String Searching using External Memory

by

Cory Fraser

A Thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for
Computer Science

Department of Computer Science
Carleton University
Ottawa, Ontario, Canada
October 2016

Copyright ©
2016 - Cory Fraser
The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the Thesis

String Searching using External Memory

Submitted by Cory Fraser
in partial fulfilment of the requirements for the degree of

Master of Computer Science

P. Morin, Supervisor

A. Maheshwari, Department Chair

Carleton University
2016
Abstract

The amount of String data around the world is constantly growing. As more of it accumulates, methods to search through it, and at higher capacities needs to be continually improved. This thesis analyzes string searching solutions that: support dynamic updates, support prefix searching, and can perform well using external memory. These solutions are compared against each other using different types of string data while performing various operations. The strengths and weaknesses of each solution is then identified and discussed.
# Table of Contents

Abstract iii

Table of Contents iv

List of Tables vii

List of Figures viii

Nomenclature x

1 Introduction 1

1.1 Motivations and Objectives 1

1.2 Thesis Contents 4

2 Background 6

2.1 Common Data Structures 6

2.1.1 Hash Tables 7

2.1.2 Binary Search Trees 8

2.1.3 Tries 11

2.1.4 Skip Lists 13

2.2 External Memory Concepts 14

2.2.1 Cache Oblivious Algorithms 15

2.2.2 Partition Based Algorithms 17
5.8 Results Summary ........................................... 85

6 Conclusion ........................................... 87

   6.1 Directions for Future Work ....................... 89

List of References ................................... 91
List of Tables

5.1 Compression Disk Space Comparison . . . . . . . . . . . . . . . . . . 40
5.2 B+ Tree Variant Space Consumption . . . . . . . . . . . . . . . . . . 53
5.3 B-Patricia Trie Variant Space Consumption . . . . . . . . . . . . . 72
## List of Figures

2.1 Hash table storage structure [1] ........................................ 8  
2.2 Treap Example Insertion [1] ........................................... 10  
2.3 Van Emde Boas Index Layout Example ............................... 16  
5.1 Insertion Comparison with Compression ............................. 42  
5.2 Search Comparison with Compression ............................... 44  
5.3 Prefix Search Comparison with Compression ....................... 45  
5.4 Removal Comparison with Compression ............................... 47  
5.5 B+ Tree Sequential Input Branching Comparison .................. 49  
5.6 B+ Tree Word Input Branching Comparison ....................... 50  
5.7 B+ Tree Random Input Branching Comparison ..................... 51  
5.8 B+ Tree Variant Comparison, Insertions ............................ 52  
5.9 B+ Tree Variant Comparison, Searches ............................. 55  
5.10 B+ Tree Variant Comparison, Prefix Searches .................... 56  
5.11 B+ Tree Variant Comparison, Removals ............................ 57  
5.12 B-Skip List Sequential Input Promotion Comparison ............. 60  
5.13 B-Skip List Word Input Promotion Comparison .................. 61  
5.14 B-Skip List Random Input Promotion Comparison ............... 62  
5.15 B-Skip List Variant Comparison, Insertions ..................... 63  
5.16 B-Skip List Variant Comparison, Searches ....................... 65  
5.17 B-Skip List Variant Comparison, Prefix Searches ............... 66
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.18</td>
<td>B-Skip List Variant Comparison, Removals</td>
<td>67</td>
</tr>
<tr>
<td>5.19</td>
<td>B-Patricia Trie Sequential Input Partition Size Comparison</td>
<td>69</td>
</tr>
<tr>
<td>5.20</td>
<td>B-Patricia Trie Word Input Partition Size Comparison</td>
<td>70</td>
</tr>
<tr>
<td>5.21</td>
<td>B-Patricia Trie Random Input Partition Size Comparison</td>
<td>72</td>
</tr>
<tr>
<td>5.22</td>
<td>B-Patricia Trie Variant Comparison, Insertions</td>
<td>73</td>
</tr>
<tr>
<td>5.23</td>
<td>B-Patricia Trie Variant Comparison, Searches</td>
<td>75</td>
</tr>
<tr>
<td>5.24</td>
<td>B-Patricia Trie Variant Comparison, Prefix Searches</td>
<td>77</td>
</tr>
<tr>
<td>5.25</td>
<td>B-Patricia Trie Variant Comparison, Removals</td>
<td>78</td>
</tr>
<tr>
<td>5.26</td>
<td>Data Structure Comparison, Insertions</td>
<td>80</td>
</tr>
<tr>
<td>5.27</td>
<td>Data Structure Comparison, Searches</td>
<td>81</td>
</tr>
<tr>
<td>5.28</td>
<td>Data Structure Comparison, Prefix Searches</td>
<td>83</td>
</tr>
<tr>
<td>5.29</td>
<td>Data Structure Comparison, Removals</td>
<td>84</td>
</tr>
</tbody>
</table>
Nomenclature

AMD - Advanced Micro Devices
API - Application Programming Interface
CPI - Central Processing Unit
DDR3 - Double data rate type three
DNS - Domain Name System
GZIP - GNU ZIP
I/O - Input/Output
IDE - Integrated Developer Environment
IP - Internet Protocol
JAR - Java Archive
JSON - JavaScript Object Notation
MB - Megabyte
MD5 - Message Digest 5
OS - Operating System
RAM - Random Access Memory
RPM - Revolutions per Minute
SSD - Solid State Drive
XML - Extensible Markup Language
Chapter 1

Introduction

1.1 Motivations and Objectives

String searching is a commonly encountered problem in Computer Science. This includes locating sub-strings within a large body of text such as a book or article. In this case creating an index on the body of text improves the speed of locating sub-strings. Another example of string searching is when creating a mapping from a string to another piece of data. A phone book maps the name of a person to their phone number, a dictionary maps a word to its definition, a Domain Name System (DNS) maps a website host name to an Internet Protocol (IP) address. In each of these example mappings, string searching involves finding the matching string key or keys.

Another often useful capability related to string searching is prefix searching. Prefix searching allows for specifying a prefix and retrieving all strings which have the specified prefix. Prefix searching is used frequently with content assist and auto-complete features which are often seen in web browser search bars or Integrated Developer Environments (IDE) such as Eclipse.

It can be seen that there are many applications where string searching is necessary which has led to many different types of solutions each with different trade-offs.
One variation of solutions involves whether or not the stored searchable strings is a static set or if the set of strings can be dynamically changed. A dynamic solution allows for the adding of new strings or the removal of existing strings from the set or map. A dynamic solution can be especially useful for the string mapping applications described above where existing mappings can become out of date frequently and new mappings continuously need to be added.

Another variation of solutions is whether or not the solution can make use of external memory, such as hard disks, for storage or if only regular Random Access Memory (RAM) can be used. The implementation complexity of RAM only solutions is usually much simpler which makes these solutions useful when the string data can easily fit into RAM. There are many cases however, where the string data cannot fit entirely into RAM. All of the mapping applications described above most likely fall into this category when considering a global scale. For these cases, a solution that can make use of external memory is required.

Any RAM based structure can be made to use external memory. A basic example would be to replace any pointers to data with file addresses containing the serialized data instead. This example would end up generating a large number of files and would often require that many of these files be accessed to complete a single operation. Since external memory is generally many orders of magnitude slower than internal memory, this example would perform very poorly in practice. Therefore, it is also required that solutions discussed in this thesis make use of external memory efficiently.

It can be seen that string searching refers to a large problem space. In order to perform an in depth analysis to pieces of the problem, this thesis imposes restrictions on the types of solutions that are considered. A candidate solution must support: prefix search functionality, dynamic data modification, and make use of external memory in addition to RAM.
Even with the restrictions stated previously, there are still many possible solutions. These solutions often come in the form of a data structure. A data structure hides the complicated details of an implementation and only publicly exposes simple operations such as: add, remove, contains, or prefix search. This thesis focuses on comparing proposed data structures against each other in order to identify cases where a particular structure performs better than another. A structure can perform better in terms of space usage or run time with respect to either its add, contains, prefix search, or remove operations.

Often, a proposed data structure is vague or unspecific on certain implementation details. This allows for different variations of the structure to be implemented with alternate solutions to the vague area. Also, sometimes a proposed structure can be modified slightly to produce a variation that may perform better in certain cases. This thesis examines several variations to the proposed data structures to gain a more thorough performance analysis.

There are many aspects of an implementation that can affect performance. Some of these aspects include: programming language, data serialization algorithm, data compression algorithm, and disk caching strategy. There are many options for each of these aspects which can have their own strengths and weaknesses. The focus of this thesis is on comparing various data structure algorithms and thus only minimal attention is spent of the aforementioned aspects. The algorithms for each of these aspects are maintained across all the examined data structures to help focus on the data structures themselves.

As mentioned previously, string searching can involve long bodies of text, such as books, or shorter word length strings, such as a host name. This thesis focuses on the shorter length word based strings in the evaluation. One reason for this is that string mappings tend to involve smaller length strings. Another reason is that a large body of text can be broken up into several smaller strings by a process called tokenization.
and then the smaller strings searched instead. A simple example of tokenization is creating a new substring at every space character found in a body of text.

When analysing the performance of algorithms, it is often the case that a specific algorithm will have strengths and weakness depending on the patterns of input given. To provide a more comprehensive analysis, it is necessary to experiment with different patterns of input data. This thesis performs a comparison of data structures using different patterns of string data.

1.2 Thesis Contents

This thesis consists of six chapters. This chapter has described and motivated the area of research making clear which pieces of the string searching problem are to be examined.

Chapter two looks at the background of string searching. Existing solutions to aspects of the problem as well essential concepts that are helpful in analysing related problems are described here.

Chapter three describes the data structures that are evaluated in this thesis. This will include a thorough description of the implementation details. Also, any custom variations implemented in addition to the standard algorithm are described here along with a justification for it.

Chapter four discusses the details about the implementation that remain constant across all data structures. This includes details such as: programming language, data compression, data serialization, and disk caching strategy.

Chapter five contains the performance analysis of the implemented data structures. The test system is described here. All measured performance data is presented for the various experiments performed. The results are summarized and conclusions are drawn from the comparison.
Chapter six provides a conclusion to the thesis. This includes a summary of the obtained results as well as directions that future research into the problem could take.
Chapter 2

Background

This chapter discusses some of the previous work that has been done in regards to string searching. Much of this work comes in the form of data structures. Several different common data structures that can be used in string searching are looked at.

This chapter also discusses some essential concepts that should be considered when dealing with external memory. Some existing techniques and data structures that help make use of external memory are also examined.

2.1 Common Data Structures

When looking for string searching algorithms to compare experimentally, it is helpful to look at existing solutions for aspects of the problem. Many of these solutions come in the form of data structures. These data structures store the string data using various techniques behind the scenes and only expose easy to use operations. The exposed operations commonly include: adding a string, removing a string, and checking whether or not a string is contained with the structure. Some less commonly supported operations are finding all strings containing a specified prefix or finding all stored strings that lexicographically compare between two input strings, which is commonly referred to as range searching.
A range search operation can act as a prefix search operation if the input range is chosen correctly. For example, an input range of 'aa' to 'ab' would return all strings that start with the prefix 'aa'. Since a range search can act as a prefix search and this thesis only compares solutions which support prefix searching, any solution which supports a prefix search or range search operation can be considered.

2.1.1 Hash Tables

There is a class of data structures that doesn’t support the prefix search operation requirement. This class of data structures make use of hash functions to compute how to store a string. A hash table or hash map is a common data structure that makes use of hash functions. In a hash table, a hash function is run against an input string to compute an array index which is where the string will be inserted at in an array. If a string is already stored in the computed index, the string can be stored in the next sequential unused index. Another option is that each index of the array can be made to store a list of elements and so index collisions would just grow the list size beyond one. Figure 2.1 shows an example of this type of hash table.

A particularly interesting feature of hash tables is that their add, remove, and search operations all have an amortized time complexity of $O(|s|)$, where $|s|$ is the length of the input string, while requiring $O(|S|)$ space usage, where $|S|$ is the total length of all strings in the structure combined [2]. These constant time complexities come at the cost of it being difficult to iterate over elements in sorted order. Iterating over array indices in order would give results that are close to random. It is because of this that hash based solutions cannot support prefix search operations.

Hash map data structure implementations are part of standard libraries for many programming languages including Java and C++. These implementations handle issues such as: automatically growing the array as required, and ensuring that no sub list within an array index grows too large.
2.1.2 Binary Search Trees

Another very common type of data structure for handling string searching is a binary search tree. A binary search tree performs comparisons of the input string against strings already stored in the tree. Each comparison ideally eliminates half of the stored strings from the search. A comparison of a new value against the value of a node in the tree usually results in either: traversing the left branch if the new value is smaller than the node’s value, traversing the right branch if the new value is larger than the node’s value, or ending the operation with a match when both values are equal. When adding a string to the tree, comparisons are performed until a complete branch of the tree is traversed.

A common problem with dynamically inserting strings into a binary search tree is that it’s branch structure can become unbalanced. The more unbalanced a tree becomes, the slower operations performed on the tree will become. A fully balanced tree is able to perform insert and search operations with a $O(|s| \log n)$ time complexity while a completely unbalanced tree will perform these operations with a $O(|s|n)$ time complexity.
complexity. Many variations of binary search trees have been created to address this problem of balancing.

One common variation of a self balancing binary search tree is called a Red-black tree. A Red-black tree involves colouring every node in the tree either red or black where each colour has a set of rules that must be followed. The rules this tree must follow are the following: the root is black, all leaves including nil leaves are black, every red node must have two black children, and every path from a given node to any of its descendant leaves must contain the same number of black nodes. New nodes that are inserted into the tree are coloured red which can end up breaking one of the rules. Similarly, when a node is removed from the tree, the rules of the tree can be broken. To bring the tree back into a state where it’s following all the rules, the nodes around the affected node and their colours are looked at to determine what kind of case the state of the tree matches. There are numerous cases that the tree can match for both insertion and removal operations which can make implementing a red-black tree an error prone process. When the case is identified, a combination of re-colouring nodes and tree rotations are used to correct the tree’s state.

Since a red-black tree is a common type of self balancing tree, existing implementations are not difficult to find. For example, the tree based data structures in the Java programming language are based off of a red-black tree.

Another variation of a self balancing binary search tree is called a Treap. Where a red-black tree relies on identifying what kind of case the state of the tree matches to balance itself, a Treap relies on randomization to keep itself balanced. Every node in the Treap stores a random number as an extra piece of information. The random number is used to determine how far down the tree a node should be. When a new node is inserted, it is first inserted as a leaf node as would normally happen in a regular binary search tree. Then the random number of the new node is compared with it’s parent node. If the new node’s number is greater than it’s parent’s, then a
Figure 2.2: Treap Example Insertion [1]

tree rotation is used to make the new node the parent while maintaining the correct element ordering that a binary search tree requires. The new node continually checks its parent’s random number until the new node’s random number is lower than its parent’s. This use of randomization instead of considering cases makes implementing a Treap much simpler than a Red-black tree.

Using randomization to determine the depth of a new node in the tree makes it improbable that long paths of nodes will form and probable that nodes will distribute fairly evenly across all branches of the tree. A Treap has an $O(|s| \log n)$ expected time complexity for its insert, remove, and search operations [3].

An interesting feature of a Treap is its ability to perform split and merge operations in $O(|s| \log n)$ expected case time complexity. Both of these operations consist of a single insertion operation. To split a Treap on a value $x$, insert a node with the value $x$ and a maximum value random number. Once this insertion is complete, the node with value $x$ will be the root of the tree and the two children sub-trees will be the split results. To merge two Treaps, where the lowest value $a$ in one Treap is higher than the highest value $b$ in the other, create a Treap where its root is a node with a value between $a$ and $b$ with a minimum valued random number. The two Treaps should then be attached to this node as children, the root node rotated down to a leaf position in the tree, and then deleted. The resulting Treap is the merged combination of two initial Treaps.

Both the Red-black tree and Treap data structures make use of an operation called a tree rotation help correct imbalances. A rotation allows a node to become a child
of one of its children while maintaining the comparison order property of a binary search tree. There are two types of tree rotations: a left rotation, and a right rotation. A left rotation makes a parent node become the right child of its left child while a right rotation makes a parent node become the left child of its right child. The child node of the new parent that the old parent node replaces becomes a child of the old parent in place of where the new parent used to be.

Since the elements in any type of binary search tree are stored in order, any binary search tree is capable of prefix searching by using a range search. A range search can be performed by searching for the nodes representing the beginning and end of the range. All nodes stored between those two nodes can then be returned as the result.

### 2.1.3 Tries

Another type of data structure that handles string searching well is a trie. A trie is a tree type structure where each branch, or edge, of a node represents a letter of the alphabet. A trie node commonly stores its edges to children nodes as an array. This strategy works well if the maximum number of children, or the size of the alphabet, is known and small. For example, if digits are the only characters that are to be stored in a trie then children array sizes of only ten are needed. If on the other hand, any unicode character could possibly be stored in the trie, then an array size of $2^{16}$ is needed which makes the memory footprint of a single node impractical. One such way to get around this limitation is to use a growable data structure with constant time access operations such as a hash map or hash table instead of an array to store children edges.

Another way to get around having to store large numbers of edges to children is to think of the stored data as bit strings instead of character strings. Each node can only have at most two children, one representing the zero bit as the next bit, and the other representing a one as the next bit. Using a binary version of a trie essentially
allows for many types of data to be stored within a trie since the only requirement is that the data have a bit string representation.

Each branch of the trie represents a character, or bit, of a stored string. This means that when searching for a string, the search would have to reach a depth of at most the length of the search string. In other words, if $|s|$ is the length of a search string, the time complexity needed to perform the search is $O(|s|)$. Similarly, the time complexity to perform insert and remove operations on a trie is $O(|s|)$. Nodes in a trie structure can have a boolean flag associated with them that indicates whether the current node represents the end of a string stored within the trie or are just a prefix of other strings stored within the trie. Thus, when inserting or removing a string that ends with a node that isn’t a leaf of the tree, the only update needed is to flip the boolean flag of the ending node. Another way that a trie implementation can represent the end of an inserted string is to have the final node store a null or terminating character.

A common variation to a trie data structure is to combine multiple characters into each edge or node. Common names for this variation of trie is Patricia Trie, or Radix trie. This would allow for a large string to be inserted requiring only one node added to the trie. This variation requires a more complicated algorithm during updates but reduces the space and time overhead by greatly lowering the number of nodes that have to be traversed in a search as well as the overall number of nodes stored in the trie. During an insertion, a new node may need to be inserted in the middle of the trie to split an existing edge. During a removal, a non-leaf node may need to be removed, connecting it’s parent to it’s child.

A trie’s structure makes it especially easy to perform prefix searches. The sub-tree rooted at the node or edge found when searching for a prefix will contain all the possible prefixes stored within the trie. A Patricia trie can perform a prefix search with a time complexity of $O(m + k)$, where $m$ is the size of the prefix and $k$ is the
2.1.4 Skip Lists

A Skip List is another data structure that can handle string searching well. A Skip List typically makes use of Linked Lists and randomization to provide insertion, removal, and search operations with an $O(|s| \log n)$ expected time complexity [4]. A skip list consists of multiple layers of sorted linked lists where the bottom layer is a regular linked list containing all stored data elements. Each layer above the bottom contains only a subset of the total stored elements where the size of the subset is determined by a promotion probability. Elements are 'promoted' from one layer up to the next based on the promotion probability $p$. An element can be promoted multiple times.

A search on a Skip List starts at the beginning of the top layer Linked List and iterates along the list until an element whose value is greater than the one being searched for is found, at which point the previous element in the Linked List is selected and the iteration continues again in the next lowest Linked List layer. This process is repeated until the element is found or a greater valued element is found in the bottom layer Linked List. An insertion into a Skip List consists of a search operation to find the insertion point, and then generating random numbers to determine in how many layers the new element should be promoted to. The element is then inserted into every promoted layer’s Linked List.

Since elements in a Skip List are stored in sorted order, a prefix search can once again be done by performing a range search. A range search involves doing a search to find the first element in the bottom layer Linked List and then iterate until an element greater than the end element is found.

The described implementation of a Skip List requires $\sum_{n=0}^{\infty} np^i = 2n$ expected space usage, when $p = 1/2$, since for every promotion, the same element is stored an extra time. The space requirement can be reduced to just $n$ by only storing
an element in its highest promoted layer. This change doesn’t impact the search process but does make a prefix search more complicated. A range search with this space saving variation will need to keep track of the search path taken through every Linked List layer to find the starting range element and then iterate through elements on all layers until the ending range element is passed. This extra complexity doesn’t change the prefix search expected time complexity from $O(|s| \log n + k)$ where $k$ is the number of results.

### 2.2 External Memory Concepts

When performing an analysis on external memory algorithms the standard Random Access Machine (RAM) model isn’t sufficient to produce accurate results. In the RAM model, every individual operation essentially has the same time cost. When dealing with external memory, a single operation that makes use of external memory is many orders of magnitude more expensive than one that doesn’t. Thus a model that can analyze this extra cost is needed.

The Input/Output (I/O) model is commonly used when analyzing external memory algorithms. The I/O model views a computer as a processor, internal memory, and external memory. Internal memory is limited and external memory is essentially unlimited. Operations done within internal memory aren’t considered and instead complexity is measured by the number of data transfers between external and internal memory, also known as the number of I/Os [5].

Another important consideration when looking at making use of external memory is the performance difference between sequential reads and random access reads. Reading a single large block of data from external memory is faster than reading multiple smaller blocks of data that are stored randomly on external memory. This consideration greatly influences algorithms that make use of external memory often
causing them to read more data items than necessary when an I/O is necessary. Since more data elements than immediately needed are read in, many algorithms try to increase the probability that the extra data read in will also be needed for the current operation.

Many external memory algorithms will define the size of a block on external memory as $B$. The size of $B$ is often determined by the hardware being used. For example, if RAM is considered as external memory and the Central Processing Unit (CPU) cache is the internal memory, then the size of $B$ is limited by the cache line width of the CPU cache. For some external memory algorithms to run optimally the size of $B$ often has to be tuned to the hardware being used. The block size usually has a large impact on the I/O complexity analysis for external memory algorithms.

### 2.2.1 Cache Oblivious Algorithms

One such type of algorithm that helps to make the extra data read in from an I/O useful are cache oblivious algorithms. An example of a cache oblivious technique is making use of a Van Emde Boas index layout when storing a binary search tree inside of an array. Consider first a basic index layout when storing a binary search tree within an array where the root in the first array index, it’s left child in the second, right child in the third, the left child of the left child in the fourth, and so on. This array can be stored as a sequential block of data in external memory. If this array was very large, an external memory algorithm would want to read this array in blocks that can fit into internal memory which would contain only a subset of the total nodes within the binary search tree. The basic index layout described above would cause every child node to be stored in a separate block or partition since a child’s array index would be about twice as large as it’s parent’s index. This means the expected number of I/Os to search this binary search tree would be equal to the height of the tree.
A Van Emde Boas index layout aims to keep child node indices as close to its parent index as possible which increases the likelihood that a parent and child are stored in the same block lowering the required number of I/Os. Figure 2.3 displays an example of how a Van Emde Boas index layout would look relative to a binary tree node structure. An important feature of a cache oblivious algorithm is that it doesn’t matter what size of blocks are read in from external to internal memory or even how many layers of memory there are. A cache oblivious algorithm will make use of smaller block sizes between a Central Processing Unit (CPU) cache and random access memory as well as the larger block sizes between random access memory and a hard disk. This can be seen in Figure 2.3 where there can be multiple parent children node pairs along a single path in the tree whose index will be only one or two indices away from each other, or a few hundred from each other.

A more complicated example of a cache-oblivious algorithm is the Cache-Oblivious String B-tree described in [6]. It supports dynamic insertion, removal, search, and prefix search operations. The structure consists of three pieces where the first piece is a Centroid tree making use of the Van Emde Boas index layout and the other two pieces are packed memory arrays. A Centroid tree is rearrangement of a compressed
trie turning it into a binary tree. An issue with the Centroid tree structure is keeping it balanced throughout dynamic updates. Rebalancing the tree requires rebuilding the sub-tree under the detected unbalanced node which is much more expensive than the balancing techniques of a Treap or Red-black tree.

Another cache oblivious algorithm described in [7] also makes use of a Centroid tree type Trie rearrangement. It describes techniques that can reduce the space usage of the data structure. This savings is mainly done by altering the way the strings are encoded and compressed into the packed memory array part of the data structure that the Centroid tree points to. This structure also has the issue where rebalancing the Centroid tree is a fairly expensive operation.

Another string based cache oblivious algorithm described in [8] supports prefix search operations that require $O(\log_B n + |P|/B)$ I/Os, where $P$ is the query string. The algorithm makes use of the Trie data structure and covers it with other structures known as Giraffe Trees. The Giraffe Trees require that some of data must be replicated, increasing the storage requirements of the algorithm by a constant factor. The algorithm also requires a Trie as input, or it’s Euler tour which means that this structure is not dynamic and requires a rebuild to modify the data stored within it.

### 2.2.2 Partition Based Algorithms

There’s another type of algorithm that makes use of partitioning to enable the use of external memory. Partitioning usually involves splitting an existing data structure into smaller pieces where any individual partition can easily be stored in internal memory. If a partition becomes too large, it can be partitioned again to a more desirable size. The use of partitioning can allow for internal memory only data structures to be used in an external memory context.

One such example of a partition based external memory algorithm is described in [9] which describes a Suffix Tree, or Patricia Trie making use of partitions. For
any node in the trie, it’s children pointers can either point to another node or to a partitioned sub tree. The partitioning scheme used is a bottom up approach. It starts from a leaf and works its way up the tree until a node whose children sub-tree sizes are too big to keep in the same partition are found. In this case, the current node and children sub-trees are all made into separate partitions. This partitioning scheme is vulnerable to having some partitions with very little data contained within them. Adding this partition scheme on to a Suffix Tree doesn’t prevent data updates from continuing to take place.

Another partition based external memory algorithm example is described in [10] which describes how to use suffix arrays in this manner. A Suffix Array involves computing all possible suffixes for an input string. The computed suffixes are then sorted lexicographically which are used to produce an array of indices based on the starting character position of the suffix relative to it’s complete original string. This array of indices is the suffix array and can be used to perform prefix searches on all suffixes of the original input string. The external memory version splits the array into many blocks and a new array of size equal to the number of blocks is created which will store the index representing the lexicographically first element of each block. A search on the external memory version first does a binary search on the newly created array to identify which data block should then be searched. The main advantage of a Suffix Array is it’s space efficiency but it is also a static data structure requiring a rebuild when updates are required. Also, in the case of this thesis, prefix searching on sub-strings isn’t required.

Hashing data structures can also work with external memory by making use of partitioning. Strings stored in a hash can be partitioned based on the hash value computed from an input string [11]. Lookups on the hash are done in two steps, one lookup to identity which partition or block should be searched and a second to lookup to search the specific partition. This external memory variation of hashing
still performs constant time searches and doesn’t support prefix searches.

A common data structure that involves partitioning blocks of data is a B-tree. In a B-tree each node stores multiple elements and internal nodes have a number of children nodes equal to the number of elements within the node plus one. The values stored in an internal node determine what range of values fall into each child node. A B-tree often has an order parameter which represents the maximum number of elements a single node can store. When a node grows larger than the limit, it is split into two nodes each having half the elements and the central element is inserted into the parent of the node.

A B-tree can perform insertion, removal, and search operations in the $O(|s| \log n)$ time complexity, similar to a binary search tree. Also, like a binary search tree, a B-tree can also perform a prefix search by making use of a range scan. Unlike a regular binary search tree however, a B-tree performs much better when using the I/O model. The maximum number of I/Os a B-tree requires is equal to the depth of the tree, which considerably smaller than a regular tree, since each node stores many elements [12].

Variations of the B-tree have been developed over time. One interesting variation is called the B+ tree. In this variation, all elements are stored in leaf nodes with copies existing in internal nodes only for tree navigation purposes. Having all data only in leaf nodes makes removal operations simpler since the data only needs to be removed from the leaf node and no internal nodes unless node merging needs to occur [12]. Also, an ordered list of leaf nodes can be created to speed up range searches by preventing the need to traverse the tree structure.
Chapter 3

Examined Data Structures

This chapter describes the data structures that this thesis has implemented. Motivations for choosing these structures are also discussed. Each data structure has variations to the standard algorithm that have also been explored. These variants are also described.

The data structures in this chapter all fall under one of two main categories. Some of the structures result in search complexities that are based on the number of elements stored within it. The other structures result in search complexities that are based on the length of the string being operated on. Looking at both of these categories can help determine the strengths and weaknesses of each type when different kinds of input strings are used.

3.1 B+ Tree

The first external memory capable data structure that was examined was a B+ Tree. A B+ Tree has had extensive theoretical and practical work. One of first times the idea of it was brought up was in [12]. Google’s BigTable database model is very similar to a B+ tree in how it stores it’s tablet locations [13]. Due to the popularity and effectiveness of a B+ tree, it was chosen to act a baseline measure of performance.
against the subsequent data structures examined.

The design of the implemented external memory B+ tree follows similarly to a RAM based B+ tree. All leaf nodes are the same depth from the root node of the tree. All inserted data elements are stored in leaf nodes, or the bottom layer, of the tree. Internal tree nodes only store routing elements and pointers to the appropriate child node. The difference from a RAM based B+ tree is that these internal node pointers point to a node that has possibly been serialized to disk instead of another in memory node.

This implementation has removed the need to store the pointers to child nodes within internal nodes by assigning an ID to each node that can be derived using the routing element and height of the internal node within the tree. A node ID has the following format ‘V-nodeHeight’, where V is the routing element value stored in the parent node, and ‘nodeHeight’ is the height of the node in the tree where leaf nodes have a height of 1. For example, ‘the-1’ would be the ID of a leaf node that stores strings greater than or equal to the string ‘the’. Every layer of the tree has a node with an ID of the form ‘-nodeHeight’ that represents either the root node, or values that are less than the value of the lowest routing element of it’s parent. This ID format requires that internal nodes in the tree store their height.

One of the configurable parameters in this implementation is the number of elements a node can store before it splits. This is also known as the branching factor, $b$. Allowing $b$ to be configurable makes it easier to find the optimal block size $B$ for the set of hardware being used in experiments.

A common method of storing data within nodes of a B+ tree is as a sorted array. A binary search can be performed on these arrays to elements in $O(|s| \log b)$ time. Each internal node in the tree can have up to $b$ children. This means that to find a child element in this B+ tree takes $O(|s| \log_b n \log b)$ time when using the RAM model, where $n$ is the total number of elements stored within the tree.
As with regular B+ tree implementations, once a leaf node reaches the defined capacity \( b \), it is split into two nodes where each contains half the elements. Unlike a regular B+ Tree implementation, during removal when a node has less than \( b/2 \) elements, the transfer of elements from sibling nodes doesn’t occur. Instead, once a node stores less than \( b/4 \) elements, it is merged with a sibling node. This change reduces the frequency of incurring extra I/Os during remove operations from having to load the data of sibling nodes. Also, the element transfer has a worst case scenario where the same two siblings can end up being continually merged and split on alternating insert and remove operations. The trade-off for these fewer I/Os is possibly having many smaller nodes than fewer larger nodes.

A common variation of a B+ tree is to link leaf nodes to their siblings using a pointer. This variation can speed up range searches by preventing the need to traverse tree branches to find subsequent leaf nodes. This implementation makes use of this variation by having leaf nodes store the partition ID of it’s next sibling.

### 3.1.1 Variations Explored

A few different variations of the B+ implementation described previously are supported. These variations all deal with how the leaf nodes of the B+ tree store their elements.

One of the more common ways to store data in leaf nodes is to simply store an array of sorted elements. This is one of the variations implemented. The ArrayList data structure from the Java programming language, which is an auto-growing array, was used for this variation. The ArrayList data structure may add an extra performance overhead over using a regular array but this is mitigated when the maximum size of the array is known.

Arrays as a storage type offer quick iterating and searching, when using a binary search algorithm on them. Arrays don’t perform well when elements need to be
inserted or removed near the middle because many elements will need to be shifted over. Arrays require linear time when being split or merged, since in either case elements need to be copied.

Another type of leaf node implementation used was Linked Lists storing sorted elements. Linked Lists were selected because where arrays perform poorly with middle insertions, Linked Lists don’t have to shift elements during modifications. Linked Lists also can be split or merged in constant time, since it is only a matter of updating pointers. The trade-off with Linked Lists is that searching for elements is slower since elements must be iterated through rather than directly accessed using an index. The LinkedList implementation from the Java programming language was used for this variation.

The next type of leaf node variation is a Set based data structure rather than a List. More specifically, the TreeSet implementation from the Java programming language was used. The Java TreeSet makes use of the Red-black tree algorithm. The benefit of using a red-black tree as a leaf implementation is that insert/remove/find operations all have a $O(|s| \log n)$ worst case time complexity. The trade-off is that split and merge operations don’t perform well since the tree will often have to rebalance many nodes.

The final type of leaf node variation is a Treap which is also a Set based data structure. With it’s use of randomization to keep itself balanced, a Treap has a $O(|s| \log n)$ expected time complexity for its insert/remove/find operations. In addition however, merge and split operations also run with a $O(|s| \log n)$ expected time complexity.

A Treap implementation doesn’t come with the standard Java API so a custom one was built. In order to have the Treap perform on par with the built in Java TreeSet, custom serialization code has been added to the Treap implementation to match the TreeSet algorithm. The serialization strategy is to only write out element data in sorted order. No other tree node data is written. On deserialization, the
sorted elements are used to construct a perfectly balanced tree. The priority values for a node is generated based on it’s depth in the tree where each layer of the tree gets a priority value that is equally spaced throughout the 4 byte integer space.

Each of the variations described above have their strengths and weaknesses. Chapter 5 will look at how each variation performs through experimentation.

3.2 B-Skip List

The next external memory capable data structure that was examined was a B-Skip List. The algorithm for this data structure was first described in [14]. A B-skip list is essentially a regular skip list that partitions elements into blocks based on when element promotion occurs.

As with regular skip lists, B-skip lists also have a promotion probability $p$. Every time an inserted element is promoted, a new partition is created. This means that the lower the promotion probability is, the larger the expected size of each partition becomes, $|E(Partition)| = 1/p$. This implementation has made the promotion probability a configurable parameter which allows for tuning the block size to the hardware being used.

When low values are used for the promotion probability in a Skip List, each ascending layer has substantially fewer elements than the layer below it. With a regular promotion probability of 1/2, it is fast to do linear scans starting with the topmost, or smallest, list layer and work down to an element. With a promotion probability of 1/1000 however, a linear scan would have to iterate over about 1000 elements per layer. A binary search instead of a linear scan solves this performance problem. In this implementation, partitions in a layer are skipped over by checking the first element of the next partition until the relevant partition is found, then a binary search is performed within the partition to locate the matching element, which is used
to descend to the next layer.

This implementation assigns a String based address to each partition created. This address is used along with a HashMap to allow for $O(|s|)$ time retrieval of any required partition, where $|s|$ is the length of the string address. Every partition address follows the format 'V-partitionHeight', where $V$ is the value of the node that was promoted to create the partition. For example, if a node with the value 'hi' is promoted from the lowest layer, it's address would be 'hi-1'. The root partition, and the first partition for every layer of the skip list, has no value for $V$. In these cases the partition address follows the format '-partitionHeight'.

Each partition needs to keep track of the addresses to the next and previous partitions on the same layer of the skip list. The next partition ID is required to be able to fully iterate over a layer of the skip list. The previous partition ID is needed when an element which had been promoted is removed from the structure. In this case, the elements in the partition created by the removed node are merged into either the previous or next partition on the layer. This partition merging has no effect on the expected partition size since the probability to remove a promoted element is the same as the probability of inserting one.

A common space optimization to skip lists where a promoted node is only stored in the highest layer was done in this implementation. The space benefit of this optimization would usually about halve the storage space required for a standard RAM based skip list where $p = 1/2$ but since much smaller values for $p$ are used for this external memory implementation, the space savings ends up being very small. The partitioning nature of the B-skip list doesn’t change how the optimization is implemented but it does make it theoretically possible that some partitions may be completely empty.

When promoted elements are not duplicated in every layer, iterating over elements becomes more complicated. The elements that aren’t stored in bottom layers, due to
being promoted, can still be computed on the bottom layer using the ID of the next partition in the layer. The promoted value is contained in the partition ID so when layer partitions change, the value in the ID can be added to the iteration. Using the described technique, the space optimization of not storing promoted elements multiple times has no effect on range search iteration performance.

### 3.2.1 Variations Explored

This implementation of the B-Skip List has variations similar to that of the B+ tree implementation. Normally each layer in a skip list is a linked list and the default choice for storing elements within a B-Skip List partition is a linked list as well. There is nothing preventing the linked lists from being replaced with arrays, red-black trees, or treaps in the same way that the B+ tree can do.

Each of the partition implementation variations have the same trade-offs as described in section 3.1.1. The difference is in how operations are called on B-skip list partitions vs B+ tree leaf nodes. Chapter 5 will look at how each of these variations perform in a B-Skip List using experimentation.

### 3.3 Partitioned Patricia Trie

The final external memory capable data structure that was examined was an External Memory Binary Patricia Trie. This structure is unique from the others in that the run time complexities of it’s operations are based on the length of the strings being dealt with instead of the number of elements [10].

The particular type of Patricia Trie implemented was a binary variation where each node in the trie only has a maximum of two children. One child represents the strings where the next bit in a sequence is a zero and the other, strings where the next bit is a one. The binary variation was chosen so that strings with a large alphabet
size, such as two byte Unicode Transformation Format (UTF) characters, could be chosen without incurring a large space usage overhead. A binary trie implementation requires making use of many bitwise operations making it’s algorithm more complex and difficult to debug than it’s non-binary variation.

The main difference of this external memory capable trie implementation over a regular Patricia Trie is that the trie is partitioned when it grows too large. Similar to the other data structures examined, once a trie reaches a maximum size, it is split into two tries. A leaf of one trie will point to the root of the other. The node that will become the new root in the new partition is determined as follows. Starting at the root, continue traversing to the child node with the largest sub-tree size, find the first child node whose sub-tree size is at most $2m/3$, where $m$ is total size of the current trie partition. This ensures that at least $m/3$ nodes remain in the parent trie. The smallest size that the new child trie can be is $((2m/3) + 1)/2 > m/3$.

The node that matches the above conditions is removed from the original trie and replaced with a pointer node. A pointer node stores only an address to another trie. An address is a string that follows the following format, '[hex encoded complete bytes of the value represented by the root node]˜[bit string of any remaining bits]'. An example address is '4849˜01001'.

Another trie partitioning technique has been described in [15] and [9]. The technique is a bottom up approach used to break a very large trie into several smaller tries all at once. Each leaf starts as it’s own partition and merges with parent partitions if the combined size is below a specified threshold. The partitioning strategy used in this implementation is essentially the same as this since with a max resulting partition size of $2m/3$, a trie would only ever be split into two using the described algorithm.

The maximum size a trie partition can reach has been made a configurable parameter. Similar to the other data structures examined, this will allow for tuning the
block size to the hardware being used.

As new partitions are created when trie partitions get too large, partitions also need to be merged when a removal operation makes a partition too small. This implementation performs partition merges when the number of nodes in a partition drops below $maxPartitionSize/6$. When this occurs, the partition is merged with the smaller of either: the parent trie partition or the last created child partition from this partition. Each partition stores the address of the last created child partition to help facilitate this. Merging the two trie partitions results in the merged partition size being bounded by, $M/3 \leq m \leq 7M/6$, where $M$ is the configured maximum trie partition size. If the merged partition is larger than the maximum defined size, it will split again.

A Binary Patricia Trie is not a data structure that comes with the standard Java API. As such, custom serialization has been created to compete with the serialization of standard API structures. Only element data and tree shape is serialized. This is done with a pre-order traversal of the tree. A byte using bits as boolean flags indicating whether left and right children exist is serialized, along with the number of bits stored in a node and the bits themselves. This serialization algorithm only serializes node data that can't be regenerated from reconstructing the trie.

Prefix searching is done in this external memory capable implementation similar to how it's done in a regular Patricia Trie. The sub-tree rooted at the node representing the desired prefix contains all stored prefixes. To maintain lexicographical ordering on returned results, a Stack data structure is required. As an in-order traversal is done, eventually pointer nodes are encountered, at this point the search progress of the current trie partition is placed onto the Stack and the trie represented by the pointer node is iterated over. This process repeats until the trie partition completes it's iteration as well as all trie iterations stored in the Stack.
3.3.1 Variations Explored

Since the Partitioned Patricia Trie is based around time complexities being determined by the length of strings being stored, the partition implementations shown in the previous sections cannot be reused. The only partition implementation supported by this data structure is a Binary Patricia Trie.

A variation that was created for the Partitioned Patricia Trie is based around how partition splitting is done. A configurable parameter was added that allows for specifying the minimum node depth that a pointer node can be created at within a partition. The basis for this variation is reduce the number of I/Os required to search the trie in exchange for having smaller child partition sizes after a split. Using the regular split algorithm described in Section 3.3, the worst case number of I/Os required to find a node is $O(h)$, where $h$ is the height of all trie partitions combined. With the variation described in this sub-section, the worst case number of I/Os is reduced to $O(h/d)$, where $d$ is the configured minimum node depth during splitting.

The worst case resulting tree size after a split given a minimum split depth, $d$, can be calculated. The worst case tree structure would be a fully balanced trie since it has the smallest maximum root to leaf depth. In this case, the resulting split size would be $m/2^d$, where $m$ is the maximum node size. To handle this, the threshold size for merging trie partitions during a remove operation needs to be adjusted from $m/6$ to $m/2^{d+1}$. Chapter 5 will look at how this variation performs using experimentation.
Chapter 4

Shared Implementation Details

This chapter looks at implementation details that are constant across all examined data structures and their variations. By keeping these details consistent across implementations, the number of external factors that could affect collected performance results is reduced.

There are four main pieces that are shared across all implementations: choice of programming language, how serialization of partitions is done, how partitions are cached out to external memory, and how compression is used. The following sections will look at how each of these pieces were done and why.

4.1 Programming Language

In order to create programs or implementations, a choice of programming language needs to be made. There have been multiple references of data structures implemented in Java throughout this thesis. The choice of programming language for all implementations described in this thesis is indeed Java. This choice has some consequences that wouldn’t arise in a lower level language like C or C++. This section looks at some of the nuances of Java and how they could affect any collected performance results.
The primary reason why Java was chosen was because of its large standard Application Programming Interface (API). Java comes with a host of optimized data structures already implemented. Many are used with the external memory implementations created for this thesis such as: LinkedHashMap, ArrayList, LinkedList, and TreeSet. Being able to make use of these structures provides a good performance baseline for the custom implemented RAM based data structures implemented manually such as: a Treap, and Binary Patricia Tree. It is also easy to extend Java’s standard API with code developed by others using Maven to download Java Archives (JARs) and their dependencies. This thesis has made use of a custom compression algorithm by acquiring the code via Maven. Java is also cross-platform, which is a nice bonus.

Java only allows for primitive types and pointers/references to exist on a program’s stack. An object always exists in heap space and a pointer on the stack can point to it. Also, all object arrays are arrays of pointers. This makes it very tough to take advantage of data locality to achieve more CPU cache hits when using arrays. The only way to have data locality for object arrays in Java is to serialize the objects into byte arrays. Due to the variable length of string data however, even a language that can store actual objects within arrays would have a difficult time achieving data locality without over allocating memory in case of large strings.

Something to keep in mind when performance testing in Java is the relatively large in-memory footprints of objects. In a 64-bit architecture, an object instance requires 16 bytes without including any member variables or the pointer pointing to it. An empty array has an overhead of 32 bytes. String objects, which this thesis makes use of frequently, has an overhead of 64 bytes and each character requires 2 bytes in Java regardless of character encoding. Despite the size of these objects, all implementations are affected by them equally. None of the implementations in this thesis are advantaged by any of this as long as they’re designed properly.
A feature in Java that can cause aberrations with performance tests is its automatic garbage collection functionality. Depending on the order that performance tests run and what they do, one test may complete without the garbage collection triggering while another may be slowed down by the collection. This effect can be mitigated however, as Java supports being able to call the garbage collection process manually. At the start of a test the collection process can be called manually to ensure a similar starting memory space between tests.

### 4.2 Partition Serialization

Whenever a program needs to move data from RAM to a disk or network, some sort of serialization function usually needs to run over the data. Serialization will remove pointers contained within the data and ensure that all data lies within a single continuous block. There are numerous formats that data can be serialized into. There are several text based formats such as: Extensible Markup Language (XML), JavaScript Object Notation (JSON), and YAML. Text based formats are human readable but require more space usage and have difficulty storing binary data.

There are also several binary based formats such as: Apache Avro, Protocol Buffers (protobuf), and Java Serialization/Externalization. These formats can store all forms of data and require less space but aren’t human readable.

Since serialization is kept constant across all data structure implementations and variations, serialization performance is not a large factor for determining what format to choose. Instead capability robustness and ease of use are the most important factors for this thesis. With this in mind, the serialization format chosen was the built-in Java Serialization/Externalization framework. The Java serialization framework is easy to use when programming in Java and when combined with it’s externalization framework, can handle any type of serialization edge case.
Care has to be taken when performance testing code that compares standard Java data structures to custom built ones. Many of the standard data structures in the standard Java API come with custom serialization code making it much faster than default Java serialization. A good example of this is the Java TreeSet. Instead of serializing the nodes in it’s red-black tree, it only writes out its element data in sorted order. On deserialization, it does a bulk insert of the sorted elements. This technique greatly reduces the size of the serialized set which in turn reduces the time needed to read and write to disk.

4.3 Caching to Disk

With serialization decided upon, the next step is to take care of how often serialization needs to be called. The need for serialization can be reduced by making use of caching. Data already stored within a cache can be used immediately instead of having to deserialize from external memory.

All the implementations discussed in chapter 3 have defined an address scheme for labelling partitions that the algorithm creates. The cache implemented in this thesis caches data in and out from external memory based on entire partitions. When a partition is cached out to external memory, it is serialized into a file that has a name equal to the MD5 hash of the address of the partition. The hashing of the address is done to ensure that all characters used in the file name are file system safe. When a partition is accessed via it’s address, if it is already stored within the cache, the partition can just be instantly accessed. If the partition isn’t already in the cache, it is deserialized from it’s address file and stored within the cache.

As more partitions are loaded into the cache, the cache will eventually grow too large. When this happens, the least recently used partition will be flushed to disk. If the partition being flushed, has already been stored onto disk before and hasn’t
changed since then, then it doesn’t have to be written to disk again. If however, the partition has been changed, or is dirty, then it will be serialized to disk.

The LinkedHashMap data structure in the Java programming language allows for constant time lookup/modification operations and keeps track of the order that the elements have been inserted. It is essentially a combination of a Hash Map and a Linked List data structure. The LinkedHashMap has been used to implement the cache.

The size limit of the cache is a configurable parameter defining the maximum number of bytes that is allowed within the cache. Having the limit be based on the number of bytes consumed rather than number of partitions or elements means that the more space efficient a data structure is, the more elements can be in the cache at one time. Using space usage over number of elements as the limit is also much more difficult to keep track of and maintain.

Every serializable class defined in these implementations have been designed to know how large their memory footprint is. Their footprint size is calculated under the assumption that a 64-bit hardware architecture is being used. This means that pointers require eight bytes instead of four. Due to the high use of pointers or references in the Java programming language, the footprint size adds up. Every object in Java has a minimum memory footprint of 16 bytes. An empty String object in Java takes up 64 bytes and each stored character takes 2 bytes regardless of selected character encoding. These facts have been programmed into all implemented serializable classes providing accurate memory footprint sizes to determine the current size of the cache.
4.4 Compression

Another factor frequently considered when writing data to disk or network is compression. Compression allows a disk to store more data and a network to reduce bandwidth usage at the cost of the time taken to run the compression algorithm.

Compression efficiency is strongly correlated to the type of data being compressed. Binary data with less data redundancy will compress less efficiently than text based data which has much more redundancy. It is not uncommon to see text data be compressed to one fifth it’s original size.

As with serialization, there are many different ways to compress data. Some common compression algorithms are: DEFLATE, GZIP, and Snappy. Each of these are supported in the Java programming language. DEFLATE and GZIP is commonly used to compress files on standard operating systems such as Windows and Linux. Snappy compression doesn’t save as much space as other algorithms but has a fast algorithm run time [16]. Snappy compression is often used in computing clusters.

Over the years, CPU and RAM speeds have risen substantially while non Solid State Drive (SSD) disks have remained much the same. This growing difference in RAM vs disk write speeds has made it possible that compressing data before writing to disk and decompressing it when reading from disk can be faster than having no compression enabled. When this is the case, the time to run the compress and decompress algorithms is shorter than the time to write the extra data of an uncompressed file to disk.

Part of the evaluation in chapter 3 will include a comparison between compressing and not compressing data before writing to disk. The comparison will determine how the compression on the data structure’s cache will be configured for the remaining tests.
Chapter 5

Performance Evaluation

In this chapter, the data structures described thus far are put through a variety of performance tests. The results of these tests are then used to identify the strengths and weakness of each implementation and their variations.

Before discussing any results, details of the testing system and its environment are described. Next, the testing procedure and collected metrics are examined. Each data structure is then analyzed on an individual basis and compared against its variations. Finally, all three data structures are compared against each other and the findings discussed.

5.1 System Specifications

In order to understand any acquired performance metrics, the system that the results are collected from needs to be understood first. The test system has an Advanced Micro Devices (AMD) FX-8350 Eight-core processor. Each core runs at 4 gigahertz. The system has 16 gigabytes of Double data rate type three (DDR3) RAM with a clock speed of 1600 megahertz. There is a single Western Digital, model WD2002FAEX, 2 terabyte hard drive installed in the testing system. This drive is a non solid state drive (SSD) running with the standard 7200 revolutions per minute (RPM). Use of
this drive means that RAM speed is several orders of magnitude faster than the disk drive.

On the software side, the testing system is running Windows 8.1 64-bit as it’s operating system (OS). All tests were run with Java 1.7.0 update 21 64-bit. Use of a 64-bit OS and Java install means that all pointers can take 8 bytes to store. This details affects how many blocks, or partitions, can fit into memory at one time. Maven version 3.0.4 was used to acquire any external Java libraries that were involved in these tests, such as the Snappy compression streams. All performance tests were run as JUnit 4 test cases.

5.2 Test Suite and Collected Metrics

This section describes in detail the tests that each data structure and variation has been run against as well as the metrics that were collected in each run.

Every test iteration includes all of the following: 1000000 elements inserted into the structure, 1600000 searches for elements within the structure, 800000 prefix searches, and 750000 element deletions. Every implementation configuration tested is run three times and the average values across the runs are examined. During each phase, 8 times are recorded equally spaced from each other based on number of operations. For example, the search phase records times at the following operation counts: 200000, 400000, 600000, ... , 1600000. Before the start of each phase, the Java garbage collector is called so that leftover garbage from previous operations doesn’t hinder future performance.

The number of operations assigned to each phase may appear too small to effectively test data structures caching to external memory. This is not the case because every implementation being tested has a maximum in-memory cache size of 50
megabytes. An empty String in Java requires 64 bytes in memory and each implementation has to store 1000000 elements, which when multiplied together requires 64 megabytes of memory already. In actuality, the input strings are not empty and more than just a string is required to store an element into each data structure.

Care has been taken to ensure that after the insertion phase, the data structure always has a size of exactly 1000000 and that no duplicate insertions were attempted. Likewise, during the deletion phase, 750000 elements are guaranteed to be removed from the data structure. At the end of an insertion phase, disk based metrics are recorded that include: disk usage size, uncompressed size in memory, memory to disk compression ratio, partition count, serialization time, and disk write time.

There are three different types of string inputs that each implementation has been tested against. These are: sequential strings, random strings, and word based strings. Sequential strings offer a best case view on performance since operations will often require blocks that are already loaded into memory. Sequential string cases can arise in practice if input string data has already been sorted beforehand. Random strings offer a worst case view on performance since operations have the lowest chance for the required block already being loaded into memory. Word based strings involve using words from books as input and offer a more balanced view on performance as fully formed sentences often repeat high frequency words.

The sequential strings used in performance tests are a fixed length of twelve characters where each character is a number. A generated input string the operation number pre-padded with zeroes. For example, the first few input strings would be: 000000000000, 000000000001, 000000000002, and so on. This format of input is used for the insertion, search, and removal operations. Prefix search input strings vary only in the length of input string produced being ten characters instead of twelve. The shorter input strings guarantee that exactly 100 values in the data structure will match any given prefix.
The random strings used in performance tests have random sizes from three to twelve characters. Each character is a randomly generated digit. For example, some random input strings could be: 349, 51262532, or 690127590125. As every tested implementation is performance tested three times and the results averaged, each of the three iterations have a separate but fixed random number generator seed. This means that every implementation will be tested against the exact same random strings. Before the insertion phase, a list of 1000000 unique elements is generated to ensure that no duplicate insertion is ever attempted. During search operations, there’s a fifty percent chance that a random string will be used as input and a fifty percent chance that a random string from the insertion list will be used as input. This ensures that at least some search strings are already stored in the data structure. During prefix search operations, there’s an eighty percent chance that a random string of a length ranging from five to nine characters will be generated and a twenty percent chance that random string of length six or more will be pulled from the insertion list. This causes two element matches to be returned from prefix searches on average. Before the remove operations, a list of random unique indices pointing to the insertion list are generated to ensure that every remove operation removes an element.

The word based strings used as insertion input are read in from word lists for various languages including: English, French, German, Italian, Japanese, Arabic, and Russian. These word lists were acquired from the Linux library aspell and the Linux '/usr/share/dict/words' file. Similarly to the random input case, a list of 1000000 unique words is generated before insertion to ensure an exact data structure size. It should be noted that these word lists are in sorted order making the insertion and deletion phases not very interesting since it closely matches the sequential case. The source of input words for the search and prefix search phases come from books acquired from Project Gutenberg. The book contents are tokenized on spaces and the resulting words become the input. For prefix searches, words of at least three
<table>
<thead>
<tr>
<th>Structure</th>
<th>Full(MB)</th>
<th>Snappy(MB)</th>
<th>Snappy(%)</th>
<th>GZIP(MB)</th>
<th>GZIP(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTree-Seq-15000</td>
<td>15.08</td>
<td>5.07</td>
<td>33.6</td>
<td>2.37</td>
<td>15.7</td>
</tr>
<tr>
<td>BTree-Seq-25000</td>
<td>15.05</td>
<td>5.03</td>
<td>33.4</td>
<td>2.33</td>
<td>15.5</td>
</tr>
<tr>
<td>PatTrie-Seq-6000</td>
<td>6.09</td>
<td>0.54</td>
<td>8.87</td>
<td>0.26</td>
<td>4.27</td>
</tr>
<tr>
<td>PatTrie-Seq-12000</td>
<td>6.06</td>
<td>0.45</td>
<td>7.43</td>
<td>0.18</td>
<td>2.97</td>
</tr>
<tr>
<td>BTree-Rand-70</td>
<td>23.97</td>
<td>16.9</td>
<td>70.5</td>
<td>12.98</td>
<td>54.2</td>
</tr>
<tr>
<td>BTree-Rand-150</td>
<td>17.28</td>
<td>11.66</td>
<td>67.5</td>
<td>8.32</td>
<td>48.1</td>
</tr>
</tbody>
</table>

characters in length have their last character cut off and the result is used as the prefix. This ends up returning 200 prefix matches on average.

## 5.3 Compression Comparison

Before looking at each data structure in depth, the effect of compression on space usage and performance should be analyzed. Knowing the effect of compression and the optimal compression strategy reduces the number of tests that each data structure needs to be put through.

Three compression strategies were programmed into the implementation which include: GZIP compression, Snappy compression, and no compression. These three strategies have been tested against an element count based performance data structure like the external memory B+ Tree as well as a string length based performance data structure like the external memory patricia trie. Also, the compression strategies are compared while using large partition sizes with sequential input data and small partition sizes with random data. The performance of each operation is examined to ensure that the effect of compression is roughly uniform across them all.

Table 5.1 shows the space usage of each compression strategy for every implementation configuration involved in the compression tests. Two different partition sizes
were selected for each input type. It can be seen in input type and implementation that a larger partition size improves the compression ratio. Also, the table shows that the GZIP compression algorithm always produces better compression ratios than the Snappy algorithm.

When looking specifically at the B+ Tree rows for sequential input data in Table 5.1, the Snappy compression reduces the data size to about one third of its original. The GZIP compression compresses about twice as well as Snappy. At the large partition sizes used, increasing the partition size from 15000 to 25000 doesn’t have much effect on improving the compression ratio.

In the table rows showing sequential input being used on an external memory Patricia trie, the compression ratios for Snappy and GZIP algorithms are much better, 8.9% and 4.3% respectively, than in the B+ Tree case. The effect of increased partition size of compression efficiency has a negligible effect with the Snappy algorithm but the GZIP algorithm does see a noticeable improvement from 7.4% to 3%.

In the rows of random input data being used on the external memory B+ Tree, the compression ratios for Snappy and GZIP are much less efficient, at 70.5% and 54.2% respectively. One reason for this is because the random input strings aren’t pre-padded with zeroes. Another reason is that the small partition sizes means that there are more partitions overall. This causes the compression overhead data to be repeated more often. It can be seen that the compression efficiency substantially improves as the partition size is doubled.

Based on the space usage data examined thus far, it appears that the GZIP algorithm is the one to use for all tests due it always compressing better. Before this can be concluded, the impact of these compression algorithms on run time needs to be looked at.

Figure 5.1 shows how each compression strategy impacts the time to perform insertion operations. Sequential input insertions have been graphed for both the B+
Figure 5.1: Insertion Comparison with Compression
Tree and Patricia trie. Random input for the B+ Tree has also been graphed. In all cases, having no compression produces the faster run times.

In the sequential input with a B+ Tree case, the GZIP algorithm performs almost twice the amount of time as compared to using no compression. The Snappy algorithm performs slightly slower than no compression being about 10% slower.

When doing sequential insertions on the Patricia Trie, all three compression strategies perform at about the same speed. In this case, the compression efficiencies are able to completely cancel out the time required to run the compression algorithm.

In the case of random input being inserted into a B+ Tree, the GZIP algorithm takes about 30% longer to run than no compression while the Snappy algorithm takes about 10% longer to run. The worse compression efficiencies don’t reduce the hard disk usage time enough to cancel out the time taken to run the compression algorithms.

Figure 5.2 shows the same implementations as shown from Figure 5.1 but being compared with search operations instead. The two sequential graphs follow the same performance for their search operations as did their matching insertion graphs.

In the random input with a B+ Tree graph, the GZIP algorithm performs twice as slow as no compression. This performance ratio is worse than random insertion ratio. The performance difference between Snappy and no compression is smaller for search operations than insertion operations, being about 10% slower.

Figure 5.3 once again shows the same implementations as shown from the previous two figures but with prefix search operations being compared instead. A notable difference here is that the GZIP algorithm takes three times as long to perform sequential prefix searches in the B+ tree than having no compression. The gap between the Snappy algorithm and uncompressed has also widened to about 50% slower. In the sequential prefix search on the Patricia Trie, the GZIP algorithm has also started
Figure 5.2: Search Comparison with Compression
Figure 5.3: Prefix Search Comparison with Compression
to become slower than uncompressed and Snappy. These larger gaps between compression strategies are because prefix searches require more partitions to be loaded to retrieve all the matches, which increases the time spent compressing and uncompressing.

The prefix searches with random input on a B+ tree performs the same as regular searches. This lack of gap widening in the random cases is because the random prefix search only return about two matches on average.

Lastly, Figure 5.4 shows the implementations having the different compression strategies compared during remove operations. During this operation, the Snappy algorithm performance mostly matches the uncompressed performance for both the B+ tree and Patricia Trie during both sequential and random inputs. The performance difference between GZIP and uncompressed strategies in the B+ tree cases also shrank down to GZIP taking about 25% longer.

Now that all the performance data has been presented, it can be seen that even though the GZIP algorithm always compresses the best, it also takes more than twice as long on prefix searches. It also performs significantly slower, at least 50% more run time, on every other operation in the B+ tree cases. The Snappy algorithm on the other hand, tracks the performance of having no compression very closely with it’s worst case being about 50% slower during sequential prefix searches.

The extra storage capacity from the space reduction offered by the Snappy algorithm is worth the minor hit to performance. The remainder of tests in this chapter are all configured to use the Snappy compression algorithm.

5.4 B+ Tree Performance

In chapter 3, different variations of an external memory B+ Tree were described. These variations were about how the data within a node should be stored. The node
Figure 5.4: Removal Comparison with Compression
storage variations listed were: storing as an array, storing as a linked list, storing as a red-black tree, and storing as a Treap. This section examines what effect each node implementation has on performance and identifies the strongest cases that are then compared against the other external memory data structure implementations in section 5.7.

5.4.1 Partition Size Selection

To accurately compare the node variations, the external memory B+ Tree needs to be configured with a branching factor that performs well on the test system. The ideal branching factor can also change based on the type of input and operations being performed on the tree. To identify the ideal branching factor performance tests were done using three different values for the branching factor where the middle value provides optimal performance and the edge values show performance degradation. Optimal performance was based on the speed of insertion and then prefix searching while using the array node variation. This process was done for each of the testing input types previously described.

Figure 5.5 shows the performance of the B+ tree when given sequential string input for three different branching factor sizes: 5000, 15000, and 25000. In the insertion graph the 15000 factor performs fastest and in the search graph, the branching factor 25000 performs the fastest but 15000 performs nearly the same. It can also be seen that effect of changing the branching factor is very low in this case since all three values perform nearly the same. The remaining B+ Tree tests with sequential input make use of the 15000 block size.

Figure 5.6 shows the performance of the B+ tree when given word based input for three different branching factor sizes: 100, 200, 400. No insertion operation comparison is shown here since the inserted words are in sorted order which would only produce results similar to the sequential strings. It can clearly be seen that the
Figure 5.5: B+ Tree Sequential Input Branching Comparison
200 value performs better than both the 100 and 400 values. The remaining B+ Tree tests with word based input make use of the 200 block size.

Figure 5.7 shows the performance of the B+ tree when given random string input for three different branching factor sizes: 40, 70, 150. In this case, the largest block size of 150 performed best during insertion with 70 value being slightly slower. During prefix searches however, the 150 value was the slowest with the 70 value being fastest and near tied with the 40 block size. The 70 block size performs best when both insertion and prefix searching are considered. The remaining B+ Tree tests with random input strings make use of the 70 block size.

5.4.2 Variant Comparison

With optimal block sizes or branching factors determined for each input type, the comparison of the different node types can take place. Each of the four operation types are examined for each of the three input types. Note that insertion and removal will not be analyzed for word based input since the sorted word lists would produce results similar to the sequential input strings.
Figure 5.7: B+ Tree Random Input Branching Comparison
Figure 5.8: B+ Tree Variant Comparison, Insertions
Table 5.2: B+ Tree Variant Space Consumption

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Disk Usage (MB)</th>
<th>Size in RAM (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTree-Seq-ArrayList-15000</td>
<td>5.07</td>
<td>96.06</td>
</tr>
<tr>
<td>BTree-Seq-Treap-15000</td>
<td>5.06</td>
<td>144.06</td>
</tr>
<tr>
<td>BTree-Seq-RBT-15000</td>
<td>5.06</td>
<td>152.07</td>
</tr>
<tr>
<td>BTree-Seq-LinkedList-15000</td>
<td>5.07</td>
<td>128.07</td>
</tr>
<tr>
<td>BTree-Rand-ArrayList-70</td>
<td>16.9</td>
<td>99.15</td>
</tr>
<tr>
<td>BTree-Rand-Treap-70</td>
<td>15.15</td>
<td>146.04</td>
</tr>
<tr>
<td>BTree-Rand-RBT-70</td>
<td>15.73</td>
<td>156.85</td>
</tr>
<tr>
<td>BTree-Rand-LinkedList-70</td>
<td>16.66</td>
<td>131.95</td>
</tr>
</tbody>
</table>

Figure 5.8 shows the insertion performance of each node variation for sequential and random string inputs. The ArrayList nodes perform the fastest with both the sequential and random inputs.

For sequential inputs, the Treap nodes almost match the ArrayList speeds but with random strings the Treap takes almost twice the time. The Treap performs better than the Red-Black Tree in both input cases.

The LinkedList performs extremely slow with the sequential input. This is because the sequential strings cause insertions at the end of the list, which triggers a full linear scan of the list. With random input, the LinkedList actually performs the second fastest.

Seeing the ArrayList nodes overtake the Treap node is surprising. The Treap not having to shift elements to the right like an array does and the $O(\log n)$ structure split time complexity of the Treap should have made it a good candidate for insertion performance. Table 5.2 sheds some light on why the ArrayList performs so well. The table shows how much space on disk and in memory that each node variation takes.

Each node variation takes about the same amount of space when a partition is serialized, compressed, and written to disk. There is a large difference to how much
space in RAM that each variation takes. The ArrayList takes the least space, followed by the LinkedList, Treap, and lastly the Red-Black Tree. This order also matches the performance ranking for random insertion. The smaller the memory footprint of the node partition, then the more partitions can fit into the cache at one time, which increases the chance of a cache hit during a random insertion. The ArrayList’s small memory footprint provides it with a performance boost that outweighs the extra cost of having to perform linear based insertion or split operations in memory.

Figure 5.9 shows the search performance of each node variation for all three input types. The ArrayList nodes perform best with all three input types. The Treap, Red-black tree and ArrayList all perform about the same for sequential string inputs. The LinkedList once again are extremely slow for sequential input.

With random strings, all non-ArrayList nodes perform about the same which was about 20% longer than the ArrayList. With word based strings, the Treap performs slightly faster than the Red-Black tree. The LinkedList performs noticeably slower than everything.

Figure 5.10 shows the prefix search performance of each node variation for all three inputs types. The ArrayList nodes also perform best in this area.

The prefix search graphs look very similar to the search graphs except that the Treap and Red-black tree nodes performed much worse relative to the other node types. This is expected for tree based structures since iterating through the results requires traversing nodes of a tree rather than a simple list. The LinkedList nodes start performing better than the tree type nodes for word based input data. Note that the random string input graph looks almost identical to the random regular search graph. This is because the random prefix search only return two elements on average.

Figure 5.11 shows the deletion performance of each node variation for sequential and random inputs types. In the sequential input case, the ArrayList and LinkedList nodes have essentially switched spots. Deleting sequentially triggers an array’s worst
Figure 5.9: B+ Tree Variant Comparison, Searches
Figure 5.10: B+ Tree Variant Comparison, Prefix Searches
Figure 5.11: B+ Tree Variant Comparison, Removals
case of continually having to delete the first element and shift everything over. The best case is triggered for linked lists where the linear scan can always stop on the first element and just update pointers. The Treap nodes perform a bit slower than the LinkedList nodes and noticeably faster than the Red-Black tree nodes. This is expected because of the Treap’s $O(\log n)$ time complexity for merging.

Deletion performance for random string inputs remains much the same as for all the other operations with random inputs. The smaller the memory footprint, the better the performance.

With all the variations examined for each operation and input type, it is clear to see that the ArrayList nodes perform best in pretty much every case. The weak sequential deletion performance can be solved by removing elements in reverse order which would trigger an array’s best case for deletion. The larger memory footprint of the other node types cause more disk I/O to occur overall which offsets any gains from faster in memory algorithms.

Based on the data presented in this section, Section 5.7 only compares the external memory B+ tree configured with ArrayList nodes against the other external memory data structures.

### 5.5 B-Skip List Performance

In chapter 3, different variations of an external memory B-Skip List were described. These variations also were related to how the data within a node should be stored. The node storage variations listed were the same as for the B+ tree: storing as an array, storing as a linked list, storing as a red-black tree, and storing as a Treap. This section examines what effect each node implementation has on performance and identifies the strongest cases that are then compared against the other external memory data structure implementations in section 5.7.
5.5.1 Partition Size Selection

To accurately compare the node variations, the external memory B-Skip List needs to be configured with a promotion probability, or expected partition size, that performs well on the test system. Similarly to the B+ tree’s branching factor, the ideal promotion probability can also change based on the type of input and operations being performed on the skip list. To identity the ideal promotion probability, performance tests were done using three different values for the promotion probability where the middle value provides optimal performance and the edge values show performance degradation. Optimal performance was based on the speed of insertion and then prefix searching while using the array list variation. This process was done for each of the testing input types previously described.

Since the expected partition size is equal to the inverse of the promotion probability, the block comparison graphs in this section label each line in the graph as the expected partition size for readability purposes instead of the promotion probability.

Figure 5.12 shows the performance of the B-Skip List when given sequential string input for three different promotion probabilities: 1/3000, 1/6000, and 1/12000. In both graphs, the 6000 and 12000 expected partition sizes perform nearly identically. It can also be seen that effect of changing the promotion probability is very low for both insertions and prefix searches since all three values perform nearly the same. The remaining B-Skip List tests with sequential input make use of the 1/6000 promotion probability or 6000 expected partition size.

Figure 5.13 shows the performance of the B-Skip List when given word based input for three different promotion probabilities: 1/25, 1/50, 1/100. No insertion operation comparison is shown here since the inserted words are in sorted order, which would only produce results similar to the sequential strings. It can be seen that the 1/50 value performs better than both the 1/25 and 1/100 values. The remaining B-Skip
Figure 5.12: B-Skip List Sequential Input Promotion Comparison
List tests with word based input make use of the 1/50 block size.

Figure 5.14 shows the performance of the B-Skip List when given random string input for three different promotion probabilities: 1/25, 1/50, 1/100. In this case, the largest expected partition size of 100 performed best during insertion with 50 being slightly slower. During prefix searches however, the 100 value was the slowest with the 50 value being fastest and near tied with the 25 expected partition size. The 1/50 promotion probability performs best when both insertion and prefix searching are considered. The remaining B-Skip List tests with random input strings make use of the 1/50 promotion probability.

5.5.2 Variant Comparison

With optimal partition sizes, or promotion probabilities, determined for each input type, the comparison of the different list types can take place. Each of the four operation types are examined for each of the three input types. Note that insertion and removal will not be analyzed for word based input since the sorted word lists would produce results similar to the sequential input strings.
Figure 5.14: B-Skip List Random Input Promotion Comparison
Figure 5.15: B-Skip List Variant Comparison, Insertions
Figure 5.15 shows the insertion performance of each list variation for sequential and random string inputs. The ArrayList lists perform the fastest with both the sequential and random inputs.

These results very closely match what was seen during the B+ Tree variant insertion tests. This is understandable since the B-Skip List list variants use the same implementations that the B+ Tree node variants did.

For sequential inputs, the Treap lists match the ArrayList speeds but with random strings the Treap takes almost twice the time. The Treap performs better than the Red-Black Tree in both input cases. The LinkedList once again performs extremely slow with the sequential input and with random input, the LinkedList performs the second fastest.

The space usage of the list variations is mostly identical to what was seen with the B+ Tree node variations in Table 5.2. There are some minor differences due to the randomness involved in how the B-Skip List creates new partitions and each partition requires a few bytes of space for overhead. Once again, the variation with the smallest memory footprint is performing the best.

Figure 5.16 shows the search performance of each list variation for all three inputs types. The ArrayList lists perform best with all three input types. The Treap, Red-black tree and ArrayList all perform about the same for sequential string inputs. For random and word based inputs, the variations all perform the same relatively to each other as was recorded during the B+ Tree variant comparisons.

Figure 5.17 shows the prefix search performance of each list variation for all three inputs types. The ArrayList nodes perform best in this area.

Once again the prefix search performance closely follows the results found with the B+ Tree node variants. The tree type variations performance was worse than in the regular search since the tree structure needs to be iterated over rather than just a simple list. The random input graph looks very similar to the regular search graph
Figure 5.16: B-Skip List Variant Comparison, Searches
Figure 5.17: B-Skip List Variant Comparison, Prefix Searches
Figure 5.18: B-Skip List Variant Comparison, Removals

due to the random prefix searches only returning two elements on average.

Figure 5.18 shows the deletion performance of each list variation for sequential and random inputs types. The deletion performance matches the B+ Tree node variation performance where the sequential deletes trigger the ArrayList’s worst case and the random deletions perform based on the size of the memory footprint.

With all the variations examined for each operation and input type, it is clear to see that the ArrayList lists once again perform best in pretty much every case. The
weak sequential deletion performance can be solved by removing element in reverse order which would trigger an array’s best case for deletion. The larger memory footprint of the other node types cause more disk I/O to occur overall which offsets any gains from faster in memory algorithms.

Based on the data presented in this section, Section 5.7 only compares the external memory B-Skip List configured with ArrayList lists against the other external memory data structures.

5.6 Partitioned Trie Performance

In chapter 3, different variations of an external memory Binary Patricia Trie were described. These variations were related to how the splitting of trie partitions should be done. There are only two variations for this structure: splitting the trie into as equal size pieces as possible, and splitting the trie into equal sized pieces but requiring a minimum depth from the root node before splitting. This section examines what effect each split variation has on performance and identifies the strongest cases that are then compared against the other external memory data structure implementations in section 5.7.

5.6.1 Partition Size Selection

To accurately compare the split variations, the external memory Binary Patricia Trie needs to be configured with a maximum trie partition size that performs well on the test system. Similarly to the B+ tree’s branching factor, the ideal partition size can also change based on the type of input and operations being performed on the trie. To identity the ideal partition size, performance tests were done using three different values for the promotion probability where the middle value provides optimal performance and the edge values show performance degradation. Optimal
performance was based on the speed of insertion and then prefix searching while using the equal trie split variation. This process was done for each of the testing input types previously described.

Figure 5.19 shows the performance of the Patricia Trie when given sequential string input for three different maximum partition sizes: 3000, 6000, and 12000. In both operation graphs, the 6000 max size performs best. It can be seen that effect of changing the partition size is very low for prefix searches since all three values
perform nearly the same. For insertion operations, performance starts significantly degrading with the 12000 max partition size. The remaining Patricia Trie tests with sequential input make use of the 6000 block size.

Figure 5.20 shows the performance of the Patricia Trie when given word based input for three different maximum partition sizes: 50, 100, 200. No insertion operation comparison is shown here since the inserted words are in sorted order which would only produce results similar to the sequential strings. Unlike the word based block
tests for the other data structures, the regular search operation performance graph was also included here. This was because the search graph shows the 50 partition size performing best while the prefix search graph shows the 200 partition size performing best. In both graphs, the 100 partition size performs second best.

Smaller partitions mean that single match searches can read smaller parts of the trie off disk to find the result but for multi-match prefix searches, finding all the matches requires loading many small partitions rather than fewer larger partitions. This trade-off causes this inverse performance between regular searches and prefix searches relative to block size.

The remaining Patricia Trie tests with word based input make use of the 100 partition size to achieve average performance for both types of searches.

Figure 5.21 shows the performance of the Patricia Trie when given random string input for three different maximum partition sizes: 40, 80, 160. The 80 size performed best during insertion. During prefix searches all three sizes performed about the same. The remaining Patricia Trie tests with random input strings make use of the 80 maximum partition size.

5.6.2 Variant Comparison

With optimal maximum partition sizes determined for each input type, the comparison of the different trie split variations can take place. Each of the four operation types are examined for each of the three input types. Note that insertion and removal will not be analyzed for word based input since the sorted word lists would produce results similar to the sequential input strings.

Figure 5.22 shows the insertion performance of each node variation for sequential and random string inputs. The equal trie split variation performs best with both input types. It should be noted that during random inputs, both split variations perform almost identically.
Figure 5.21: B-Patricia Trie Random Input Partition Size Comparison

Table 5.3: B-Patricia Trie Variant Space Consumption

<table>
<thead>
<tr>
<th>Trie Type</th>
<th>Disk Usage (MB)</th>
<th>Size in RAM (MB)</th>
<th>Partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq-EvenSplits-6000</td>
<td>0.59</td>
<td>225.75</td>
<td>500</td>
</tr>
<tr>
<td>Seq-5MinDepth-6000</td>
<td>0.88</td>
<td>226.01</td>
<td>1364</td>
</tr>
<tr>
<td>Rand-EvenSplits-80</td>
<td>15.41</td>
<td>223.85</td>
<td>33992</td>
</tr>
<tr>
<td>Rand-5MinDepth-80</td>
<td>15.67</td>
<td>247.15</td>
<td>56993</td>
</tr>
</tbody>
</table>
Figure 5.22: B-Patricia Trie Variant Comparison, Insertions
It may appear strange that there’s a noticeable performance decrease using a minimum node depth with sequential input but no decrease for random input. Table 5.3 shows the space usage on disk, in memory, and the number of partitions created for each split variation and each input type. As expected, enforcing a minimum node split depth from the root creates more partitions with fewer elements stored within. This causes more overhead data for a trie partition to get duplicated increasing the size of the data on disk. The extra data increases the serialization and disk write times. With sequential input, there are more than 2.5 times the number of partitions required for the minimum split depth variation. With random input, the minimum depth variation needs only about 1.7 times more partitions. The effect of increasing the trie overhead is lower for random input than with the sequential input.

The very small on disk storage sizes with the sequential inputs should be noted. These sizes are much smaller than what was seen for the B+ Tree node variation in Table 5.2. This space savings is one of the benefits of using trie data structures. Each node in the trie makes use of the prefix provided by it’s parent node meaning a shared prefix between values in the trie only has to be stored once. It can be seen that the space efficiency of the trie partitions largely diminishes in the random input tries. The smaller partitions create more overhead and there are fewer elements to share prefixes with.

While the on disk storage size of the tries is small, the in-memory footprint is quite large. In fact, it is larger than any of the B+ Tree node implementations. The overhead of a Patricia Trie node is about the same as for a Treap node. Where a Treap needs to store a single node for each stored element, a Patricia Trie needs to use internal nodes to represent the common prefix of two nodes representing actual values. These internal nodes are what creates the large in-memory footprint.

Figure 5.23 shows the search performance of each split variation for all three inputs types. The equal splits remaining significantly faster for sequential input. The 5 node
Figure 5.23: B-Patricia Trie Variant Comparison, Searches
minimum depth variation performs significantly faster with word based searches and the relative performance gap increases for random input.

Figure 5.24 shows the prefix search performance of each split variation for all three inputs types. The sequential and random input graphs are very similar to the regular search equivalents.

The word based prefix search graph shows that the 5 minimum node depth from root variation only performed slightly faster than the equal split variation. The minimum node depth variation has slowed down relative to the equal split variation because the word based prefix searches return about 200 matches on average. The minimum depth variation has many smaller trie partitions which cause prefix iterations to have to hop between more tries than if equal splits were used. Note that the random string input graph looks almost identical to the random regular search graph. This is because the random prefix search only return two elements on average.

Figure 5.25 shows the deletion performance of each split variation for sequential and random inputs types. In both input cases, the equal split variation performs better although in the random case both variations perform almost the same. The equal split variation suddenly performing deletions faster on random input when it was slower on every other operation can be attributed to its higher minimum partition size merge threshold. Where the equal split variation can merge tries when one reaches \( \text{maxPartitionSize}/6 \) elements, the 5 minimum node depth can only merge at \( \text{maxPartitionSize}/2^{5+1} \) elements because of it’s worse guarantee on minimum size for new trie partitions.

With both variations examined for each operation and input type, it is clear to see that the equal split variation is best for sequential input strings. For word and random inputs, the 5 minimum node depth variation performs better in all cases except random deletion. Given that minimum depth only performs slightly worse for random deletion, it is still a better option overall.
Figure 5.24: B-Patricia Trie Variant Comparison, Prefix Searches
Figure 5.25: B-Patricia Trie Variant Comparison, Removals
Based on the data presented in this section, Section 5.7 uses the equal split variation to compare against the other external memory data structures for sequential input. For word based and random input, the 5 minimum node depth split variation is used instead.

5.7 Cross Structure Comparison

The previous sections in this chapter have identified the optimal partition sizes and variations for each data structure and each of the three string input types. The best variations for each data structure were: the external memory B+ Tree with ArrayList nodes, the external memory B-Skip List with ArrayList nodes, the external memory Binary Patricia Trie with an equal partition split strategy for sequential input and 5 minimum node depth from root for word and random inputs. This section compares the performance of these optimal variations for each of the four operations and three input types.

Figure 5.26 shows the insertion performance of each data structure for sequential and random inputs types. For the sequential string input graph, the B+ Tree performs the fastest with the B-Skip List being slightly slower. The Patricia Trie is the slowest taking more than three times the time for the insertion operations.

In the random string input graph, the B+ Tree and B-Skip List perform at about the same speeds with the B-Skip List being slightly faster. The Patricia Trie performs worse here taking about 2.5 times more time for the insertions. Based on these graphs, the on-disk space savings that the Patricia Trie provides isn’t enough to make up for its much larger memory footprint. The B+ Tree and B-Skip List have mostly equivalent disk and memory sizes. The random insertion graph appears to reflect this.

Figure 5.27 shows the search performance of each data structure for all inputs types. In all input type graphs, the B+ Tree performs fastest although the B-Skip
Figure 5.26: Data Structure Comparison, Insertions
Figure 5.27: Data Structure Comparison, Searches
List very closely matches in performance. For sequential and word based strings, the Patricia Trie performs even more poorly that during the insertion cases being more than four times slower than the B+ Tree. The gap between Patricia Trie and B+ Tree shrinks during the random input case to being 1.5 times slower. The relative gap has shrunk but the Patricia Trie still about 50 seconds slower.

Figure 5.28 shows the prefix search performance of each data structure for all inputs types. In the sequential and word based input graphs, the B+ Tree once again performs best trailed very closely behind by the B-Skip List. The relative performance gap between the B+ Tree and Patricia Trie shrank substantially during the sequential input case and grew slightly with word based input. The large block sizes used the in sequential input cases cause the B+ Tree and B-Skip List to have to read in a lot of unnecessary data when a prefix search is spread across more than a single partition. Conversely, in the Patricia Trie’s case, any extra partitions read in from a prefix search will all be matching results. The data structures use smaller partition sizes for word based input so the B+ Tree and B-Skip List perform closely to their regular search graphs. The smaller partition sizes, and the word prefix searches returning 200 matches on average, require the Patricia trie to iterate through many partitions to find all matches thus greatly increasing its run time.

The B+ Tree and B-Skip List perform about the same as their regular search counterparts. The Patricia Trie performs roughly the same relative to the B+ Tree as it did during regular searches.

Figure 5.29 shows the insertion performance of each data structure for sequential and random inputs types. With sequential input, the B+ Tree performs best, followed by the B-Skip List and then Patricia Trie. Despite sequential removals being the ArrayList’s worst case for deletion, the Patricia Trie still performs substantially slower.

During random removals, the B-Skip List performed faster than the B+ Tree.
Figure 5.28: Data Structure Comparison, Prefix Searches
Figure 5.29: Data Structure Comparison, Removals
The Patricia Trie performed almost three times slower. It’s use of a minimum node depth for splitting greatly lowers its minimum partition split result size, which correspondingly lowers the merging size threshold resulting many small partitions during removals.

5.8 Results Summary

Chapter 5 has taken an in-depth look of performance for each of the three external memory data structures discussed in Chapter 3. The B+ Tree using ArrayList nodes performed best for all input types during search and prefix search operations. It also performed best for sequential insertion and removal operations. The B-Skip List performed better than the B+ Tree during random insertion and removal operations.

Throughout the data structure comparison, it was seen that the B+ Tree and B-Skip List performed very closely to each other. When the promotion probability of a Skip List is lowered, and when partitions are created after each promotion, a Skip List is essentially a randomized B+ Tree. With a high promotion probability, linear scans of the lists can be used to find elements, but as the promotion probability is lowered and the lists grow much longer, binary searches become necessary, similar to when a high branching factor is used in a B+ Tree. Based on the results shown in this chapter, either a B+ Tree or B-Skip List could be used to achieve largely the same performance.

One result that should be noted is the significant savings in disk space that can be achieved by using a partitioned Patricia Trie with large partition sizes. An ArrayList node for both the B+ Tree and B-Skip List took about 5 megabytes of space to store sequential data while the Patricia Trie took 0.5 megabytes to store the same data. This is about an order of magnitude in space savings but its effect on disk read/write time wasn’t enough to overcome its slower algorithm run times or large in-memory
footprint. If disk space is a primary concern then the Patricia Trie could still be an option.

After all the tests performed in this chapter, the largest factor affecting performance appears to be the size of the in-memory footprint of a partition. This can be seen in variant comparisons with random input where the ArrayList out-performed in pretty much every operation. The smaller the footprint of a partition, the more of them that can fit into memory at one time resulting in higher rates of cache hits.
Chapter 6

Conclusion

In this work, three data structures designed to work well with strings on external memory while also supporting dynamic updates and prefix searches were analyzed. These structure were: a B+ Tree, a B-Skip List, and a Partitioned Patricia Trie. Each of these structures had a number of variations proposed for their implementations that could affect their performance.

Every implemented data structure was designed to hold certain aspects, required by all external memory data structures, constant in order to isolate performance comparisons down to the data structure level. These aspects were: the programming language used, the serialization library and style used, the disk caching back-end used, and the compression algorithm when flushing to disk used. Java was used for all implementations along with the Java serialization framework where serializable partitions were required to implement their own custom serialization manually to match the quality of serialization algorithms that come with the standard Java API. All implemented external memory data structures shared the same code to flush least recently used partitions out to disk restricting the size of partitions in memory to a common limit. Lastly, all implementations used the Snappy compression algorithm as it was found to provide substantial space savings with minimal performance impact.

All implemented external memory data structure variations were compared
against each other to determine the most performant cases. These comparisons involved analyzing the performance of insertions, searches, prefix searches, and removals for three different types of input strings: sequential strings, word based strings from books, and random strings. Using basic ArrayLists as the partitions for both the B+ Tree and B-Skip List was found to perform best due to it having the smallest in-memory footprint. The ArrayLists had a weakness when removing data sequentially but it’s impact was minimal and could be solved by removing the data in reverse order.

In the Partitioned Patricia Trie’s case, always trying for equal partition splits resulted in optimal performance when using sequential string data. For word based and random strings, requiring a minimum node depth from the root being at least five performed best. Requiring a minimum depth sacrificed space efficiently for better guarantees on the required number of I/Os needed per operation.

The best variations of each external memory data structure were compared against each other on each of the four operations and three input types. The B+ Tree and B-Skip List were found to perform similarly, but the B+ Tree was slightly faster in all cases except insertions and removal with random input. The partitioned Patricia Trie performed far worse than the others but was also almost an order of magnitude more space efficient on disk when large partition sizes were used.

The most important factor affecting performance of the data structures tested against was in-memory footprint size. The disk read and write costs were so much larger than anything else that whatever implementation could fit the most partitions into memory at one time performed best.
6.1 Directions for Future Work

This work took steps to perform a comprehensive performance evaluation of the data structures. As results were recorded, it started to become clear that certain factors affect external memory performance more than others. The following are some ideas that could be used on the implementations to possibly enhance performance.

It was shown in chapter 5 that in-memory footprint size greatly affected performance when using random input. From the space usage tables, it could also be seen that the difference between the in-memory footprint size and on-disk size was quite large. The size of this difference varied greatly between the data structures and their variations. A possible direction for future work could be to modify the disk caching code to have two stages of caching. The first level cache would be as it is now, fully deserialized partitions, and the second level would be another in-memory cache but storing serialized data rather than deserialized.

This would greatly increase the number of partitions that can fit into memory making the limit based on serialized size rather than in-memory size. This would help the patricia trie, treap, and red-black tree implementations more than the others since they have larger differences between their serialized and in-memory sizes. This cache change may cause one of the other node variations to outperform the ArrayList nodes since the ArrayList would receive the least effect from this. This change may also just take memory space from the deserialized data causing excessive amounts of deserialization algorithms to run.

Since the in-memory footprint was such a significant factor in performance, shrinking that footprint as much as possible can affect random input results significantly. There were a lot of tree type structures used throughout such as the: Treap, Red-black tree, and Binary Patricia Trie. All of these need node objects which require pointers to their children and often the parent. Removing the need for these pointers
can trim down the size of a node substantially. Storing a tree within an array can remove the need for pointers since children and parent indices can be derived from the current index. Using an array for a tree can make non-leaf updates much more expensive though since child elements often have to be rewritten into other indices. Also, if the tree isn’t balanced, the required array size can become impractically large.
List of References


