lines highlighted in red represent operations performed at the server.

1. define access(a, op, data')
2. I ← level(a) // retrieve position information
3. r ← position(a)
4. initialize empty query Q // create a batch query to send to the server
5. for i ∈ occupiedLevels do
   6. if i = l then
      7. Q ← Q ∪ (i, π_l(r))
   8. else
      9. Q ← Q ∪ (i, π_r(dummyCntr_i))
     10. dummyCntr_i ← dummyCntr_i + 1
8. send Q to the server
9. initialise empty output O // server executes Q
10. for i ∈ occupiedLevels do
   11. retrieve (i, index) from Q
   12. O ← O ∪ Φ_l[|index]
13. send O to the client // returns result to client
14. (a, data) ← retrieve and decrypt block a from O
15. // unpack output from the server
16. if op = write then
   17. data ← data'
18. T₀ ← encrypt(a, data)
19. Φ₀ ← a
20. count ← count + 1 mod 2^l
21. Rebuild()
22. return data

Algorithm 2: Rank ORAM Rebuild

1. define rebuild()
2. l ← msb(count) + 1
3. π_l ← a pseudo random permutation on the domain [2^l]
4. Φ_l ← Φ₀ ∪ · · · ∪ Φ_{l−1}
5. I ← evict(0) || · · · || evict(l−1)
6. I ← I || [n] dummy blocks
7. T_l ← shuffle(π_l(Φ_l.rank(·)), I)
8. for i ∈ {0, ..., l − 1} do
   9. Φ_i ← ∅
10. dummyCntr_i ← Φ_i
11. occupiedLevels ← {l} ∪ occupiedLevels \ {0, ..., l − 1}
12. return

A dummy counter ensures that the scheme only returns untouched blocks. When a dummy is retrieved (line 9), we increment the counter (line 10) so that an untouched dummy is retrieved at the next dummy request. An example of the access procedure is available in Figure 1.

5.4 Rebuild Phase

Recall that rebuilds enforce the invariant that a block is never retrieved, more than once, from a given hash table instance. This ensures that the sequence of hash table probes appears random to the adversary. The rebuild carries out the invariant by periodically moving blocks up the hierarchy into fresh hash-table instances. Consequently, a block at level l < L has recency less than 2 · 2^l.

Given that the hash tables are implemented as permuted arrays, the rebuild function is straightforward. We first update historical Membership and rearrange the server’s memory accordingly. For a rebuild into level l, we merge the compressed dictionaries in levels {0, ..., (l − 1)} to obtain the dictionary Φ_l. The rebuild at the server involves a single oblivious shuffle. The input array is the concatenation of the untouched blocks (including dummies) in levels {0, ..., (l − 1)}. A evict (Algorithm 3) is executed to construct the array of untouched elements. The input array is padded with dummy blocks to the width of the output array. The dummy blocks are indexed so that their encryptions are indistinguishable from real blocks. We generate a new pseudo-random permutation, π_l, (line 2) and the input permutation for the oblivious shuffle is the composition of functions π_l ◦ Φ_l.rank. Any oblivious shuffle algorithm can be used here [29, 32]. (However, to reduce constants, we provide a shuffle optimized for RankORAM in the next section.)

The procedure concludes by updating some client-side data; the dictionaries at levels {0, ..., (l − 1)} are deleted; the dummy counter is set for level l; and the set of non-empty (or occupied) levels is adjusted (line 10). The rebuild procedure is formalized in Algorithm 2. The function msb (line 2) computes the most significant bit of the input.

5.5 Eviction procedure

Hash table eviction (used in the rebuild phase) involves removing all untouched elements from the hash table chronologically, from lowest index to highest index. This procedure does not have to be done in an oblivious manner and can be carried out by the server. For completeness, we provide two efficient ways for the client to perform eviction. We could store a bitmap, locally at the client, that indicates the untouched indices. This approach requires an additional 4n bits but does note impact our asymptotic result. Otherwise, we can use the inverse permutation function, π_l−1 for level l, to enumerate the ranks of the elements in the correct order. The rank can be used to determine if the corresponding element is real or dummy; an element with a rank larger than the cardinality of the level is a dummy block. We can then determine if it is a touched block. If it is real, and also belongs to a lower level, then it is touched. If it is a dummy and the rank is lower than dummyCntr, then it is touched. We skip over touched elements and only retrieve untouched elements. Finally we note that we could avoid eviction by requesting the server delete every block retrieved in line 14 of Algorithm 1.
6 SHUFFLING FOR RANKORAM

When implementing shuffle (line 7 of Algorithm 2) RankORAM can invoke any oblivious shuffle [29, 32], out of the box. However, in our context, we have dummy blocks in both the input and output arrays. A given dummy block can be placed in any vacant location in the output array: we can exploit this fact to gain performance improvements relative to general shuffling algorithms. Therefore, we construct a variant of the cacheShuffle [32] that leverages a shuffling instance where dummies are placed in the output array. For comparison, with the standard cacheShuffle, permuting level \( l \) would cost \( 9 \cdot 2^l \) blocks of bandwidth. With our procedure, we reduce this cost to \( 7 \cdot 2^l \).

Given our prior optimizations, for a rebuild into level \( l \), our algorithm, named shortQueueShuffle, takes as input an array of \( n_l \) untouched blocks (possibly including dummies) and produces an output array of length \( 2n_l \), which contains a random shuffling of both the untouched blocks and an additional \( n_l \) dummy blocks. This is a variation of the functionality of Definition 3.2.

For the sake of generalization, for the remainder of the exposition, we set \( n := n_l \) and \( \pi := \pi_l(\Phi_l.r\text{-rank}) \). Similar to cacheShuffle, shortQueueShuffle uniformly at random assigns the indices of the input array into \( \sqrt{n} \) buckets of equal size \( S := \sqrt{n} \). Let \( I_1, \ldots, I_{\sqrt{n}} \) denote the buckets of the input indices. The client also initializes a temporary array at the server, divided into \( \sqrt{n} \) chunks of size \( 2S \). Let \( T_1, \ldots, T_{\sqrt{n}} \) denote the chunks of the temporary array. The client initializes, in private memory, the queues \( Q_1, \ldots, Q_{\sqrt{n}} \). Through \( \sqrt{n} \) rounds, the client performs the following operations. For round \( j \):

1. Download the input chunk \( I_j \) into private memory.
2. For each real block \( x \in I_j \), let \( d = \lceil \pi(x)/\sqrt{n} \rceil \), and place \( x \) in queue \( Q_d \).
3. In \( \sqrt{n} \) rounds, for each queue \( Q_k \), place two blocks in \( T_k \).

If the queue is empty, place dummy blocks instead of real blocks.

At the conclusion of this subroutine, all untouched blocks are either in the correct chunk in the temporary array, that is, a chunk that contains its final destination index, or the correct queue. The routine is named “short queue shuffle” as the arrival rate for each queue is half the departure rate. Subsequently, the client, in consecutive rounds, downloads each chunk in the temporary array; combines the chunk with any remaining blocks in its corresponding queue; arranges the real blocks according to \( \pi \); fills empty spaces with fresh dummies; and uploads the shuffled chunk to the output array.

The procedure is oblivious as the access pattern at the initial downloading of the input buckets does not depend on the input and the remaining accesses are identical for all inputs of the same length. However, if the combined size of the queues becomes \( \omega(\sqrt{n}) \), we exceed our memory threshold and the algorithm fails. Fortunately, this happens only with negligible probability. We summarize performance with the following Lemma.

**Definition 6.1 (Dummy shuffle functionality).** On an input array of length \( n \) and a permutation function \( \pi : [2n] \rightarrow [2n] \), the dummy shuffle functionality outputs an array of length \( 2n \) with the input elements placed according to \( \pi \) and the remaining locations filled with dummies.

**Lemma 6.2.** The shortQueueShuffle is an oblivious dummy shuffling algorithm, it completes in \( 7n \) blocks of bandwidth and uses \( O(\sqrt{n} \cdot B) \) bits of private memory.

Due to its similarities to Lemma 4.2 in [32], the proof is placed in the full version of the paper.

7 RANKORAM PERFORMANCE AND SECURITY

In this section, we reduce both online and offline bandwidth and analyze the performance and security of RankORAM.

7.1 Optimizations

We begin with a modification to the hierarchical ORAM template (see Section 3.3). As our client-side structures, historicalMember-ship and a buffer to perform oblivious shuffle, together occupy \( O(n + \sqrt{\pi n}) \) bits, we can afford to store the smaller levels at the client. To stay within the asymptotic memory bound, we trim the server-side structure and store levels 1 to \( L/2 \) at the client. This reduces the offline bandwidth by a half.

To reduce the online bandwidth, we use an XOR trick, introduced by Dautrich et al. [10], to reduce the size of the batch of requested blocks sent to the client by the server. The server’s output from the access request is a collection of encrypted dummies plus the encrypted target block. Therefore, the server can XOR the ciphertexts together to produce a single block to send to the client. The client can unpack the retrieved element by assembling the corresponding collection of encrypted dummies and apply one XOR against the returned block. Formally, let \( B_i \) denote the block the server retrieved from level \( i \) and let \( l' \) denote the level that contains the accessed block. If the server returns the compressed batch \( B_S = B_1 \oplus \cdots \oplus B_{L'} \) (ignoring for simplicity the fact that some levels might be empty), the client can decode \( B_{L'} \) by first constructing

\[
B_{C} = B_{1} \oplus \cdots \oplus B_{L'-1} \oplus B_{L'+1} \oplus \cdots B_{L}
\]

as a collection of encrypted dummy blocks. Then the client performs \( B_{L'} = B_S \oplus B_{C} \). Note that this optimization requires server-side computation. Nonetheless, this is a standard optimization from the literature [10, 35].

In addition, to save bandwidth, the evict and shuffle subroutines of the rebuild can be intertwined. That is, instead of constructing a temporary array that is used as input to the shuffle algorithm, and padded with the requisite number of dummy blocks, we can use \( \text{evict}(0) \mid \cdots \mid \text{evict}(l-1) \) and retrieve untouched blocks or generate fresh dummies as the shuffle algorithm requires.

7.2 Performance

With our oblivious shuffle algorithm in Section 6, we can calculate the concrete bandwidth overhead of RankORAM.

**Lemma 7.1.** The amortized bandwidth overhead for RankORAM is \( 4 \log n \).

**Proof.** We reduce bandwidth by storing levels 1 to \( L/2 \) at the client (see Subsection 7.1). Therefore, for online bandwidth, RankORAM downloads, \( L/2 = \log n / 2 \) blocks from the server per access. Similarly, for offline bandwidth, we only need to account for the
Algorithm 3: Rank ORAM hash table eviction: the procedure removes all blocks that were not touched during access at level \( l \)

1. define evict\((l)\)
2. \( S \leftarrow [] \) // empty array of length \( n_l \)
3. for \( i \in \{0, 1, \ldots, 2 \cdot n_l - 1\} \) do
4. \( r \leftarrow \pi^{-1}_l(i) \)
5. if \( r < |\Phi_l| \) then
6. \( // r \) represents a real element
7. \( a \leftarrow \Phi_l(r) \)
8. \( l' \leftarrow \text{level}(a) \)
9. if \( l' = l \) then
10. \( // a \) is untouched
11. \( S \leftarrow S \parallel V[i] \)
12. else if \( r \geq \text{dummyCnt}r \) then
13. \( // r \) represents an untouched dummy index
14. \( S \leftarrow S \parallel V[i] \)
15. return \( S \)

The amortized cost of rebuilding levels \((L/2+1)\) to \( L \). For \( l > (L/2+1), \) a level rebuild costs \( 7n_l \) blocks of bandwidth by Lemma 6.2. Further, a rebuild occurs every \( n_l \) updates. Thus, the amortized bandwidth cost of maintaining a level stored at the server is \( 7 \) blocks. As there are \( L/2 \) levels stored at the server, the amortized offline bandwidth overhead is \( 7 \cdot L/2 = 3.5 \log n \). Therefore, total bandwidth is \( 4 \log n \) \( \Box \).

The bandwidth cost can be improved further by allowing server-side computation to support the XOR trick. When the server XORs a batch of blocks, online one block is sent across the channel per access. This reduces online bandwidth from \( 1/2 \cdot \log n \) to \( 1, \) viz.

**LEMMA 7.2.** With server-side computation, the online bandwidth cost for RankORAM is \( 1 \) and the amortized bandwidth cost is \( 3.5 \log n \).

### 7.3 Security

**LEMMA 7.3.** RankORAM is oblivious

**Proof.** In the pseudocode, all server operations are highlighted in red boxes. The security of the access procedure is standard. If the invariant is upheld (an block is only accessed once in a given level build), then any block retrieved from a hash table appears as a random index (independent of the input logical address) to the adversary. Similarly, all dummy fetches are “fresh” and retrieve an untouched physical address determined by the pseudorandom permutation.

To complete the proof, we need to establish the security of the rebuild method. A rebuild involves two interactions with the server. First, we construct the input array for the shuffle. The security is inherited from the Evict routine. The untouched elements are removed chronologically and are already known to the adversary. Second, we perform an oblivious shuffle. The Evict algorithm is oblivious as the set of untouched indicies is fixed in size \( n_l \) for level \( l \) and determined by the permutation. The server already knows the locations of the untouched elements and, as the indicies are accessed chronologically, no information is leaked. \( \Box \)

Now we have all the components of Theorem 1.1. Security is given by Lemma 7.3. The memory cost is incurred by the underlying shuffling algorithm (Lemma 6.2) and the client-side data structure (Lemma 4.1). Finally, the bandwidth cost is secured by Lemma 7.1.

### 8 COMPRESSING POSITION MAPS

Stefanov et al., with PartitionORAM, were the first to suggest compressing client-side metadata to support a client-memory efficient ORAM protocol [37]. Their method constitutes a key baseline for historical membership. However, the authors omit details for implementing their approach. To supplement their work, we provide those details here. We begin with an overview of their concept.

PartitionORAM partitions the database into \( \sqrt{n} \) smaller hierarchical ORAMs of size \( \sim \sqrt{n} \). Each block is randomly assigned to a point in the partition. The position map of PartitionORAM stores the following pieces of metadata for a block: (1) the partition number; (2) the level number; and (3) the hash table offset. The partition numbers are selected uniformly at random and thereby have high entropy. Consequently, to achieve a more compressible position map, the count of each block is stored, instead of the partition number, and used as input to a pseudorandom function. For example, let \( \text{ctr}_i \) be the count of block \( i \) and PRF denote the function. Then block \( i \) is assigned to partition \( \text{PRF}(i \mid \text{ctr}_i) \). On each access, the count is incremented and the pseudorandom function generates a fresh, and seemingly random, partition number. The advantage of storing the counts is that they are highly compressible for sequential access patterns. This is demonstrated by Opera et al. [30], who outline a compression method designed to leverage sequentiality.

Further, Stefanov et al. [37] note that, if all levels are full, each block has probability \( 2^{l-1} \) of being in level \( l \). Thus, the level level information has low entropy and is highly compressible. No compression algorithm is nominated. Lastly, the hash table location metadata is dispensed with by uploading the blocks, including dummies, with random “aliases”. Then, during retrieval, the client requests blocks by their alias and the server finds the block on the clients behalf. Consequently, server-side computation is introduced. We refer to this combined approach of compressing metadata as compressedCounters. It requires two data structures; one for compressing the block counters; and one for compressing the level information. We now provide an instantiation of both.

#### 8.1 Data structures

The method for compressing counters involves storing counter intervals instead of a separate count for each block. A counter interval stores a single count for each interval of consecutive blocks with the same access frequency. The data structure contains two arrays. An index array \( \text{ind} \) stores the starting index for each interval and a counter array \( \text{ctr} \) stores the count for each interval. Thus the count for an index \( i \in [\text{ind}(j), \text{ind}(j+1)] \) has count \( \text{ctr}[j] \).

The challenge is to keep the arrays compact under a dynamic workload. We want to avoid resizing the array at each update. Thus, we divide each array into segments of width \( \lceil 2Z, Z \rceil \) for some parameter \( Z \). Each segment is implemented with a dynamic array that
we resize in accordance with its current capacity. Thus, to keep the memory allocation tight, at most one segment is resized after each access. The segments are stored in the leaves of a balanced binary search tree (we use an AVL tree [13]). When the combined cardinality of adjacent segments falls below \( Z \), we merge the segments. Inversely, when the cardinality of a segment exceeds \( Z \), we split the segment. The mechanics of the tree implementation keep the structure balanced. The parameter \( Z \) invokes a trade-off: small \( Z \) results in fast updates, as the smaller segment requires less shifting of elements and reallocation of memory, but incurs a large tree structure. On the other hand, large \( Z \) induces slow updates but a small auxiliary tree structure.

The level information can be stored in any dynamic string implementation, such as a wavelet tree [27]. We adopt a run-length encoding, similar to the constituent dictionaries in Section 4. To maintain a dynamic run-length code, we apply the same method for maintaining counter intervals. The code is split into segments, implemented with dynamic arrays, and stored in the leaves of a balanced binary search tree.

8.2 Performance

For completeness, we provide a theoretical bound for compressedCounters. The method is suited for sequential access patterns and has poor worst-case behaviour. This could limit its application in memory-constrained environments under dynamic workloads with changing distributions. In the worst-case, if the memory allocation exceeds the application bounds and this scenario is observable to the adversary, it introduces an additional side-channel.

**Lemma 8.1.** For a database of size \( n \), for \( Z = \Theta(\log n) \), to store the block frequencies on a workload of length \( \text{poly}(n) \), compressedCounters requires \( O(n \log n) \) bits.

The proof is in the full version of the paper.

9 EXPERIMENTS

In this paper we have presented RankORAM, a new hierarchical ORAM scheme based on a novel client-side data structure, historicalMembership. RankORAM trades off client memory to achieve bandwidth efficiency. The focus of the experimental evaluation is to measure the overhead of storing and retrieving metadata at the client when using HierarchicalORAM ORAM. We have implemented two baseline approaches, array and compressedCounters, whose properties are summarized in Table 3. The former is the standard in practice [9, 10, 35, 37] and the latter has not been implemented in an ORAM context. For each data structure, we measure peak client memory and the update time as these measurements depend on the access patterns. The update time measures the total time per access. This would include any query costs that are required to execute the access. We refer the reader to Table 2 for the bandwidth costs of these schemes (compressedCounters can be used with both PartitionORAM and RingORAM) as the bandwidth performance is determined only by input size (otherwise a scheme would leak information).

In order to evaluate the performance of historicalMembership and baseline approaches in practice, we use real and synthetic workloads. The real workloads come from two separate commercial cloud storage network traces. The first trace, provided by Zhang et al. [42], is collected on Tencent Cloud Block Storage over a 6 day period. The average block size is 4KB. The second trace, provided by Oe et al. [28], is collected on the Fujitsu K5 cloud service. The properties of the traces are summarized in Table 5.

In addition, we have generated synthetic workloads based on uniform and Zipfian distributions. With the Zipfian, or skewed, datasets, we vary the size of the problem instance \( n \), from \( 2^{20} \) to \( 2^{29} \) in powers of two? and the skew parameter \( \phi \in \{1.1, 1.2, 1.3, 1.4, 1.5\} \), where 1.1 represents low skew and 1.5 represents high skew. The synthetic datasets allow us to test the schemes on average and worst-case access scenarios that ORAM is designed to protect. We use two block sizes \( B \), 64 bytes and 4KB, simulating the size of a cache line and a page size.

Recall that large-client ORAM schemes, including RankORAM, utilize client memory to temporarily store and shuffle \( O(\sqrt{n}) \) blocks. To this end, we also measure the amount of temporary memory required for reshuffles, that we refer as blockBuffer, and compare it to the memory requirements of the index data structures stored at the client (i.e., array, compressedCounters and historicalMembership). Intuitively, the size of blockBuffer is the minimum requirement of these ORAMs. Hence, the use of methods such as compressedCounters and historicalMembership would be justified only if (1) the memory allocation of array significantly exceeds the allocation of blockBuffer; and (2) the compression methods produce memory allocations less than or equal to the allocations of blockBuffer. Our experiments demonstrate when this is the case.

Prior works [37] have stated that, for a sufficiently large block size, in practice, the memory allocation of block cache significantly exceeds that of the metadata array. However, even for 4KB blocks, experimental work has demonstrated that this is not always the case [9].
### 9.1 Experimental setup

All code is written in C++. We simulate the client and server on a single machine. The server is simulated by an interface that abstracts array management: all data is stored and retrieved on disk.

Each workload is executed on a Hierarchical ORAM to generate a dataset of accesses and rebuilds. The client-side data structures are evaluated on these datasets through the metrics of client-memory and update time. This approach of simulating the accesses and rebuilds at the client allows for a more accurate calculation of update time.

Parameter for compressedCounters. As one of the baselines [37] we use did not provide details on how to implement the compressed counters, in Section 8 we proposed a candidate data structure called compressedCounters. Performance of compressedCounters depends on a parameter, Z, that we tune as follows and use for the main experiments.

Recall that the core of the compressedCounters data structure is a dynamic array, for the counters, and a dynamic string of run lengths, for the levels. Both dynamic structures are split into segments of size $\frac{1}{2}Z$, for a parameter Z, and the segments are stored in the leaves of a balanced binary search tree. To infer the effect of Z on performance we tested compressedCounters on a synthetic dataset ($n = 2^{21}$ and $\phi = 1.2$) for values $Z \in \{20, 200, 2000, 20000\}$. The results are displayed in Figure 2. The test demonstrates a clear trade-off between client memory and update time. For $Z = 20$ updates are fast, as only a small segment is updated on each access. However, due to the size of the auxiliary binary tree, the memory allocation is close to double the other instances of compressedCounters. As Z increases, the memory drops sharply and begins to plateau for $Z = 20$. In contrast, the update time steadily increases. To achieve a good balance between memory and throughput, we select $Z = 20$ for all experiments.

### 9.2 Real data

The results on cloud traces are displayed in Table 5. The size of the blockBuffer, with $B = 4$KB, is 540MB for Tencent and 260MB for K5cloud. Notably, the blockBuffer size is significantly less than the memory allocation for array, including a factor 80 difference for the Tencent cloud. This indicates that, particularly for large $n$, the array represents the significant component of client memory. historicalMembership outperforms array and compressedCounters in client memory. On the Tencent dataset, historicalMembership encodes the block metadata in 0.53 bits per block and reduces client memory by a factor of 135 against the baseline array. On the K5cloud dataset, compressedCounters encodes the block metadata 9.8 bits per block and attains a memory allocation that is larger than historicalMembership by a factor of 10. The encoding is larger than the 2 bits per block hypothesized by Stefanov et al. [37] and demonstrates its sensitivity to the access pattern. Recall that, unlike historicalMembership, compressedCounters has poor worst-case behaviour (see Table 6).

Both compression techniques, historicalMembership and compressedCounters, obtain memory allocations comparable to the blockBuffer. Further, their update times are markedly smaller than a standard network latency of 30–40ms. Thus, the experiments demonstrate the feasibility of both these techniques in practice.

### 9.3 Synthetic Data

The results on synthetic data are displayed in Figure 3. Both plots contain lines that approximate the size of the blockBuffer for $B = 64$ bytes (long dashed) and $B = 4$KB (dashed). Figure 3a expresses the effect of the problem size for $\phi = 1.2$. As expected, the array is proportional to $O(n \log n)$ and grows notably faster than the blockBuffer. Both historicalMembership and compressedCounters grow linearly with the database size. For historicalMembership, this is in line with the theoretical bounds. In contrast, for compressedCounters, this indicates that the worst-case bounds do not hold when there is moderate skew in the access pattern. Further, for an Intel SGX secure processor, with an Enclave Page Cache of 96MB and block size $B = 64$ bytes (matching a typical processors cache line), Figure 3a demonstrates that RankORAM can be executed, on these access patterns, entirely in private memory for $n \leq 2^{27}$.

To illustrate the effect of skew on the compression techniques, Figure 3b plots client memory against the skew parameter. The database size is fixed at $n = 2^{27}$. Both historicalMembership and compressedCounters, on highly skewed access patterns, reduce client memory by a factor of 100 against the baseline array. However, the performance of compressedCounters degrades significantly as the amount of skew decreases. To test this further, we conduct a separate experiment on a uniformly distributed access pattern, which is the worst-case. In this instance compressedCounters obtained a memory allocation larger than array (1.5 GB for

### Table 5: Performance of data structures on real cloud traces with block size $B = 4$KB. The memory required for the rebuild phase (i.e., the size of blockBuffer) is 540MB (0.54 GB) for the Tencent dataset and 260MB (0.26 GB) for the K5cloud dataset.

<table>
<thead>
<tr>
<th>dataset</th>
<th>data structure</th>
<th>client memory (GB)</th>
<th>update time (μseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tencent</td>
<td>array</td>
<td>39.33</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>compressedCounters</td>
<td>0.56</td>
<td>14.4</td>
</tr>
<tr>
<td></td>
<td>historicalMembership</td>
<td>0.29</td>
<td>10.6</td>
</tr>
<tr>
<td>K5cloud</td>
<td>array</td>
<td>9.60</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>compressedCounters</td>
<td>1.30</td>
<td>8.1</td>
</tr>
<tr>
<td></td>
<td>historicalMembership</td>
<td>0.13</td>
<td>2.7</td>
</tr>
</tbody>
</table>
compressedCounters and 1.2GB for array when $n = 2^{27}$). In contrast, historicalMembership, still outperformed array by a factor of 12 when the access pattern was uniformly distributed.

On all instances of synthetic data, historicalMembership requires a lower memory allocation than the blockBuffer (the memory required for the rebuild phase). At the same time, for larger values of $n$, the array obtains a memory allocation that exceeds the blockBuffer by a factor of at least 10. This testifies to the efficacy of our approach.

We note that one has to be careful when deploying historicalMembership and compressedCounters in practice as varying memory requirements between the skews could introduce an additional side-channel revealing the type of access pattern. To this end, reserving memory for the worst-case is advisable. For such cases, historicalMembership data structure would be preferred due to an order-of-magnitude smaller worst-case memory requirement.

10 CONCLUSIONS

We have presented the first protocol for Hierarchical ORAM that can retrieve the required block in a single round without requiring server computation. Our construction, RankORAM, exploits a larger client-memory allocation, relative to prior work, to achieve improved bandwidth and latency performance. The foundation of RankORAM is a novel client-side data structure, historicalMembership, that maintains a compact representation of the locations of the blocks at the server. Significantly, historicalMembership can be used in any Hierarchical ORAM to reduce the number of round trips of communication, per access, from $\log n$ to one. Further, with historicalMembership levels at the server can be stored as permuted arrays, avoiding complex hash tables, and allowing fast and practical oblivious shuffle algorithms to be used for rebuilds.

Compared to state-of-the-art practical solutions, PartitionORAM and RingORAM, we reduce client memory by a logarithmic factor, while maintaining comparable bandwidth and latency performance. The standard for practical ORAMs is to use an array to store position maps at the client. Our experiments, on real network file system traces, demonstrate a reduction in client memory by a factor of 100 compared to the array approach and by a factor of 10 compared to closest related work.

ACKNOWLEDGMENTS

William Holland is in part supported by the Australian Government RTP Scholarship Research Training Program. Anthony Wirth is in part supported by the Australian Research Council, grant number DP190102078. We acknowledge the Wurundjeri and Bunurong Peoples of the Kulin Nations as traditional owners of the land on which we live and work.

REFERENCES

Chapter 6

Trade-offs in Oblivious Permutation

Oblivious permutation is a key primitive in almost all Oblivious RAM solutions. To supplement our work on Oblivious RAM, we provide a novel Oblivious Permutation algorithm that explores a new point in the trade-off between client memory and bandwidth. The work is presented as a paper published\(^1\) in AsiaCCS 2022 and is a solution to Research Question 4. The contribution is a new routing algorithm for the Waksman permutation network, WaksmanOP, which reduces client memory by a logarithmic factor against prior approaches.

The idea for this problem emerged when we started our work on RankORAM (Section 5). For a database of size \(n\), the latter provides an oblivious storage solution in \(O(n)\) bits of client memory. This memory allocation is not typical within the oblivious permutation literature, where solutions are generally given with either \(O(B)\) or \(O(\sqrt{n} \cdot B)\) bits of client memory, for a \(B\) bit block size. Thus, as a starting point, we asked whether new performance improvements, or trade-offs, were available for oblivious permutation under the \(O(n)\)-bit bound. Notably, and this is a key contribution, we looked to remove the sensitivity to the block size, which can be in the megabytes for some applications, found in earlier work \(^{117}\).

Sorting networks are a popular choice for oblivious permutation, as they require only \(O(B)\) bits of client memory. However, they are a poor choice in practice. Our study began by considering permutation networks, which have superior bandwidth performance to sorting networks, but require \(\Theta(n \log n)\) bits to perform the routing. The memory cost has prohibited their use in practice. The Waksman network \(^{143}\) is the optimal permutation network in terms of the number of switches and represented an obvious starting point. An alternative network, on which our approach could be applied, is the Benes network \(^{112}\). This alternative would allow for a simpler implementation and routing algorithm, but at the cost of a moderate increase in bandwidth\(^2\). In the interest of bandwidth efficiency, we focused on the Waksman network. Consequently, our goal was to construct a routing algorithm for the Waksman network within \(O(n)\) bits of client memory.

Similar to Chapter 5, this work involves trade-offs with respect to existing solutions and builds an argument, regarding the contribution, that draws from the research context. However, the application scenario is reverse of that in Chapter 5. With RankORAM, we demonstrate that

---


\(^2\)The Waksman network has \(n \log n - n\) switches, compared to \(n \log n - n/2\) switches for the Benes network.
the scheme is suitable for parameter values falling in the case \( n \log n \gg \sqrt{n} \cdot B \). In contrast, WaksmanOP is suitable for large block sizes and when \( \sqrt{n} \cdot B \) is “large”. Unfortunately, this does appear to be a smaller window of application, but our experiments demonstrate a range of concrete parameter values where WaksmanOP would be the algorithm of choice. In addition, the Waksman network has been applied in a number of security applications outside of oblivious storage and computation, so there is broad scope for the use of our approach.

The network configuration algorithm in WaksmanOP has a promising direction for future work. Chen et al., in their work on tree-based ORAMs, construct a homomorphic permutation algorithm based on the Waksman network [37]. The client configures the network settings in private memory, encrypts the settings and sends them to the server, which executes the permutation on behalf of the client. This approach works well for tree-based ORAMs as shuffles are evaluated on a constant number of blocks. Therefore, the client can afford to store the network settings in private memory. In contrast, for RankORAM, shuffles at the top level occur on input arrays of length \( 2^n \) and require \( O(n \log n) \) bits of private memory using standard techniques. Methods from WaksmanOP can reduce this cost to \( 2^n \) bits, matching the memory bound of RankORAM, and has the potential to reduce the amortized bandwidth cost to \( o(\log n) \) blocks. We discuss this direction in more detail in Chapter 7.
ABSTRACT

Memory accesses to data stored on an untrusted server are known to leak information, even if the data is encrypted. The oblivious permutation (OP) is a key primitive for algorithms and protocols that are designed to hide these client accesses to the server. An OP algorithm permutes outsourced data blocks according to a given permutation without revealing the permutation to the server.

Existing solutions strive to use only \( O(B) \) bits of client memory when permuting \( n \) blocks of \( B \) bits. State-of-the-art \( O(B) \) bit solutions, require an optimal \( O(n \log n) \) I/Os to complete. However, the hidden constant factor is at least 19600. In this work, we depart from this memory constraint and, in pursuit of an I/O efficient algorithm, consider the context of cloud storage where a client can have a larger amount of private memory.

We propose an algorithm, WaksmanOP, that uses \( 2n + o(n) + 2B \) bits of client storage and permutes data in at most \( 4n \log n + 3.6n \) I/Os. WaksmanOP is based on the Waksman network and involves a novel routing algorithm that carefully configures network switch settings using small client space. We implement WaksmanOP and compare it with existing solutions. Compared to practical methods based on sorting, WaksmanOP reduces the number of I/Os by a \( \log n \) factor and uses significantly less client space than methods based on shuffling for large values of \( B \). (e.g., 41MB vs. 6.4GB of private memory to permute 16TB of data).

CCS CONCEPTS

- Security and privacy → Management and querying of encrypted data.

KEYWORDS

oblivious algorithms, permutations, access pattern

1 INTRODUCTION

Cloud servers are a common option for satisfying personal storage requirements (e.g., OneDrive, Dropbox, GoogleDrive) and business needs (e.g., Amazon, Microsoft Azure, Google Cloud). Storing sensitive and confidential data with a cloud provider naturally raises security and privacy concerns that encryption alone does not protect against. In particular, a client’s read and write accesses to the server, including the order, correlation and frequency of the accesses, have been shown to reveal information about the stored data [14, 25, 44].

Data-oblivious algorithms aim to access memory such that a “curious” server cannot determine information about the algorithm’s private input besides some public information, for example, data size. They do so by carefully arranging, accessing and occasionally shuffling the server’s memory. Since additional accesses are required to obfuscate the original memory accesses, a significant amount of research [23, 27, 40, 42] has been dedicated to improving this overhead. This current of research includes fundamental theoretical work [6, 28] as well as practical implementations [38, 39]. Recent interest in data-oblivious algorithms [2, 34, 39, 39] has also been fuelled by the availability of Intel SGX technology [4] and its adoption by cloud providers such as Microsoft [30] and IBM [26]. In this setting, the trusted processor has access to encrypted memory and acts as a secure client while the memory outside is regarded as untrusted storage. Careful use of data-oblivious algorithms can then be employed to protect against side-channel attacks based on, for example, cache accesses and page faults [14, 15, 29, 31, 44].

In this paper we revisit the fundamental building block of many data-oblivious algorithms — data-oblivious permutation (OP). We introduce an algorithm that exploits a new trade-off between client storage and the number of additional accesses (i.e., I/O overhead) as compared to the state-of-the-art algorithms (see Table 1 for comparison). Oblivious permutation shuffles the contents of the server’s memory of \( n \) elements according to a given permutation \( \pi \), without revealing \( \pi \) to the server through its memory accesses. In other words, an adversary observing the access sequence should have (approximately) a \( 1/n! \) chance of guessing the correct permutation if \( \pi \) is chosen uniformly from the set of all possible permutations of \( n \) elements. Besides being a fundamental primitive for hiding the location of elements, the oblivious permutation is an underlying building block in oblivious RAM solutions [6, 19, 21, 35], oblivious data structures [35], and permuting data records in the shuffle model of differential privacy using trusted processors [13, 17, 18].

There are two families of state-of-the-art OP solutions: those based on sorting networks [1, 7, 20, 32], and those based on shuffling [33, 36]. Sorting-network approaches use constant client memory, enough to read and sort a constant number of elements each of size \( B \). Although \( O(n \log n) \) I/Os is used in [1] and [20], the hidden
constants are high (e.g., 19,600 for Zig-Zag sort [20]). Therefore, Batcher’s and bitonic sorting networks [7, 32], with \( n \log n \) operations, are preferred in practice. The shuffle-based solutions rely on client memory sublinear in \( n \) in order to read larger batches of elements, shuffle them and write them back, resulting in a linear number of I/Os with \( O(\sqrt{nB}) \) bits of client space.

Client storage plays an important role in the design of oblivious algorithms. Intuitively, the more element blocks the client can read, the fewer I/Os it takes to complete the permutation. Though, historically, small client storage was preferred, the constraint of constant-sized memory appears to be too pessimistic in a cloud storage setting [12, 22, 38, 41]. In particular, Bindschaedler et al. [12] remark that “seek[ing] to minimize the client-side storage…is increasingly out of the touch with the reality of modern computing”. Though we agree with the principle, we also argue that a bound of \( O(\sqrt{nB}) \) on client storage is too lenient, especially for a large block size, \( B \). To this end, by evading sensitivity to the block size, we exploit a new position in the trade-off between reasonable client storage and the I/O overhead.

We develop an oblivious permutation algorithm, WaksmanOP, based on the Waksman switching network [43]. Through careful use of client and server memory, when configuring the network, our algorithm can permute an array in at most \( 4n \log n - 3.6n \) I/Os using \( 2n + o(n) + 2B \) client space. The I/O overhead is asymptotically equal to state-of-the-art theoretical constructions based on sorting, while not suffering from high constants. Compared to methods based on shuffling, in practice, it uses significantly less client space when the block size, \( B \), is large. For example, consider a setting where the client has a secure processor, such as the Intel SGX, to shuffle 640,000 elements each of size 1KB (= 1GB of data). WaksmanOP can be implemented using a processor’s 256KB private mid-level cache [24] since it requires only 160KB of temporary space. This is in contrast to the 4MB and 1MB memory requirements for permuting elements with state-of-the-art algorithms, respectively, the Melbourne Shuffle [33] and Bucket ORP [5].

The Waksman switching network — first proposed in 1968 [43] for a power-of-two and later generalised in 2002 to arbitrary-sized networks [8] — contains \( n \log n - n \) binary switches that connect an element of an input array to a unique index in the (shuffled) output array. Binary switches in the network are used to decide whether to swap the order of a unique pair of inputs:

![Figure 1: The persist (left) and swap (right) switches. A switch is a permutation network of size two.](image)

The settings of the switches and the structure of the network determine the consequent output permutation. Naïve use of the network, for the purposes of OP, requires \( \Theta(n \log n) \) bits of client memory to store the values of the switches. Unfortunately, for \( B = O(\log n) \), this memory commitment is of the same size, asymptotically, as the outsourced storage.

Our WaksmanOP overcomes the client space overhead by interleaving network configuration and routing in a way that requires only \( 2n + o(n) \) bits of client memory. Our implementation of the network builds on a novel method of recursively configuring the switches to avoid using more than a couple of bits of client memory per input element (or block). We emphasize that, in the execution of the algorithm, client memory is orders of magnitude smaller than server memory for non-trivial block sizes. For example, \( 2n + o(n) \) bits is used to shuffle \( nB \) bits, where the block size \( B \) can vary from several kilobytes to megabytes depending on the application.\(^1\)

In summary, our contributions are:

- We take a new view of the problem of efficient oblivious permutation and focus on a larger (but affordable in practice) client memory that is not sensitive to the block size.
- We propose a novel algorithm for routing on the Waksman network (WaksmanOP). The algorithm requires \( 2n + o(n) + 2B \) bits of client memory and completes in at most \( 4 \log n - 3.6n \) read/write accesses to the server. With reference to Table 1, the algorithm provides a meaningful trade-off between client storage and I/O for large block sizes, \( B \), compared to existing methods.
- We implement and compare WaksmanOP with recognized baselines. Our experiments show that WaksmanOP would be the algorithm of choice for a set of parameters that reflect many realistic scenarios. Our implementation is publicly available\(^2\).

## 2 PRELIMINARIES

Let \( \lambda \) denote the security parameter and \( z \overset{\$}{\leftarrow} Z \) denote an item drawn uniformly at random from the set \( Z \). Oblivious random permutation algorithms implement the following functionality.

**Definition 1** (Functionality: Array Permutation). Let \( \mathcal{P} \) denote a set of permutations. On input array \( U \) and permutation \( \pi \in \mathcal{P} \), the Array Permutation outputs the array \( V = \pi(U) \), where \( V[i] = U[\pi(i)] \).

Let \( n \) be the number of elements or blocks in array \( U \). We assume that a permutation function \( \pi : [n] \rightarrow [n] \) is given to an algorithm in a form that allows for the efficient evaluation of \( \pi(i) \). For example, it could be provided as a seed to a pseudo-random permutation. However, we emphasize that the above functionality does not assume anything about the randomness in the choice of \( \pi \) nor that it specifies that \( |\mathcal{P}| = n! \). The goal is simply to arrange elements according to the order given in \( \pi \). As the input \( \pi \) can be selected by the client, according to any distribution, this is a stronger definition than the random permutation functionality [5]. However, we note that the security of methods that use oblivious random permutation often rely on \( \pi \) being a pseudo-random permutation (e.g., oblivious RAM).

**Probabilistic encryption** is an important ingredient in all oblivious algorithms, and one that we incorporate in our solution. Everything stored at the server is encrypted using semantically secure symmetric encryption [9] and each time an element is read from the server, the user decrypts it, re-encrypts it and writes it back. Hence, the

\(^1\)Minimal billable objects for some of Amazon S3 storage classes are set to 40 KB and 128 KB. [3]
\(^2\)Code is available here: https://github.com/wCloudRain/orp
server, aka the adversary, cannot tell whether the ciphertexts correspond to the same element or not, since the ciphertexts produced for the same element are likely to be different.

### 2.1 Storage Model

Our model is explicit in the client-server application. We consider a framework where the client stores, at the server, an array of \( n \) blocks with \( B \) bit values and maintains a local memory allocation (asymptotically) smaller than the size of the outsourced data, that is, an allocation of \( o(nB) \) bits. The client can use temporary arrays, or storage, at the server to facilitate the execution of a function on the stored data. The client can access and modify the data with read and write operations of the form \( \text{data} \leftarrow \text{read(name, addr)} \) and \( \text{write(name, addr, data)} \) where:

- \( \text{name} \) is the name of the array being accessed.
- \( \text{addr} \in [n] \).
- \( \text{data} \in \{0,1\}^B \).

If the client sends the operation \( \text{op} = \text{read} \) to the server, the server returns the \( B \)-bit value stored at address \( \text{addr} \). Alternatively, if \( \text{op} = \text{write} \), then the value data is written at address \( \text{addr} \) at the server. In addition, the client can initialize new arrays and delete existing arrays.

We define the access pattern as the sequence of memory accesses or operations at the server by the client. A client-server algorithm implements a functionality on some array stored at the server. To provide security, the client encrypts array elements. We follow the notation \( u_i = U[i] \) to refer to the \( i \)th element of array \( U \).

### 2.2 Obliviousness

We define the adaptive security of an oblivious algorithm in terms of a two-player game, following precedent adaptive security definitions [9]. In essence, the adversary should not be able to distinguish between the access patterns of two inputs of their choosing, even after observing access patterns for other permutations. Let \( \text{AccPtrn}(M(\cdot)) \) denote the sequence of memory addresses accessed by the client at the server during the execution of algorithm \( M(\cdot) \). The game in Algorithm 1 is played between \( M \) and an adversary, \( \mathcal{A} \). The adversary wins if the game outputs True. The advantage of the adversary is defined as \( 2 \cdot \Pr[\text{OA-CPA}_M^{\mathcal{A}}(\lambda, k) = 1] - 1 \). In the following definition the adversary is permitted to request a number, \( k \), of inputs that is polynomial in \( \lambda \).

**Definition 2** (Security: Oblivious algorithm). Let \( \lambda \) be a security parameter and let \( M \) be an algorithm implementing the function of Definition 1. We say that \( M \) is oblivious if for all \( k = \text{poly}(\lambda), k_i \in \{1, \ldots, k\} \), and for all PPT adversaries \( \mathcal{A} \), the advantage of the adversary in OA-CPA\(^\mathcal{A}\)(\(\lambda, k\)) is negligible.

A consequence of this definition is that an adversary observing the access pattern of \( M \) learns only the size of the input and nothing about the input itself, since \( |U_0| = |U_1| \) to avoid trivial detection.

### 2.3 Performance

For measuring performance we consider two key parameters: client memory and the number of I/O operations. The size of the client memory measures the amount temporary storage required to execute the oblivious permutation algorithm. The number of I/Os refers to the number of read and write operations performed by the client, where each I/O can transfer one block of size \( B \).

Two secondary measures we consider are: client work and server memory. Client work refers to the amount of computation performed at the client. As the algorithms employ temporary storage...
at the server, the size of this temporary storage needs to be consid-
ered, as it typically occurs at some financial cost to the client.

3 RELATED WORK

Sorting and switching networks for OP. Early solutions to oblivious permutation involve assigning distinct labels to elements and applying a sorting network. The labels are assigned by the input permutation, \( \pi \). Sorting networks are composed of several levels of carefully laid out comparators that gradually transform the input to a sorted output. The comparator layout depends only on the input size. Thus, sorting networks are oblivious. A comparator takes two elements and rearranges them based on their relative order. The output of each comparator serves as an input into comparators of the next level. Since the comparators can be evaluated in a sequence, client memory is small: the algorithm reads the pair of elements that correspond to the current comparator and writes them back to the server according to their relative order. The number of I/Os depends on the number of comparators, which we refer to as network size. With optimal\(^3\) theoretical sorting networks, such as AKS [1] or Zig-Zag [20], the permutations can complete in \( O(n \log n) \) I/Os. However, the asymptotic bounds hide large constant factors (e.g., 19,600 for Zig-Zag [20]) and the non-optimal bitonic sort [32], with \( n \log^2 n \) comparators, is considered a practical alternative. For example, AKS or Zig-Zag become feasible alternatives only for \( n \geq 2^{1900} \).

Additionally, oblivious random permutations can be achieved through randomized switching networks [16], but are not considered practical due to their size. The latter work of Czumaj is significant as small deterministic switching networks, which perform all possible permutations, fail when the switches are set uniformly and independently at random: the output permutation of the randomly configured network is not uniform. This observation has largely prevented their adoption in both theory and practice.

Oblivious Shuffles. The Melbourne shuffle is an oblivious permutation that does not rely on oblivious sort [33]. The authors were the first to leverage larger client memory, relative to prior work, in pursuit of lower I/O and bandwidth costs. In contrast to algorithms based on sorting networks, the Melbourne Shuffle is randomized and may fail with small probability without affecting its security. In terms of performance, it is the state-of-the-art for I/O efficiency in settings where the client is permitted \( \sigma(nB) \) bits of memory. As the algorithm requires \( \Omega(n \log B) \) bits of client memory, it is sensitive to the block size and can become untenable for large values of \( B \). Patel et al. [36] show that the Melbourne shuffle can be adapted to client memory of size \( O(BS) \) while incurring an \( O(n \log B) \) I/O cost, for \( S = \omega(\log n) \).

A recent work, named Bucket ORP [5], provides a slightly different functionality: it randomly permutes the array to some unspecified permutation\(^4\). The algorithm assigns elements to random bins and routes elements into the bins through a butterfly network.

\(^3\)The notion of optimal here assumes that the keys are sufficiently large. This is always the case during oblivious random permutation as we are sorting the key set \( \{0, 1, \ldots, n-1\} \). In contrast, one-bit keys can be obliviously sorted in \( \sigma(n \log n) \) time [28].

\(^4\)Note that the functionality of Definition 1 can be satisfied by first applying Bucket ORP to randomly permute the array to some unspecified permutation \( \sigma \) and then non-obliviously rearranging the elements according to the input \( \pi \). Subsequently, the bins are individually permuted and then concatenated together. Bucket ORP achieves a competitive balance between I/O and client memory efficiency and constitutes a key benchmark. Bucket ORP does not achieve perfect obliviousness. Notably, when it shuffles the input towards some unspecified permutation, not all \( n! \) permutations are achievable. Though the set of impossible permutations is negligible and is governed by a security parameter, it consists of permutations that follow a specific structure such as the identity permutation or the reverse permutation. As a result it cannot be used for applications where perfect obliviousness is required. In contrast, WaksmanOP achieves perfect obliviousness as its accesses are deterministic and reveal nothing about the input permutation.

Table 1 summarizes the asymptotic performance of approaches for oblivious permutation, including our algorithm, WaksmanOP,

Applications of the Waksman network. Prior to this work, the Waksman network has been adopted in the setting of multi-party secure computation (MPC). Zahur et al. [45] revisit the square root oblivious RAM of Goldreich and Ostrovsky [19] in the setting of MPC and obtain performance improvements by replacing the oblivious sorting primitive with a Waksman switching network. The authors demonstrate performance improvements across a concrete parameter space. Zahur et al. [45] use the Waksman network in a model that is significantly different from ours. They do not have the same constraints on client storage and can configure the Waksman network with the algorithm outlined in the original paper [43], which is to say, with more client memory than is permitted here. Finally, routing networks, including the Waksman network, have been used to verify the integrity of memory accesses in outsourced computation [10, 11]. This includes an implementation of the Waksman network [37] for verifiable computation, where the server employs the network to prove the correctness of the computation to the client.

Waksman for Oblivious permutation. The Waksman network can route an input of \( n \) elements to any permutation using \( n \log n - n \) switches. Hence, it provides superior I/O efficiency against oblivious permutation algorithms based on practical sorting networks with \( n \log^2 n \) comparators. However, despite this promise, the Waksman network has not previously been adopted for oblivious permutation. This is largely due to the fact that one requires a *global view of the network* in order to set the switches according to an input permutation. The burden of configuring and storing the switch settings, along with the consequences for algorithm performance, need to be managed by the client. Note that sorting networks do not suffer from this restriction since the “setting” of each comparator depends only on the relative order of the input elements (e.g., determined using the elements’ permutation labels).

The goal of our work is to rescue the Waksman network from this fate. We construct a non-trivial routing algorithm, WaksmanOP, for an oblivious execution of the Waksman network, that admits client memory allocation asymptotically smaller than prior (or naive) approaches.
For even-sized networks (a) the subpermutations each have size $\lfloor n/2 \rfloor \times \lfloor n/2 \rfloor$ and for odd-sized networks (b) the subnetworks have sizes $|\pi_0| = \lceil n/2 \rceil \times \lceil n/2 \rceil$.

4 ARBITRARY SIZED WAKSMAN NETWORK

Our exposition begins with a high-level outline of the structure of the Waksman switching network and, for the time being, ignores memory considerations within the client-server framework. We conclude the section with a handful of naive solutions which serve to highlight the rationale behind our approach in Section 5.

4.1 Network structure

Fixing notation, given a permutation, $\pi$, and the input array

$$U = \{u_0, u_1, \ldots, u_{n-1}\},$$

the network returns the output array

$$\pi(U) = V = \{v_0, v_1, \ldots, v_{n-1}\},$$

where $v_{\pi(i)} = u_i$. The elementary units of the structure are:

- a permutation network on one input, named a wire;
- a permutation network on two inputs, named a switch.

As there are two instances of a size-two permutation, a switch has two available settings. We refer to the identity permutation ($\{(0) = 0, \pi(1) = 1\}$) as the persist switch and its complement ($\{(0) = 1, \pi(1) = 0\}$) as the swap switch. The negation operator ($\neg$) is used to refer to a reversed switch setting: $\neg$(persist) = swap.

The two settings are visualized in Figure 1.

A switching network is an arrangement of interconnected switches and wires that link the $n$ inputs to the $n$ outputs along edge-disjoint paths. The structure of the Waksman network is expressed recursively (illustrated in Figure 2). The network consists of an exterior of switches and an interior of (sub)permutation networks. The exterior switches are split into entry switches $E^\pi = \{E_i^\pi\}$, which connect the input wires to the interior subpermutations, and exit switches $X^\pi = \{X_j^\pi\}$, which connect the interior subpermutations to the output. On input $U$, the entry switch $E_i^\pi$ receives the pair $(u_2i, u_2(j+1))$ and the exit switch $X_j^\pi$ returns the pair $(v_2j, v_2(j+1))$, where $v_{\pi(i)} = u_i$. We drop the superscripts from $E_i^\pi = \{E_i^\pi\}$ and $X_j^\pi = \{X_j^\pi\}$ when obvious from the context or when referring to the switches in general.

The interior of the network is divided into two subpermutation networks, $\pi_0$ and $\pi_1$. On even-sized inputs each subpermutation has size $n/2$ (see Figure 2a) and on odd-sized permutations the interior is split into two subpermutations, of sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ (see Figure 2b). The subpermutations perform a fragment of the global permutation $\pi$. All entry switches have one output wire connected to $\pi_0$ and one output wire connected to $\pi_1$. Thus, each entry switch segregates pairs of elements into separate subnetworks. Reciprocally, the input wires of each exit switch combine elements from the subnetworks $\pi_0$ and $\pi_1$. The subpermutation networks are implemented with the Waksman structure. Thus, the structure proceeds recursively. The recursion terminates at either the size-one or size-two permutation.

The network structure is fixed for each $n$, while the values of the switches depend on the permutation $\pi$. Note that for even-sized permutations one of the exit switches (i.e., $X_{n/2-1}$ in Figure 2a) is “missing”. This is the result of an observation made by Waksman in which one switch in the exterior can be set prior to knowing $\pi$.

The last two outputs are connected directly to the interior subpermutations. The following result bounds the number of switches in a given network.

Lemma 1 (Corollary 3 [8]). An arbitrary-sized Waksman network, of size $n$, contains less than $n \log n - 0.9n + 1$ switches.

4.2 Waksman Subpermutations

The Waksman network can be decomposed into a collection of subpermutations that route the elements towards the global permutation, $\pi$. The arrangement of subpermutations can be conceptualized as a binary tree, referred to as the permutation tree. In reference to Figure 2, $\pi$ constitutes the root node and its interior subnetworks $\pi_0$ and $\pi_1$ designate its children. In turn, the node $\pi_0$ branches into the subnetworks $\pi_{00}$ and $\pi_{01}$. The tree branches to
the base case: a permutation network of size one (a wire) or two (a switch). An example of the subnetwork tree, for a permutation of size 9, is available in Figure 3.

The function that corresponds to a subpermutation depends on both the global permutation and the switch settings in the subsuming exteriors, that is, the ancestor nodes in the permutation tree. For example, in Figure 2, \( \pi_0(i) \) is a function of \( \pi \) and \( E_{i}^\pi \) since the entry switch \( E_{i}^\pi \) supplies the \( j \)th input wire of \( \pi_0 \). Moreover, the relation \( \pi_0(i) = j \) dictates that an element from the pair \((u_{2i}, u_{2i+1})\) maps to an element in the pair \((u_{2j}, u_{2j+1})\) under \( \pi \). As a result, the subpermutation functions are set as follows.

\[
\begin{align*}
\pi_0(i) &= \left\{ \begin{array}{ll}
\lfloor \pi(2i)/2 \rfloor & \text{if } E_i = \text{persist}, \\
\lfloor \pi(2i+1)/2 \rfloor & \text{otherwise.}
\end{array} \right. \\
\pi_1(i) &= \left\{ \begin{array}{ll}
\lfloor \pi(2i)/2 \rfloor & \text{if } E_i = \text{swap}, \\
\lfloor \pi(2i+1)/2 \rfloor & \text{otherwise.}
\end{array} \right.
\end{align*}
\]

(1) (2)

A consequence of this formulation is that, in order to evaluate a subpermutation \( \pi \), at depth \( l \), one needs access to the values of the exterior settings of the ancestor subpermutations. For example, \( \pi_{01} \) depends on \( \pi_0 \) and the settings \( E_{i}^{\pi_0} \), while \( \pi_0 \) depends on \( \pi \) and \( E_{i}^{\pi} \). As a result, only the root \( \pi \) is easy to compute (see Section 2) and the evaluation of a subpermutation at depth \( l \) requires knowledge of the exterior settings of its ancestor subpermutations and, consequently, takes \( O(l) \) time to compute. Another important observation is that the settings need to be set at the exterior of the parent network before evaluating the subpermutation function of a child node. The inverse subpermutation \( \pi_{01}^{-1} \) can be defined analogously to Equations (1) and (2) using \( \pi^{-1} \) and the exit switches, \( X^\pi \).

### 4.3 Setting the exterior switches

Procedures for setting the exterior switches are all based on the observation that an input permutation enforces dependencies between the exterior switches. As has been observed in prior work \([8, 43]\), the exterior of the network, in conjunction with the input permutation, can be viewed as a bipartite graph. The nodes of the graph are the elementary units of the exterior and the edges are determined by the input permutation. For example, the relation \( \pi(i) = j \) implies that an edge between nodes \( E_{i/2} \) and \( X_{j/2} \) exists. An illustration of a bipartite graph representation of a Waksman exterior is provided in Figure 5.

Each node in the bipartite graph has at most two edges. Beaquier et al demonstrate that the problem of setting the exterior switches reduces to 2-colouring the bipartite graph representing the exterior \([8, \text{Theorem 4}]\). The edge colours determine how to connect the subnetworks to the input elements through the switches. For example, the output element \( v_{n-1} \) is always wired directly to \( \pi_1 \). Thus, edges with the same colour as the edge representing the output element \( v_{n-1} \) route elements through the subpermutation \( \pi_{1} \). If the edge corresponding to the relation \( \pi(i) = j \) has the same colour as the edge representing output \( v_{n-1} \), then the switch \( E_{i/2} \) is set to swap if \( i \) is even and persist if \( i \) is odd.

It is trivial to 2-colour a bipartite graph with node degree bounded by two. All cycles are even in length. Therefore, beginning with an arbitrary edge color, we can traverse each cycle alternating the colours of the edges. When a cycle is completed, we can commence from an arbitrary uncoloured edge and arbitrarily assign it a starting colour, as shown in Figure 5. We refer to this 2-colouring algorithm as \( \text{SetExterior} \).

The input permutation, \( \pi \), and the switch settings determine, respectively, the graph and the colouring. Given \( \pi \), it suffices for the algorithm to store two bit vectors: one to indicate whether a switch has been set and one to store the settings. The former is used to determine the starting points of new cycles and can be deleted at the conclusion of the algorithm. Each bit vector has length at most \( n \). As the bit vectors dominate the memory commitment, the algorithm executes in \( 2n + o(n) \) bits.

**Lemma 2.** The algorithm \( \text{SetExterior} \) computes the settings of the exterior of a Waksman network in \( 2n + o(n) \) bits.

### 4.4 Towards Oblivious Permutation with Waksman Network

The routing of the elements through the recursive structure of the Waksman network is facilitated by a sequence of temporary arrays. Each temporary array holds (concatenated) the elements of the subpermutations at a given depth in the permutation tree. Fixing notation, let \( V_i \) denote the temporary array corresponding

---

**Figure 3:** The tree of subnetworks for a permutation network of size 9. The configuration phase performs a pre-order traversal of the tree. Upon arriving at each node, the \( \text{SetExterior} \) primitive is applied and elements are routed into the subnetworks of the child nodes.

**Figure 4:** A 9 × 9 Waksman network set to \( \pi = (821507346) \). The colouring of the bipartite graph (Figure 5) is depicted through the connections to the subnetworks \( \pi_0 = (1032) \) and \( \pi_1 = (41023) \).
Figure 5: Bipartite graph determined by $\pi = (821507346)$. A solution to the 2-colouring is provided. Blue (dotted) edges route elements through the upper subnetwork ($\pi_0$) and black edges route elements through the lower subnetwork ($\pi_1$). The blue edge $E_0 - X_1$ is determined by the relation $\pi(1) = 2$. The colouring implies the setting $E_0 = \text{swap}$ (see Figure 4). The colouring can be performed by starting at node $v_8$ (which is wired directly to $\pi_1$) and alternating the edge colours on the traversal.

Algorithm 2: WaksmanOP algorithm for permuting array $U$ according to permutation function $\pi$.

```
1 define ObliviousWaksmanShuffle(U, $\pi$)
   // Setup root node for global permutation $\pi$
   // and its switches
2   $\pi$.function $\leftarrow \pi$; $\pi$.size $\leftarrow |U|$; $\pi$.offset $\leftarrow 0$
3   $\pi$.depth $\leftarrow 0$; $\pi$.E $\leftarrow$ [null]; $\pi$.X $\leftarrow$ [null]
4   ConfigurationRouting($\pi$) // Configure all switch settings and route elements according to entry switches
5   EmptyRoadPhase // Route elements according to exit switches
```

to the subpermutations at depth $l$. Thus, the first half of $V_j$ holds the input to $\pi_0$ and the second half of $V_j$ holds the input to $\pi_1$. For example, in reference to Figure 4, following the persist setting of $E_2$, we have $U[4] = V_j[2]$ and $U[5] = V_j[6]$. The combined collection of the input, temporary and output arrays are all stored at the server. We note that not all temporary arrays need to be stored contemporaneously. An element occupies one temporary array at any given time. Hence, the server stores at most $n/2$ temporary values.

Naive Solutions. The Waksman network can be used to perform the array permutation function, efficiently, in small client memory: shuffle the input by setting Waksman network switches at random and then non-obliviously re-arrange the shuffled output according to $\pi$. Unfortunately, the random switch assignment does not produce a uniform permutation. As not all permutations are equally likely, the adversary learns a correlation between the element locations in the input and the output. This dependency would then propagate when arranging this “not-properly” shuffled output according to $\pi$ directly. Hence, this approach is not oblivious.

On the other hand, solutions based on pre-configuring the network settings according to an input permutation $\pi$ also have drawbacks. Storing the switch settings locally costs $\Theta(n \log n)$ client memory, which is proportional to the server allocation for $B = O(\log n)$, thus defeating the purpose of using server storage. Alternatively, configuring the network settings efficiently at the server, in an attempt to reduce the client memory overhead, produces an access pattern that leaks information about the permutation.

These two naive solutions, one insecure and one inefficient, indicate the direction we must take in pursuit of an efficient and oblivious permutation algorithm: for security, we are required to compute switches at the client; for efficiency, we are forbidden from storing the full network. Thus, our approach, WaksmanOP, configures portions of the network, locally at the client, and routes elements through the network portions.

5 WaksmanOP: EFFICIENT OBLIVIOUS PERMUTATION

Our method simultaneously configures and routes elements, while maintaining only portions of the network settings in client memory. At any given time, the client stores at most $2n$ switch settings and routes elements through the network portions as the switch settings become available.

The full routing algorithm is split into two phases. The first phase, the configuration phase, recursively applies the SetExterior primitive (see Section 4.3) to compute the exterior switches of a branch of subnetworks. Recall that a subnetwork is itself a Waksman network comprised of an exterior of entry and exit switches. After each instantiation of SetExterior, the algorithm routes elements according to freshly set entry switches and, prior to writing elements back to the server — either swapped or not — encysts each element with the value of its associated exit switch. Consequently, portions of the computed network settings, that is, the exit switches, are moved to the server.

At the conclusion of the configuration phase, elements, following a sequence of entry switches, have been routed through half the network and are stored at the server with their upcoming (exit) switch values. Hence, the remaining routing can be done in the standard manner of retrieving pairs of elements and then either swapping or persisting them as per the switch setting. As a result, we call this second phase the empty road phase. We now describe the algorithm in more detail (pseudo-code is in Algorithm 2). A visualization of the configuration phase (ConfigurationRouting) is provided in Figure 6.

5.1 The configuration phase

At a high level, ConfigurationRouting utilises the binary tree decomposition of the network structure (Figure 3) to perform configuration and routing. The process is executed with a tree data structure, which is built and pruned in concert with a preorder traversal of the permutation tree. Each node in the tree $\pi$, refers to a subnetwork. The tree nodes at depth $l$ account for movement from the temporary array $V_l$ to $V_{l+1}$. Each node at depth $l$ shuffles a portion of $V_l$ with the level-wide shuffles combining to produce the temporary array $V_{l+1}$. A node contains the following attributes: a parent
Algorithm 3: The algorithm configures entry and exit switches of a subnetwork and routes elements according to entry switches. It sets switches of the sub permutation node $\pi$ before setting its left and right subpermutations. CreateNode sets up permutation node data structure including ancestor switch settings and information of where node is in the subnetwork tree. Attribute $\pi$.function is the permutation function of the corresponding node (e.g., it is $\pi$ if $\pi$ is a root node).

```
define ConfigurationRouting($\pi$)
  if $\pi$.size $\in$ \{1, 2\} then
    RouteLeafNode($\pi$) // Route elements along network wires according to $\pi$.function
  else
    ($\pi.E$, $\pi.X$) $\leftarrow$ SetExterior($\pi$.function, $\pi$.size)
    RouteElements($\pi$)
    $\pi_0$ $\leftarrow$ CreateNode($\pi$.depth + 1, $\pi$.offset, $\lfloor\pi$.size/2$\rfloor$, $\pi$)
    ConfigurationPhase($\pi_0$)
    $\pi_1$ $\leftarrow$ CreateNode($\pi$.depth + 1, $\pi$.offset + $\lfloor\pi$.size/2$\rfloor$, $\lfloor\pi$.size/2$\rfloor$, $\pi$)
    ConfigurationPhase($\pi_1$)
  delete $\pi$
```

($\pi$, parent); its depth in the tree ($\pi$.depth), the function that corresponds to the subpermutation ($\pi$.function); the size of the subnetwork ($\pi$.size) (i.e., size of permutation $\pi$.function); the offset that determines the portion of the temporary array that serves as input to the network, ($\pi$, has input $V_{\pi.\text{depth}}[\pi$.offset $\ldots\pi$.size $- 1]$); and the exterior network settings ($\pi.E$ and $\pi.X$).

The configuration phase builds the permutation tree during a preorder traversal. It begins by initializing the root $\pi$ with the input permutation $\pi$ constituting the attribute $\pi$.function. We then compute the exterior settings of the network $\pi.E$ and $\pi.X$ with the subroutine SetExterior($\pi$.function, $\pi$.size). At this point, elements are retrieved in pairs, packaged with their associated exit switches from $\pi.X$ and routed, through the entry switches $\pi.E$, into $V_l$. The process then recurses into the left child. First, a new node is created with CreateNode(depth, offset, size, parent):

$\pi_0$ = CreateNode($\pi$.depth + 1, $\pi$.offset, $\lfloor\pi$.size/2$\rfloor$, $\pi$)

Then, the process repeats; the exterior of the subnetwork is evaluated and elements are retrieved, packaged with their associated exit switches and routed, following the network wires, into the temporary array $V_l[0 \ldots \pi_0$.size$/2$ - 1] following the preorder traversal, the process recurses into the left child of $\pi_0$.

To avoid storing all the switch settings, pruning of the tree is performed throughout the traversal. Recall that the subpermutation function of a node depends on the exterior settings of all its ancestors (see Equations (1) and (2)). Thus, when the process recurses into a right child node, we can delete the left child. As a result, at most one branch of exterior network settings is stored at the client. Pseudo-code of the configuration phase is provided in Algorithm 3. Next, we detail the mechanics of the $\pi$.function.

5.1.1 Subpermutation evaluation. The exterior switches of each node $\pi_i$ are determined by applying SetExterior($\pi$.function, $\pi_i$.size). Though SetExterior operates conceptually in the same manner for each subnetwork, the input permutation functions are determined and computed differently depending on whether SetExterior is called for the global permutation $\pi$ (the root node) or a subpermutation at a deeper level (e.g., $\pi_{01}$).

At the root node, $\pi$.function is defined by an efficiently computable permutation, $\pi$, given to the algorithm as part of its input. However, the subpermutation function $\pi_i$ is not available to the algorithm, in the same form, at depths $l > 0$. To this end, the $\pi_i$.function utilizes the permutation tree to propagate the information required for its evaluation. At this point in the traversal, the exteriors of all ancestors are set. Therefore, following the logic of Equations (1) and (2), to evaluate $\pi_i$.function($i$), we check the $i$th entry switch at the parent and call its parent’s subpermutation function. This process repeats until we reach the root node.

5.1.2 Use of client and server storage. The algorithms use the temporary arrays $V_l$ to facilitate routing. The element array $V_l$ is always stored at the server, unless the client reads two elements and routes them according to a switch setting. The exterior switch settings of a node are initially stored at the client in the arrays $\pi.E$ and $\pi.X$ (Line 2 in Algorithm 2 and Line 5 in Algorithm 3). During the routing, entry switches are actioned, i.e., elements are routed accordingly, and exit switches are moved to the server, concatenated into the element blocks. After the node is traversed according to the preorder traversal, it is deleted (Line 11, Algorithm 3). Node deletion is permissible as the values of the exit switches have been moved to the server and elements have passed through the corresponding entry switches. We emphasize that the permutation tree is stored at the client and the primitive SetExterior operates locally and does not interact with the server. As observed in Section 4.2, at any moment, at most a single branch of exterior network settings is needed for evaluating a subpermutation function. This leads to a client storage of at most $2n + o(n) + 2B$ bits. In sum, only portions of the network settings are stored at the client and elements are routed as their paths through the subnetworks materialize.

5.1.3 Routing elements. RouteElements simply applies switch settings to the elements: it takes a pair of elements and either swaps them or does not. For completeness, Algorithm 4 provides pseudocode for RouteElements, and details the placement of the output switch settings at the server.
(a) The input to the configuration phase is an array $U$, stored at the server and a permutation $\pi$. At this stage, the network settings are not configured.

(b) The procedure begins at the root node ($\pi$) of the tree of permutations. The primitive SetExterior($\pi, n$) (Line 5 of Algorithm 3) is called to set the switch settings $E_\pi$ and $X_\pi$ at the client. The procedure routes the elements in $U$ according to switch settings in $E_\pi$. Routed elements are stored in $V_1$ with their output switch settings $X_\pi$.

(c) The procedure branches into the subnetwork $\pi_0$ and uses SetExterior($\pi_0, n/2$) to configure switch settings $E_{\pi_0}$ and $X_{\pi_0}$. Recall that evaluation of $\pi_0$ relies on the settings of $E_\pi$ and $X_\pi$. The elements in the first half of $V_1$ are then routed according to $E_{\pi_0}$. They are stored at the server in the first half of $V_2$ together with the $X_{\pi_0}$ settings. The algorithm proceeds to branch into $\pi_{00}$. When the procedure traverses into $\pi$, the arrays $E_\pi$ and $X_\pi$ can be deleted from client memory.

Figure 6: Visualization of client and server memory during WaksmanOP execution for $n = 8$. Configuration of network switches and routing of the elements proceeds simultaneously. Intermediate network settings are stored between the client and the server. The server storage is depicted as a sequence of $O(\log n)$ arrays. In practice, it suffices to use a constant number of arrays and alternate their usage. For example, with reference to Figure 6 (c), the first half of $V_2$ can be stored in $U$.

5.2 The empty road phase

At the end of the configuration phase, all elements have passed through the first half of the network and sit at the server packaged with the values of their upcoming exit switches. In turn, the empty road phase operates in the second half of the Waksman network and is relatively straightforward as all of its switches are set. We can iteratively retrieve element pairs belonging to switches, decrypt the switch setting and route the elements according to the network wires into a succeeding array.

In contrast to the configuration phase, during the empty road phase, we are moving from adjacent interior subnetworks into subsuming exteriors. The notion of the binary tree of subnetworks, introduced above to express and execute the configuration phase, is also instructive here. The procedure conducts a reverse level-order traversal; it begins at the leaf nodes and traverses the nodes from left-to-right, before moving to the next level. As sibling nodes form the interior of their parent, they are processed together. For example, when processing the siblings $\pi_{00}$ and $\pi_{01}$, with parent $\pi_{0}$,
Algorithm 4: Route elements acc. to switch settings in $\pi$.

1. **define** RouteElements($\pi$)
2. $d \leftarrow \pi$.depth
3. for $i \in \{0, 2, \ldots, 2^d - 2\}$ do
4.   $u_i \leftarrow \text{read}(V_d, \pi \cdot \text{offset} + i)$
5.   $u_{i+1} \leftarrow \text{read}(V_d, \pi \cdot \text{offset} + i + 1)$
6.   $j \leftarrow \pi \cdot \text{function}(i)/2; j' \leftarrow \pi \cdot \text{function}(i + 1)/2$
7.   // Route elements according to switches
8.   and write them out concatenated ($\circ$)
9.   with exit switch settings
10. if $\pi \cdot E_{i/2} = \text{persist} \; \text{then}$
11.   write($V_{d+1}, \pi \cdot \text{offset} + i, u_i \circ \pi \cdot X_j$)
12.   write($V_{d+1}, \pi \cdot \text{offset} + i + n/2, u_{i+1} \circ \pi \cdot X_{j'}$)
13. else
14.   write($V_{d+1}, \pi \cdot \text{offset} + i, u_{i+1} \circ \pi \cdot X_{j'}$)
15.   write($V_{d+1}, \pi \cdot \text{offset} + i + n/2, u_i \circ \pi \cdot X_j$)
16. if $\pi \cdot \text{size is odd and } \pi = \text{the root}$
17.   route $u_{n-1}$ into the next right-child with even size

we retrieve the $i^{th}$ switch from both subnetworks simultaneously and route the elements into the subsuming extent in tandem.

5.3 Security and Performance Analysis

In this section, we evaluate the security and performance of our algorithm. As mentioned earlier, we gauge performance through the amalgam of client memory and the number of I/Os. The trade-off between the two performance measures is at the center of our proposition that the Waksman network offers a viable solution to oblivious permutation in practice. Further, both client computation and server memory cannot be ignored and we discuss their implications under our approach.

**Client storage and computation.** For reasonable block sizes, the memory allocation is dominated by the storage of the network settings. Therefore, we begin our analysis with the configuration phase at the client.

**Lemma 3.** On input of $n$ elements of size $B$ each, the ConfigurationRouting completes in $2n + o(n) + 2B$ client memory and requires $O(n \log n)$ time of client computation.

**Proof.** To evaluate Equations (1) and (2), a node in the permutation tree requires the exterior network settings of all ancestors. As a preorder traversal of the permutation tree is performed, a node, along with its exterior network settings, can be deleted after it is visited for the last time, that is, when both its subtrees have been traversed. The exit-switch settings can be deleted ahead of time as they are moved to the server. Therefore, the client only stores a single (root-to-leaf) branch of the network settings. The memory in a node is dominated by the arrays storing the exterior-switch settings. We assume that the registers $\pi \cdot \text{size}$, $\pi \cdot \text{depth}$, $\pi \cdot \text{offset}$ and $\pi \cdot \text{parent}$ can be stored in $O(n \log n)$ bits. The arrays $\pi \cdot E$ and $\pi \cdot X$ are bitvectors of length ($\pi \cdot \text{size})/2$. The size of a node at depth $l$ is $n/2^l$. Hence, the memory required to store a branch is

$$
\sum_{l=0}^{\log n} 2 \cdot 2^{l} \cdot n/2^{l+1} + O(\log n) = 2n + o(n).
$$

In addition, by Lemma 2, at a node of size $\pi \cdot \text{size} = n^2$, SetExterior executes with $2n^2 + o(n)$ bits memory. This memory is split between $n^2$ bits for the arrays $E$ and $X$ and the rest for the temporary data used only during execution of SetExterior (to keep bitvectors with null settings). Therefore, SetExterior removes $n^2 + o(n)$ bits of memory from the client at completion, retaining the $n^2$ bits to store the exteriors $E$ and $X$. This occurs before the traversal proceeds in the next child node. Therefore, while traversing down a given branch, from the root, the client memory does not exceed $2n + o(n)$ bits. The algorithm requires $2B$ space to read two elements and either swap or persist them.

The runtime of SetExterior at node $\pi$, is dominated by $\pi \cdot \text{size}$ invocations of the node’s permutation function. At level $l$, there are $2^l$ nodes of size $n/2^l$ and the evaluation of the permutation function $\pi \cdot \text{function}$ takes $O(l)$ time. Summing over all levels, we arrive at the following runtime cost

$$
\sum_{l=1}^{\log n} 2^l \cdot n/2^l \cdot O(l) = n \sum_{l=1}^{\log n} O(l) = O(n \log^2 n).
$$

The computation can be thought of as requiring $O(n \log n)$ work per switch. In contrast, configuration algorithm provided by Waksman, in their original publication, takes $O(n \log n)$ time to complete, or rather, requires $O(1)$ work per switch. The increase in client computation can be considered as the cost of moving to a smaller client.

**Server storage.** Throughout the execution of the routing algorithm, the server contains $O(n \log n)$ temporary (partially shuffled and filled) arrays. With a naive implementation, this would lead to server cost of $O(nB \cdot \log n)$ bits. However, once a switch has read two elements from an array and written them as swapped or not — to the array at the next level of the network, the original two elements can be removed. Hence, as each element is stored in a single temporary array, server memory does not exceed $O(nB)$ bits.\footnote{For example, our implementation uses $4nB$ bits of server storage ($4$ arrays of length $n$)} We note that exit-switch settings are also stored at the server. However, since there are at most $O(n \log n)$ switches, and each switch stores a Boolean value, this sums to $O(n \log n)$ storage. Since $B = \Omega(\log n)$ to store at least the element’s index, switch settings do not increase server memory asymptotically.

**Security.** The obliviousness of the scheme, as per Definition 2, comes from the observation that the access pattern of the algorithm depends only on the switch layout of the network. In turn, the switch layout of the network depends only on $n$ and is independent of the input content. The algorithm performs two read and two write operations per switch (Algorithm 4). The algorithm depends on semantically secure encryption since the adversary observes not only memory accesses but also the encrypted content. As a result, it is secure against computationally bounded adversaries.
Observe that each switch demands two read and two write operations. By Lemma 1, the total I/O cost is $4n \log n - 3.6n$. Combined with Lemma 3, we arrive at the following theorem.

**Theorem 1.** ObliviousWaksmanShuffle$(U, \pi)$ (Algorithm 2) is an oblivious array permutation algorithm for an array of $n$-sized elements. The algorithm requires $2n + o(n) + 2B$ bits of client storage, $O(nB)$ server storage and completes in at most $4n \log n - 3.6n$ read/write operations.

### 6.2 Results

Our results are presented in Figure 7. We observe the general trend that WaksmanOP outperforms Bitonic sort in terms of the number of I/Os and uses significantly less client memory than the Melbourne shuffle. It also outperforms Bucket ORP under several parameter settings, especially those involving larger blocks. In addition, recall from Section 5, Bucket ORP has a weaker security guarantee than Waksman OP and does not achieve perfect obliviousness.

Figures 7a and 7b illustrate the client memory/I/O trade-offs for block sizes $B$, of 1KB and 100KB. The plots express a parameter elements/blocks stored at the server, $B$, from cache-line sized blocks of 64 bytes to blocks with sizes 1KB and 100KB, representative of object sizes for some cloud storage types [3]. As bitonic sort, Bucket ORP and Waksman all perform best when $n$ is a power-of-two, for a fair comparison, we select problem sizes outside the best-case scenario.

### 6.1 Experimental setup

All code is written in C++\(^6\). We simulate the client and server on a single machine. The algorithms can create and delete arrays or read and write to a given array. The server is simulated by an interface that abstracts array management: all data is stored and retrieved on disk. The number of I/Os is counted internally in the code. We note that this simulation is adequate for measuring the number of I/Os and client storage since these metrics are deterministic in the setting of oblivious algorithms and do not depend on client/server implementation or the content of the data. Since the Melbourne shuffle is a randomized algorithm and may fail during execution, we set its failure probability upper bound to $2^{-80}$.

\(^6\)Code is available at https://github.com/wCloudRain/orp.
space in which no algorithm is superior in both measures. In other words, there is an efficient frontier of algorithms and the choice of algorithm would depend on the setting. However, WaksmanOP is always superior to other methods in at least one of the measures (I/O or client space) and is strictly better than Bucket ORP for certain parameter values (larger values of $n$ and $B$). We emphasize that the client memory allocations for the WaksmanOP are feasible for the tested parameter space. In Figure 7b, experiments with $n$ data points show the performance of the algorithms when permuting a $10000 \times 4^7 \times 100KB$ (16TB) dataset. Here, the WaksmanOP requires only 41MB of client memory. This is in contrast to the 6.4GB memory allocation for the Melbourne shuffle.

We compare client memory requirements in Figures 7d–7f. The memory allocation for bitonic sort is independent of the problem size. For WaksmanOP, the memory allocation is dominated by the block size and a constant factor overhead at small values of $n$. Once these values become negligible, the client memory begins to grow linearly in the problem size. For fixed $B$, the Melbourne shuffle has $O(\sqrt{B})$ client memory. Compared to the Waksman shuffle, the superior asymptotic efficiency is expressed in the plot lines of Figures 7d and 7e. However, the client memory allocation of the Melbourne Shuffle is considerably impacted by the block size. In Figure 7f, for a 100KB block, the Melbourne Shuffle client memory allocation is at least 500 times larger than the Waksman shuffle for problem sizes $n \leq 10^7$, i.e., outsourced data of around 1TB. The memory allocation for Bucket ORP is independent of the problem size, but is impacted by the block size. For 100KB blocks, Bucket ORP requires 100MB of client space.

The number of I/Os as a function of the problem size is displayed in Figure 7c. Compared to the linear number of I/Os incurred by the Melbourne shuffle, WaksmanOP and Bucket ORP require an additional $O(\log n)$ factor of I/Os and this is reflected in the figure. Due to constant factors, the WaksmanOP is superior to Melbourne shuffle for values $n \leq 10^5$. The I/O cost for Bucket ORP grows slightly faster than WaksmanOP and incurs a higher cost than WaksmanOP for $n > 10^6$. For large values of $n$, the WaksmanOP is 12 times faster than the bitonic sort and 2.5 times slower than the Melbourne shuffle. Note that number of I/Os for each of the four algorithms is deterministic and independent of the block size $B$.

7 CONCLUSION

In this paper we revisited a fundamental data-oblivious primitive — array permutation — and developed a solution that effectively trades off client space and number of I/Os. Our solution, WaksmanOP, is based on the Waksman network and includes an efficient algorithm that carefully configures its switches using small client space. Both the proven asymptotic behaviour of the algorithm, and our experimental results, show that under several realistic scenarios (regarding size of each data block and number of blocks), compared to baseline solutions, WaksmanOP is the best trade-off.

ACKNOWLEDGMENTS

William Holland is in part supported by the Australian Government RTP Scholarship Research Training Program. Olga Ohrimenko is in part supported by a Facebook research award. Anthony Wirth is in part supported by the Australian Research Council, grant number DP19102078.

We thank the anonymous reviewers for their feedback on the paper. We acknowledge the Wurundjeri and Bunurong Peoples of the Kulin Nations as traditional owners of the land on which we live and work.

REFERENCES


Chapter 7

Conclusion and Future Directions

Our thesis presents a study on recency. We cover two types of recency. Informally, these types correspond to the following questions: For item $x$,

- How many *items* have been observed since $x$ last occurred on the stream?
- How many *arrivals* have been observed since $x$ last occurred on the stream?

These questions are difficult to evaluate with limited memory. Our work establishes new methods that overcome this challenge and provides new theoretical insights into handling temporal queries in resource constrained environments. In addition, we expand the scope of a recency query and demonstrate how our methods can be leveraged to achieve performance improvements in the field of oblivious storage. To summarize our contributions in more detail, the following subsection responds to each research question directly. The high level summary of contributions is presented in Table 7.1.

Evaluation of Research Questions.

RQ1: Is there a succinct representation of an indexed list that admits queries and updates in optimal time?

The content of Chapter 3 corresponds to our answer to Research Question 1. The question arose when investigating solutions to item-recency in small-memory. The standard approach to solving item-recency is to store item arrivals in a move-to-front list. When a new update occurs, the observed item is placed at the head of the list. Therefore, item-recency is calculated by counting the distance of the queried item to the head. To avoid linear scans, auxiliary structures can be used to accelerate searching and updates. One data structure that does this efficiently is the indexed list. However, existing solutions, by opting for extensive use of pointers and a supplementary hash-table, are memory-inefficient. Therefore, we were interested in whether the operations of the indexed list could be supported with less memory.

Our solution, the PrincessList is a template for an indexed list structure. For a concrete instantiation, existing dynamic string and indexed list solutions can be used as a black box. The template is an uncomplicated approach to list indexing. The algorithms are simple and
The Recency Problem and its Applications

William L. Holland

<table>
<thead>
<tr>
<th>RQ</th>
<th>Aim</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Item-recency query (Definition 2) in small memory</td>
<td>PrincessList: First succinct indexed list with optimal time operations. First method to handle strings with a very large alphabets.</td>
</tr>
<tr>
<td>2</td>
<td>Time-recency query (Definition 1) in small memory</td>
<td>HistoricalMembership: Recency queries with bounded relative error and a succinct representation.</td>
</tr>
<tr>
<td>3</td>
<td>Application of recency queries in Hierarchical ORAM</td>
<td>RankORAM: First practical Hierarchical ORAM. Logarithmic reduction in memory vs. state-of-the-art practical ORAMs</td>
</tr>
</tbody>
</table>

Table 7.1: High level summary of contributions.

offer immediate improvements over solutions that employ linear traversals. To respond to the research question, we provide a concrete instantiation of the template (PrincessList+) that obtains a succinct representation of the list with high probability. Importantly, the operation times are optimal and match the prior state-of-the-art. Therefore, we are able to reduce memory, against existing art, without impacting runtime performance. To our knowledge, the only approach that (asymptotically) matches our memory allocation is a packed array. The cost of updating a packed array is linear, the same as a linked-list. For comparison, PrincessList+ supports all operations in $O(\log n / \log \log n)$ time.

As an additional contribution, if we allow for repetitions, the PrincessList+ can maintain a succinct representation of a dynamic string in a (high-entropy) setting where the size of the alphabet exceeds the length of the string. Prior works have not managed to handle this case.

**Future Work** There are a number of directions for further work. First, all prior solutions to list indexing admit update operations that are amortized. Thus, even in the non-succinct case, designing worst-case update operations is an open problem. The challenge with succinct dynamic data structures is that they need to be periodically rebuilt. With amortized updates, this is generally not an issue, as the full rebuild can be conducted in between two updates at a small amortized cost. To achieve efficient worst-case behaviour, the structure is gradually rebuilt behind-the-scenes between updates. This has to be done without exceeding the memory bound and there are a number of precedents for successful rebuilds [75].

Second, the PrincessList+ does not obtain, a succinct representation with probability one. Therefore, achieving a non-probabilistic memory allocation with optimal time operations would conclude this line of work.

Lastly, the PrincessList template is simple and easy to implement. This is also the case for some solutions to list indexing, such as the binary-tree approach detailed in Figure 2.7. Currently, there is no experimental evaluation of these techniques, even though the index and
position operations are staples in many programming languages. Therefore, such an evaluation would be of value. We have done some preliminary work in this direction. Early results indicate that both the PrincessList and the binary-tree index list outperform, in runtime, standard techniques, such as dynamic arrays and linked-lists, by orders of magnitude. Measuring the memory reduction in practice would also be an interesting result.

**RQ2: What is the memory lower bound for a time-recency query? Can a summary of the stream meet this lower bound and support efficient query and update operations?**

There is a concession to this research question. One of the challenges of measuring time-recency, with memory efficiency, is the unbounded nature of the stream. In this setting, the size of the time-stamps are unbounded and the number of distinct items, which could be drawn from a large universe, can grow to an intractable measure. Therefore, we restricted time-recency queries to a sliding window of the most recent updates. This restriction is justified. The sliding-window model is well-established and we are often not interested in “old” items. In this instance, it is sufficient to say that an item has not occurred recently.

Under this revised query, summarized in Problem 2, we present HistoricalMembership, which supports recency queries with bounded relative error on top of a succinct representation of the items on the window. The logic of HistoricalMembership is tied to the *impression* of an equivalence class, wherein items occurring at an equivalent moment in history can be assigned the same estimate. Existing work, from the sliding membership literature, provides solutions to Problem 2, but cannot accommodate a combination of accuracy and small memory. On the one hand, ExactCuckoo supports exact queries in large memory and, on the other, OptimalSM obtains a succinct representation, but with approximate queries with only bounded *absolute* error. Therefore, OptimalSM cannot provide strong accuracy guarantees on more recently occurred items. Interestingly, our data structure matches the lower-bound, established by Naor *et al.* [111], for the sliding-membership problem. As recency solves sliding membership, our memory allocation is optimal. Our update and query operations are logarithmic, compared to constant time for our competitors. However, we still deem this outcome favourably with respect to the research question.

**Future Work** If we think of accuracy-space-time as a triangle, the data structures ExactCuckoo, OptimalSM and HistoricalMembership each occupy a unique face. This leads to the question as to whether operations for Recency data structures can be supported in constant time without sacrificing the memory and accuracy attributes of HistoricalMembership. Alternatively, the three data structures represent the contours and boundaries of the Recency problem.

Similar to Chapter 3, an experimental evaluation is not conducted. However, we have done some preliminary work in this direction. On synthetic power-law data, against baselines with similar accuracy, HistoricalMembership observes a factor-14 decrease in memory on low skew streams and a factor-120 decrease on high skew streams.
RQ3: To store a data set of \( n \) blocks, the class of practical ORAMs achieve optimal online bandwidth and require \( \Omega(n \log n) \) bits to store a position map in client memory. With HistoricalMembership, is it possible to reduce this client memory overhead to \( O(n) \) bits without degrading bandwidth performance?

Hierarchical ORAMs share an affinity with the recency query. The execution of an access depends on the recency of the accessed block. Research Question 3 emerged from this observation. With RankORAM, building on techniques from HistoricalMembership, we construct the first practical Hierarchical ORAM scheme. In a trade-off, we exploit a larger client memory allocation, relative to existing Hierarchical ORAMs, to achieve improved bandwidth and latency performance. Significantly, HistoricalMembership can be used in any Hierarchical ORAM to reduce the number of round trips of communication, per access, from \( \log n \) to one. Further, by supporting rank queries to the oblivious hash tables (stored at the server), HistoricalMembership allows levels to be rebuilt with fast and practical oblivious shuffle algorithms. Our first contribution is this addition to the tradition of Hierarchical ORAM.

Returning to the research question, compared to state-of-the-art practical solutions, PartitionORAM and RingORAM, we reduce client memory by a logarithmic factor, achieving the desired \( O(n) \)-bit bound. This is accomplished while maintaining comparable bandwidth and latency performance. The standard for practical ORAMs is to use an array to store position maps at the client. Our experiments, on real network file system traces, demonstrate a reduction in client memory by a factor of a 100 against the array approach. In addition, we investigate the feasibility of position map compression techniques hypothesized in prior work [140]. The technique has poor worst-case behaviour and is outperformed by HistoricalMembership except when the access pattern has very high skew. However, experiments demonstrate that the technique is viable on favourable access patterns.

**Future Work** The most pertinent direction of future work involves expanding the experimental evaluation. Our exposition is limited to simulating the client-side data structures against different workloads. It would be illuminating to implement the full protocols (for RankORAM, PartitionORAM and RingORAM), similar to Chang et al. [35], and measure their performance deployed in a realistic client-server setting. Further, with a fully deployed ORAM, we can test the trade-offs involved when using compression vs. recursion to store client-side metadata.

RQ4: Follow existing implementations, the Waksman network requires \( O(n \log n) \) bits of client memory. For Oblivious Permutation, can the client memory overhead be reduced? Further, does this lead to performance improvements in practice?

Our final research question revisits a fundamental data-oblivious primitive: array permutation. RankORAM operates with \( O(n) \) bits of client memory. This is not a memory allocation seen in prior work on oblivious permutation. Under this observation, we explore a new trade-off, between client memory and bandwidth, in the oblivious permutation problem.

Our solution, WaksmanOP, is based on the Waksman network. The primary innovation is a novel routing algorithm that carefully configures the network switches using small client space. The high-level technique, which entails configuring and routing portions of the network
at a time, can be applied to other permutation networks, such as the Benes network [112].

We reduce the routing cost to $2n + o(n)$ bits, a logarithmic improvement against existing routing algorithms. The significance here is that, with respect to client memory, we remove the dependence on the block size incurred by the bandwidth efficient shuffling algorithms [117, 123]. Our experiments illustrate the effectiveness of our approach. For large block sizes (of 100KB) and reasonable sized problem instances (less than or equal to $10^7$), WaksmanOP observes a client-memory allocation smaller than MelbShuffle by a factor of 500. This comes at the small cost of a factor-2 increase in bandwidth. On the other hand, WaksmanOP uses more client memory than the Bitonic sorting network, the most bandwidth efficient sorting network for reasonable problem sizes, but reduces the bandwidth cost by a logarithmic factor. Consequently, our experimental results show that, under several realistic scenarios, compared to baseline solutions, WaksmanOP is the best trade-off.

The routing algorithm can be used in any application of the Waksman network that doesn’t require all the switches to be evaluated a priori. For example, Chen et al. construct a homomorphic permutation algorithm, for an ORAM scheme,

**Future Work**  A promising area of future work is the application of WaksmanOP for RankORAM. Chen et al., for a tree-based ORAM scheme, developed a homomorphic permutation algorithm based on the Waksman network [37]. At a high-level, the construction works in three phases:

1. The client computes the network settings for a given input permutation.
2. The client encrypts the switch settings, packing multiple settings into each cipher text, and sends them to the server.
3. The server homomorphically executes the permutation on behalf of the client.

The bandwidth cost is equal to the cost of uploading the cipher-texts and is less than the cost of routing the elements in a traditional manner. It also requires a single round trip of communication, compared to $O(n \log n)$ trips for standard routing. This approach works well in tree-based ORAMs as shuffles only occur on a constant number of blocks. Therefore, the client is not burdened with a large memory allocation when configuring the network.

This algorithm can be used in the shuffle phase of Hierarchical ORAM to reduce the bandwidth cost of the rebuild phase. Essentially, rebuilds could be executed without downloading any blocks from the server. However, following the method of Chen et al., this would require a $O(n \log n)$ bit memory allocation. Alternatively, with the algorithm for configuring network portions, contained within WaksmanOP, the network settings could be configured and encrypted in batches and sent to the server. This would require $2n$ bits of private memory, making it a prime candidate for RankORAM.

The key to this direction of work is the cost of producing the cipher-texts. The levelled fully homomorphic encryption scheme produces larger cipher-texts than the semantically secure symmetric encryption schemes typically used in ORAM. Further, it requires extensive computation on behalf of the client and server and increases the amount of storage at the server. Evaluating these costs, relative to WaksmanOP, is an important open question. If successful,
the low-memory homomorphic Waksman permutation could be deployed in RankORAM for significant reductions in bandwidth in practice.
Bibliography


Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
Holland, William Lauchlan

Title:
The Recency Problem and its Applications

Date:
2022

Persistent Link:
http://hdl.handle.net/11343/318026

Terms and Conditions:
Terms and Conditions: Copyright in works deposited in Minerva Access is retained by the copyright owner. The work may not be altered without permission from the copyright owner. Readers may only download, print and save electronic copies of whole works for their own personal non-commercial use. Any use that exceeds these limits requires permission from the copyright owner. Attribution is essential when quoting or paraphrasing from these works.