View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Texas A&amp;M Repository

ANALYSIS, MODELING, AND ALGORITHMS FOR SCALABLE WEB CRAWLING

A Dissertation

by

SARKER TANZIR AHMED

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee, Dmitri Loguinov
Committee Members, Riccardo Bettati
James Caverlee
A. L. Narasimha Reddy
Head of Department, Dilma Da Silva

August 2016

Major Subject: Computer Science

Copyright 2016 Sarker Tanzir Ahmed

ABSTRACT

This dissertation presents a modeling framework for the intermediate data generated by external-memory sorting algorithms (e.g., merge sort, bucket sort, hash sort, replacement selection) that are well-known, yet without accurate models of produced data volume. The motivation comes from the IRLbot crawl experience in June 2007, where a collection of scalable and high-performance external sorting methods are used to handle such problems as URL uniqueness checking, real-time frontier ranking, budget allocation, spam avoidance, all being monumental tasks, especially when limited to the resources of a single-machine. We discuss this crawl experience in detail, use novel algorithms to collect data from the crawl image, and then advance to a broader problem – sorting arbitrarily large-scale data using limited resources and accurately capturing the required cost (e.g., time and disk usage).

To solve these problems, we present an accurate model of *uniqueness probability* the probability to encounter previous unseen data and use that to analyze the amount of intermediate data generated the above-mentioned sorting methods. We also demonstrate how the intermediate data volume and runtime vary based on the input properties (e.g., frequency distribution), hardware configuration (e.g., main memory size, CPU and disk speed) and the choice of sorting method, and that our proposed models accurately capture such variation.

Furthermore, we propose a novel hash-based method for replacement selection sort and its model in case of duplicate data, where existing literature is limited to random or mostly-unique data. Note that the classic replacement selection method has the ability to increase the length of sorted runs and reduce their number, both directly benefiting the merge step of external sorting and . But because of a priority-

queue-assisted sort operation that is inherently slow, the application of replacement selection was limited. Our hash-based design solves this problem by making the sort phase significantly faster compared to existing methods, making this method a preferred choice.

The presented models also enable exact analysis of Least-Recently-Used (LRU) and Random Replacement caches (i.e., their hit rate) that are used as part of the algorithms presented here. These cache models are more accurate than the ones in existing literature, since the existing ones mostly assume infinite stream of data, while our models work accurately on finite streams (e.g., sampled web graphs, click stream) as well. In addition, we present accurate models for various crawl characteristics of random graphs, which can forecast a number of aspects of crawl experience based on the graph properties (e.g., degree distribution). All these models are presented under a unified umbrella to analyze a set of large-scale information processing algorithms that are streamlined for high performance and scalability.

DEDICATION

To my mother, To my father, To my brother, and To my niece.

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest gratitude to Dr. Dmitri Loguinov for all that he has done for me. He has changed me from a person with no research experience to someone who is ready to take on the challenges of real world research. His commitment to research, attention to details, persistence for high quality, most of all his hardwork have left a deep impression on me, as I intend to reflect these qualities in my own research career. I am also deeply thankful to him for his patience with me and for the countless hours that he spent with me not just to solve the problems at hand, but also for teaching me how to solve those on my own.

I am thankful to the Department of Computer Science for generously supporting me as a Teaching Assistant (TA) for the most part of my study here. The job of TA also gave me the opportunity to meet new students, who estimated highly of me and gave me a great deal of confidence, affecting my research positively. I also consider myself lucky to work in the exciting IRL lab, where Xiaoyong Li, Yi Cui, Zain Shmasi have been more than friends to me. I have learned a great deal from working with them.

Finally, I would like to acknowledge my family – my mother and father who worked so hard to give me and my brother the best possible environment to be good human beings and the opportunity to get best out of us. My brother is the best in the world, who although being almost the same age as me, has always been a fatherly figure to me. He has been by me through thick and thin, and continually pushed me to my goal, even when I faltered. I want to mention that without these three persons, I would never achieve this. Thank you for being my family.

TABLE OF CONTENTS

Page

ABSTRACT . . . ii

DEDICATION . . . iv

ACKNOWLEDGEMENTS . . . v

TABLE OF CONTENTS . . . vi

LIST OF FIGURES . . . x

LIST OF TABLES . . . xiii

1. INTRODUCTION . . . 1

    1.1 Motivation . . . 4

2. UNDERSTANDING LARGE-SCALE WEB CRAWLS . . . 6

    2.1 Introduction . . . 6
    2.2 Understanding Web Crawls . . . 8
        2.2.1 Crawler Operation . . . 9
        2.2.2 Duplicate Elimination . . . 11
        2.2.3 Ranking and Admission Control . . . 11
    2.3 IRLbot . . . 12
        2.3.1 Duplicate Elimination . . . 13
        2.3.2 Ranking . . . 13
        2.3.3 Admission Control . . . 15
        2.3.4 Software and Hardware . . . 16
        2.3.5 Discussion . . . 19
    2.4 Page-Level Analysis . . . 19
        2.4.1 Admitted URLs . . . 21
        2.4.2 Crawled URLs . . . 22
        2.4.3 Downloaded URLs . . . 23
        2.4.4 Links . . . 25
        2.4.5 Summary . . . 26
    2.5 Server-Level Analysis . . . 28
        2.5.1 DNS and Robots . . . 28

2.5.2 Bandwidth 31
2.6 Extrapolating Crawls 32
2.6.1 Stochastic Model 33
2.6.2 Data Extraction 35
2.6.3 URLs 36
2.6.4 Hosts and PLDs 38
2.7 Internet-Wide Coverage 39
2.7.1 Basic Properties 39
2.7.2 Observations 41
2.7.3 TLD Coverage 43
2.7.4 More on Budgets 46
2.8 Related Work 47
2.9 Conclusion 48

3. RANDOMIZED DATA STREAMS 49

3.1 Introduction 49
3.2 Literature Review 50
3.3 Randomized 1D Streams 51
3.3.1 Terminology 52
3.3.2 Stream Residuals 52
3.3.3 A Few Words on Simulations 54
3.3.4 Single Node 55
3.3.5 Uniqueness Probability 60
3.3.6 Set of Unique Nodes 62
3.4 Randomized 2D Streams 63
3.4.1 Terminology and Assumptions 64
3.4.2 Degree of the Seen Set 65
3.4.3 Destination Nodes 69
3.5 Applications 72
3.5.1 Least Recently Used Cache 73
3.5.2 Random Replacement Cache 77
3.5.3 MapReduce 79
3.5.4 Frontier in Graph Traversal 83
3.6 Conclusion 88

4. CLASSICAL EXTERNAL MEMORY ALGORITHMS 89

4.1 Introduction 89
4.1.1 Contributions 90
4.2 Related Work 91
4.2.1 MapReduce Performance 91
4.2.2 Replacement Selection 106

4.3 Random Stream Properties 107
  4.3.1 Terminology and Assumptions 107
  4.3.2 Uniqueness Probability 108
  4.3.3 Experimental Setup 110
4.4 Basic Load-and-Sort 111
  4.4.1 Preliminaries 111
  4.4.2 Merge Sort 112
  4.4.3 Hash Tables 115
4.5 Replacement Selection 118
  4.5.1 Traditional Two Queue Implementation 119
  4.5.2 Disk I/O in Replacement Selection 120
4.6 Discussion 122
  4.6.1 Comparisons 122
  4.6.2 Multi-Core MapReduce 125
  4.6.3 Cluster MapReduce 127
4.7 Conclusion 127

5. ADVANCED EXTERNAL MEMORY ALGORITHMS 129

5.1 Introduction 129
5.2 Related Work 130
  5.2.1 Caching 130
  5.2.2 High-Performance Sorting 134
5.3 Preliminaries 142
  5.3.1 Finite Stream 142
5.4 Incremental Hash Table 143
  5.4.1 Individual Run Analysis 144
  5.4.2 Partial Key Space Functions 145
  5.4.3 Key Density Functions 146
  5.4.4 Miss Rate 149
  5.4.5 Main Results and Convergence 150
  5.4.6 All-Unique Keys 155
  5.4.7 Extension to Caching Techniques 157
5.5 Space Efficient Caching 158
  5.5.1 WATCH 158
  5.5.2 Analysis of WATCH 160
5.6 Bucket Sort 163
  5.6.1 Bucket Sort with Cache 163
  5.6.2 Data Compression under Caching 164
  5.6.3 Homogeneous Split 165
  5.6.4 Heterogeneous Split 167
  5.6.5 Optimal Split 169
5.7 Comparisons 171

5.8 Conclusion 175

6. SUMMARY AND FUTURE WORK 177

6.1 Summary 177
6.2 Future Work 179

REFERENCES 181

LIST OF FIGURES

FIGURE Page

1.1 MapReduce with external-memory deduplication. 2

2.1 Components of modern crawlers. 10

2.2 Common BFS pitfalls. 13

2.3 IRLbot crawl analysis. 20

2.4 DNS and robots.txt requests. 28

2.5 Downloaded HTML pages. 32

2.6 Model verification for $p(t)$ and URL discovery rate $\tilde{p}(z)$ in IRLbot. . . 35

2.7 Host/PLD discovery rate $\tilde{p}(z)$ in IRLbot. 39

2.8 TLD coverage (Google order). 45

2.9 Crawl depth distribution. 47

3.1 Simulation with Zipf $\mathcal{I}$ with $\alpha = 1.2, E[\mathcal{I}] = 10, n = 10$K. 55

3.2 Verification under Zipf $\mathcal{I}$ with $\alpha = 1.2, E[\mathcal{I}] = 10, n = 10$K. 58

3.3 Verification of (3.22) under $E[\mathcal{I}] = 10, n = 10$K. 61

3.4 Verification of (3.27) under $E[\mathcal{I}] = 10, n = 10$K. 63

3.5 Verification of $p(t)$ in BFS crawls with $E[\mathcal{I}] = 10, n = 10$K. 65

3.6 Verification of $E[\phi(S_t)]$ in BFS crawls with $E[\mathcal{I}] = 10, n = 10$K. . . . 66

3.7 Verification with Zipf $\mathcal{I}$ under $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10$K. 68

3.8 Verification of models (3.41) and (3.45) with Zipf $\mathcal{I}$ and $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10$K. 71

3.9 Verification of miss rate model (3.48). 74

3.10 Existing vs proposed LRU miss rate model on finite streams. 76

3.11 Existing vs proposed RND cache miss rate model on finite streams. . 80

3.12 MapReduce with external sort. 81

3.13 Verification of (3.56). 82

3.14 Verification of frontier out-degree (3.60) with Zipf $\mathcal{I}$ and $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10$K. 84

3.15 Verification of the frontier size in BFS with Zipf $\mathcal{I}$ and $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10$K. 86

3.16 Verification of model (3.64) in FRN with Zipf $\mathcal{I}$ and $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10$K. 87

4.1 Verification of (4.3) on real graphs. 110

4.2 Verification of disk I/O (4.8) in merge sort. 113

4.3 Two different hash table data structures. 116

4.4 Disk I/O for sort and merge phases in HT MapReduce. 117

4.5 Number of sorted runs in merge sort and hash table designs. 118

4.6 Verification of Theorem 15. 120

4.7 Comparison of disk I/O various MapReduce designs with respect to merge sort. 122

4.8 Comparison on synthetic Zipf ($\alpha = 1.5$) graphs with $T = 5$M. 124

4.9 Multi-Core hash table MapReduce 126

5.1 System model in IHT. 144

5.2 Verification of (5.7) with $x = 0.5$. 148

5.3 Verification of q_∞ in (5.15). 152

5.4 Verification of (5.13), (5.14), and (5.21). 153

5.5 Verification of sorted runs generated by IHT MapReduce. 154

5.6 Verification of disk I/O by IHT MapReduce. 155

5.7 Verification of q_∞ on rand. replacement. 157

5.8 Tradeoff between speed and r in terms of hit rate (128M lookups, 64K cache size). 159

5.9 Partitioning tree in bucket sort for homogeneous split. 164

5.10 Verification of L_1 (5.33) on real graphs under LRU and CLOCK. . . . 165

5.11 Verification of L (5.34) for homogeneous b on real graphs under LRU and CLOCK. 167

5.12 Verification of Algorithm 2 for heterogeneous b on IRLbot under LRU and CLOCK. 170

5.13 Comparison of disk I/O various MapReduce designs with respect to merge sort. 172

5.14 Duality between MS-HT and RS-IHT pairs. 174

5.15 Comparison on synthetic Zipf graphs with $T = 10$M, $E[D] = 5$. 176

LIST OF TABLES

TABLE Page

2.1 Parsing speed of http://whitehouse.gov using a single core of a 2.6 GHz AMD Opteron 2218. 17

2.2 HTTP status codes of downloaded URLs 24

2.3 Internet coverage of existing crawls 40

2.4 Commercial datasets 43

2.5 Google-ordered top-10 TLD List 44

4.1 Processing speed of different data structures. 125

5.1 Run lengths (normalized by h) of IHT on the IRLbot graph. 154

5.2 Verification of (5.32) on the IRLbot graph. 161

5.3 Sorting and aggregation speed of different methods. 174

1. INTRODUCTION

The growth of the Internet and Internet-facing applications in the wild has been an explosive phenomenon, leading to humongous amount of data generated all-over the globe across different companies and enterprizes. These data are mostly about user-interaction with the online services and thus carry a great deal of value if utilized for business intelligence. Therefore, companies with large customer base are usually interested to process these data in scalable and timely fashion.

The abundance of valuable data has inspired numerous infrastructure and algorithmic techniques to process them. In the past, companies were to purchase expensive hardware resources and commit to those for a longer period of time. But this reality has changed since the inception of cloud computing which enables pay-per-use based strategy, where the customers lease compute resources as they wish and pay accordingly. With hardware availability, along came different revolutionary software paradigms that are suited for handling such data. The most popular among all is Google's MapReduce and its open source implementation Hadoop.

MapReduce is a programming paradigm consisting of two phases - *map* and *reduce* as shown in Fig. 1.1. The former runs a user-provided parsing function over the data and produces a structured stream of key-value records. The latter, which is our focus in the thesis, identifies all duplicate keys in the result and runs another user-provided function to merge the value fields of each discovered key. Due to the scale of the data, deduplication often requires external memory, where the input is split into chunks containing partial solutions. These are later combined to produce the final result. Due to duplicate removal, the stream undergoes shrinkage in both "dedup" boxes of Fig. 1.1, where the outcome depends on the algorithm (e.g., hash table, quicksort),

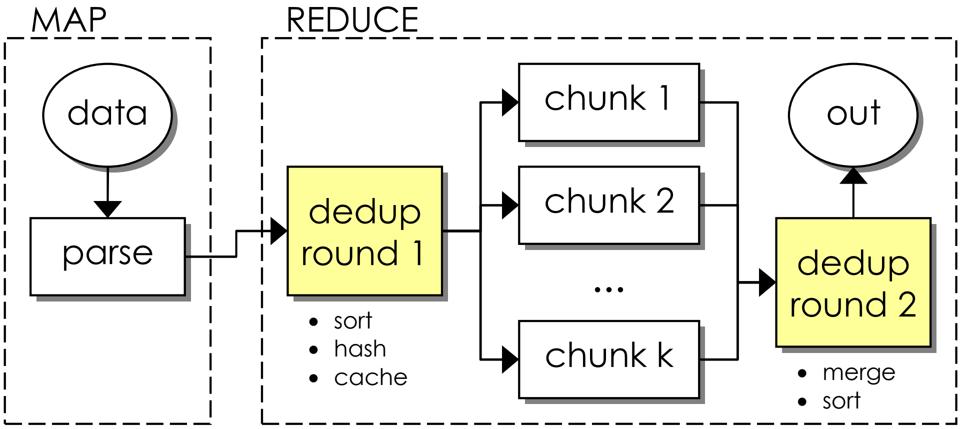


Figure 1.1: MapReduce with external-memory deduplication.

amount of available RAM, and key distribution. Therefore, a MapReduce model can be considered comprehensive only if takes into account these nuances, which we target in this thesis.

MapReduce hides the details of cluster programming and easily scales to a large number of compute nodes available in owned or rented clusters. There have been many other models and paradigms proposed by research community that either extend MapReduce framework to support different missing features in the original MapReduce, or propose fundamentally new frameworks with similar or exceeding capabilities.

Note that the business quires can vary widely based on the desired information outcome (e.g., building an inverted index of a search engine, ranking pages using PageRank, analyzing DNS traffic, investigating user click streams for adware, optimizing database queries). Furthermore, the nature of these workloads (e.g., repetition pattern of the data and correlation among them) also vary significantly from each other, and thus pose varied degree of resource requirements.

For instance, a word count MapReduce job would experience much compression from combining repeated words in its local memory if there is only a small number of distinct words each appearing frequently in the corpus. The amount of intermediate data in this case reduce significantly compared to another scenario where the words are mostly unique and do not repeat as much. Since the intermediate data volume dictates how much data to spill to disk or send over to the correct reducer node, this also in turn controls the runtime of a MapReduce system. Similarly the underlying algorithm also has noticeable impact on the performance. For instance, in the frequent-word case, a hash-based pre-aggregation (i.e., local aggregation before the shuffle phase of MapReduce) delivers a great deal of advantage over a merge-sort design, where data are just sorted first and then sent over to the reduce nodes.

Because of all these different choices in terms of hardware, algorithms and data structures, a new problem emerges in this elastic hardware-software ecosystem. Users are faced with a number of crucial questions – how much resource to acquire for a particular task at hand, given other business constraints (e.g., financial and time line budget)? What type of sorting and aggregation algorithms should be chosen? How many compute nodes will suffice? On top of all, how these choices are affected by the frequency distribution and other properties (e.g., correlation) of the input data? In this dissertation, we address these questions systematically, focusing on what happens inside each compute node. We view the problem as a classical external memory sorting problem and accurately relate its performance with available computing resources (e.g., available main memory) and the workload characteristics. We then extend our analysis to cluster environments as well.

In the last part of this work, we specifically investigate bucket sort, where the key motivation is the $O(n)$ complexity of this sort and its leverage in the presence of space-efficient caching. Since caching can reduce the amount of intermediate data from bucket sort significantly, we examine a number of caching choices (e.g., LRU, CLOCK, random replacement) and present corresponding models. To solve overhead and throughput problem of these caching methods, we devise a novel caching algorithm, we call WATCH, along with its hit-rate model, which has much less space overhead (i.e., 1 byte per key) compared to existing methods, but shows similar and sometimes exceeding hit-rate. We demonstrate through our experiments and the models that bucket sort in conjunction with this newly designed cache shows much better overall performance compared to other external-sorting techniques under a wide range of hardware configurations and workload properties.

1.1 Motivation

Crawling the web itself is an example of large-scale data processing applications, where a generic crawler has to manage huge volume data sets. Here, by a generic crawler, what we mean is the crawler does not access to a oracle for discovering the set of pages to crawl, or apply priority among them to decide crawl order. Rather, the crawler has to start with minimal information about the Internet (e.g., just one seed page), discover all the links or pages to crawl by parsing crawled pages, apply prioritization (e.g., PageRank) and politeness mechanism, spam avoidance, and manage the ever-growing unseen URLs that accumulate. The problem becomes far more challenging when the objective is perform all these on single machine, rather than a thousands-of-machine cluster which is a more common approach these days [27], [92], [101], [124], [128], [142].

This monumental problem was approached by Internet Research Lab (IRL) with success in collecting the largest non-commercial crawl of the web to date. The crawl experiment happened in June 2007 for 41 days of continuous crawling resulting in 8.2B web pages. While the details of the crawl mechanism, uniqueness check of unseen URLs, spam avoidance, real-time ranking, and other aspects of the experiment are available in [126], [87], they do not contain the detailed analysis of the data. We start by analyzing this massive 7-TB dataset, which is a massive web graph with 41B nodes and 394B links. Processing this workload poses unique challenges specially when the platform is a single machine, since there are different algorithms and data structures available (e.g., hash sort, merge sort, replacement selection, bucket sort). In addition, these algorithms behave differently based on the repetition pattern of the workload and the computation task at hand. All these problems motivated the research work documented in this dissertation, where the objective is to choose the best sorting method depending on the task, input characteristics, and the hardware resources provisioned.

2. UNDERSTANDING LARGE-SCALE WEB CRAWLS

2.1 Introduction

Web crawling is not just an important component of major search engines, but also a vital experimental activity that provides indispensable research data for such fields as networking, distributed systems, databases, machine learning, information retrieval, security, and linguistics. For years, research into large-scale web crawling has been led by industry players (e.g., Google, Microsoft, Yahoo) and has remained shrouded in secrecy. While many academic crawlers have been proposed in the literature [5], [17], [24], [25], [29], [28], [30], [57], [65], [67], [74], [96], [114], [122], [142], their scale, Internet coverage, download speed, and ability to deal with spam have not kept up with the evolution of the web.

Part of the problem is the perceived impossibility, and thus lacking attempts, to rival commercial search engines using a research implementation. This notion commonly stems from a belief that to achieve meaningful results web crawlers must be heavily parallelized and even distributed across multiple domains [17], [27], [92], [101], [105], [122], [124], [128]. As a consequence, not much effort has been put into optimizing individual servers, improving their algorithms, or reducing complexity, all under the assumption that arbitrary scalability could be achieved just by acquiring "enough" hardware. Unfortunately, due to the serious financial investment needed for this vision, few academic crawlers have gone beyond small-scale prototypes.

The second problem is that prior crawls are often poorly documented and difficult to interpret. As the field stands today, there exists no standard methodology for examining web crawls and comparing their performance against one another. With each paper providing different, and often very limited, types of information, little can be said about the relative strengths of various crawling techniques or even their web coverage. Setting aside the financial aspect discussed above, this lack of transparency has helped stifle innovation, allowing industry to take a technological and scientific lead in this area.

Finally, the majority of existing web studies have no provisions to handle spam [17], [21], [28], [96], [114], [122], [142]. A popular technique for tackling the massive scale, infinite script-generated traps, and uncertainty about content quality is to select a handful of "good" sites and then crawl them until some maximum number of pages is reached within each host [17], [28]. While perfectly suitable for limited-scale sampling, this approach does not easily generalize to Internet-wide crawling scenarios.

Our philosophy for IRLbot [87] was to challenge the perception that web crawlers should be confined to fixed parts of the Internet and/or consist of many low-performance hosts that could execute arbitrarily complex tasks in real-time (e.g., sorting the frontier by PageRank [29]). Instead, the goal was to develop a crawler that could explore the web at unprecedented scale using low-overhead algorithms and an efficient single-server implementation that would not bottleneck on duplicate detection, URL prioritization, and politeness rate-limiting. While [87] addressed the main IRLbot algorithms and [126] scrutinized its domain ranking, both papers omitted almost all measurement results from our 2007 crawl of 6.3B HTML pages (20% of Google's index at that time).

While working on IRLbot [87], we found plenty of information [16], [19], [132] about the topology of the webgraph (e.g., size of the largest connected component, degree distribution, clustering coefficient), but actual hands-on knowledge from large-scale experiments was much more scarce [105], [122]. This placed a steep learning curve on crawler design, limited opportunities for comparison, and hindered emergence of better approaches.

To overcome these limitations, our contribution in this chapter is to propose a new methodology for understanding web crawls, set forth guidelines for systematically analyzing crawler performance, and provide evidence that high-performance web exploration is possible using a research implementation. Here, we dissect our massive-scale high-performance crawl experiment *IRLbot* [71], [86] on the Internet performed in June-July 2007 for a duration of 41 days. In addition to its scale and performance, IRLbot used real-time prioritization of downloaded pages in an effort to avoid spam. Our additional contribution is to analyze the growth rate of various data structures as crawl size scales up and introduce techniques for assessing web coverage using commercial search engines, neither of which has been attempted before.

2.2 Understanding Web Crawls

Internet-scale web crawling is a more challenging issue than it may appear at first. Recall that the web can be viewed as a directed graph consisting of distributed objects (e.g., pages, documents, images) and links between them (i.e., URLs). The raw size of this structure is often considered infinite due to the virtually unlimited amount of unique pages and host names that can be generated by scripts. Even after pruning spam, infinite branches, and links generated by duplicate content, the useful web is still extremely large. For example, Google reported seeing links to 1T unique pages in 2008 [6] and 30T four years later [123].

The enormous scale of the web and the presence of sophisticated spam alliances on the Internet require crawling techniques that can efficiently eliminate duplicates, rank the frontier, and distribute crawling budgets between various websites, while processing hundreds of tera bytes of discovered data and manipulating graphs with trillions of edges. This section examines these challenges in more detail and analyzes various approaches to handling scale from prior work.

2.2.1 Crawler Operation

As shown in Fig. 2.1, crawler operation can be reduced to a cycle with eight major components. Besides maintaining many concurrent HTTP sessions and parsing HTML, crawlers must eliminate duplicate URLs before attempting to crawl them, rank the frontier (i.e., decide importance of all seen, but not-yet-crawled pages), perform admission control on pending URLs using ranks computed in real-time, execute DNS lookups, enforce `robots.txt` directives of crawled websites, and finally adhere to politeness rate limits (both per-IP and per-host) that prevent crashing of individual servers.

The ability of each component to keep up with the crawl is determined by two factors – size of the underlying data structure and speed at which it must operate. Suppose $\mathcal{D}$ is the number of downloaded pages, q is the fraction of them with error-free (i.e., 200 OK) HTML content, and h is the number of crawling servers. Further assume that the parser produces on average l links per page that are *locally* unique (i.e., within that page). After verification against all previously seen URLs, suppose fraction p of them are *globally* unique.

Armed with these definitions, we can now ballpark the storage and processing demand of each crawling server. Duplicate elimination uses a data structure with $\mathcal{D}qlp/h$ hashes of seen pages, admission control keeps track of all unique pages minus those already crawled, which amounts to $\mathcal{D}q(lp-1)/h$ full URL strings, and frontier ranking operates on webgraphs with $\mathcal{D}ql/h$ edges. The overhead of the other three components is dependent on the number of hosts visited by the crawler, cache expiration delays (DNS and robots), and the desired website concurrency in the politeness

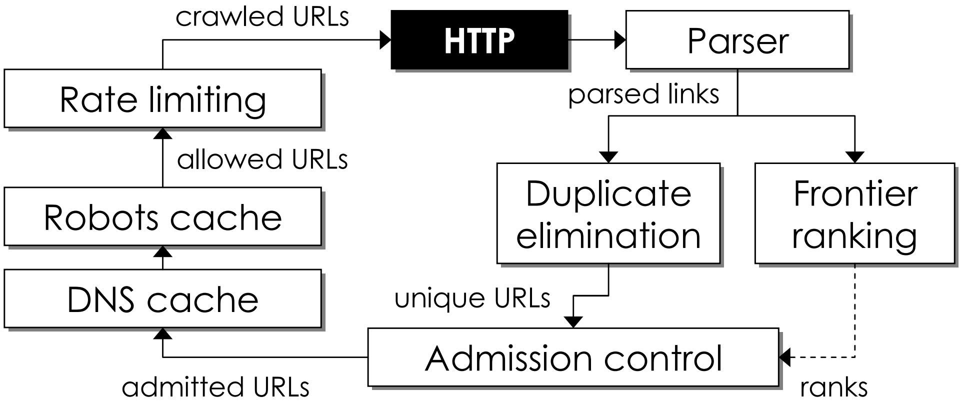

crawled URLs
HTTP
Parser
Rate limiting
parsed links
allowed URLs
Duplicate elimination
Frontier ranking
Robots cache
DNS cache
unique URLs
Admission control
admitted URLs
ranks

Figure 2.1: Components of modern crawlers.

scheduler.[1]

Components closer to the parser in the clockwise direction need to support higher throughput rates. For a target speed of $\mathcal{S}$ crawled pages/sec (pps), both duplicate elimination and frontier ranking must operate at $\mathcal{S}ql/h$ links/s (lps), while admission control must sustain an injection rate $\mathcal{S}qlp/h$ URLs/s and extraction rate $\mathcal{S}/h$, the latter of which is also the speed of the remaining components in the figure. Given $l \approx 50$ and peak rate $\mathcal{S} = 3$K pps from IRLbot experiments, verification of uniqueness and injection of edges into the webgraph must proceed at rates well in excess of 100K/s.

As larger data structures require more processing overhead, crawler design boils down to a tradeoff between four parameters $\{\mathcal{D}/h, \mathcal{S}/h, q, l\}$, where increase in one parameter typically requires reduction in other parameters. However, this also leads to crawls with incompatible results and difficulties in gauging performance of their underlying algorithms. We discuss several examples next.

---

[1] For example, IRLbot began stalling if the number of websites with backlogged (i.e., allowed, but not yet crawled) URLs dropped below 200K.

2.2.2 Duplicate Elimination

Due to the tiny size of the early web, first-generation crawlers [18], [64], [96], [114] either kept all data in RAM or used random disk access to verify URL uniqueness. With 8-ms seek delays and worst-case access locality, these methods in today's Internet would bottleneck around $\mathcal{S} = 125/l \approx 2.5$ pps. Second-generation crawlers [105], [122] replaced disk seeking with batch-mode sorting that periodically scanned the file of previously seen URLs and merged the new ones in. While this approach works well for a few hundred million pages, scaling this technique further requires a significant reduction in $\mathcal{S}$, in some cases pushing the crawler to a virtual standstill.

Subsequent literature, which we call third-generation, universally dismissed single-server designs and assumed that scalability was only achievable horizontally, i.e., by increasing h. This work [27], [92], [101], [124], [128], [142] focused on parallelizing the URL workload across server clusters and P2P networks. However, even with distributed operation, these designs have been limited to crawl size 400K-25M pages and experiments lasting only minutes, with no measurable improvement in the last decade.

Besides trading $\mathcal{S}$ for $\mathcal{D}$ and scaling h, other methods include reduction in q (i.e., download of non-HTML objects) [64], [106], elimination of dynamic links (e.g., forums, blogs, social networks, shopping sites) to reduce l [37], and avoidance of disk-based uniqueness verification altogether by either keeping all data structures in RAM [9], [17], [28] or revisiting the same pages on a regular basis [21], [65], [67], [70].

2.2.3 Ranking and Admission Control

Several papers [5], [10], [24], [25], [29], [30], [57], [74] have proposed that crawlers compute a certain graph-theoretic metric for each page (e.g., PageRank [18], OPIC [2]) and that pending URLs be served in the order of their rank. However, due to the high CPU and I/O cost, most of these efforts remain limited to offline simulations. Among the crawls in the literature with at least 50M pages [17], [28], [64], [105], [122] none have used real-time spam avoidance or global frontier prioritization, most often relying on variations of polite BFS to automatically find good pages [106].

While open-source implementations exist with non-BFS capability [107], they do not generally publish performance results or disclose operational details, which makes their analysis difficult. They also often require substantial resources (e.g., clusters with large h, terabytes of RAM) and significant reduction in l to sustain non-trivial crawls. One notable example is ClueWeb09 [37], which parallelized Apache Nutch [107] using a number of servers in the NSF-Google-IBM cluster[2]. After discarding all dynamic links, the experiment finished 1B pages in 52 days at an average rate of 222 pps; however, little additional detail is available about this dataset, its crawl dynamics, web coverage, or the employed algorithms.

2.3 IRLbot

IRLbot was conceived in 2004 with the goal to surpass the size of Google's advertised index at that time (i.e., 8B pages). This objective took longer than expected, transpiring into 3 years of research and implementation to overcome various obstacles in the way. In 2007, IRLbot came close to this target by attempting to crawl 8.2B URLs, out of which 6.3B were error-free `text/html` pages. If it had not triggered an exploit in a remote database that caused deletion of records, complaints to the university president, and involvement of lawyers, IRLbot would have reached its target 11 days after it was shut down under threat of legal action.

Our main principle with IRLbot [87] was to approach scalability from a new angle, which consisted of never assuming that the frontier and various graphs would fit in

---

[2] Contained a hefty 90K nodes in 2008 [1].

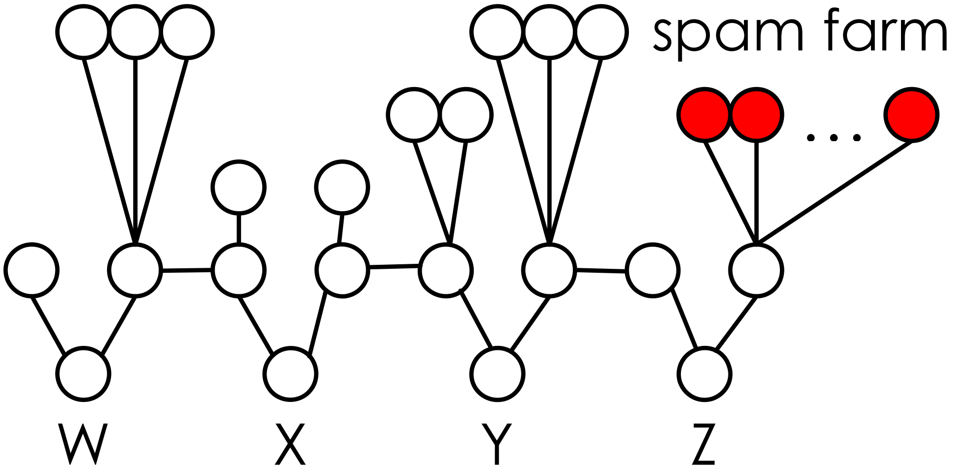


Figure 2.2: Common BFS pitfalls.

RAM, identifying and addressing performance bottlenecks rather than throwing more hardware at the problem, and aiming to discover the inherent limits of a single-server implementation before attempting cluster-based parallelization.

2.3.1 Duplicate Elimination

The first major obstacle was scaling verification of URL uniqueness to billions of pages without sacrificing download speed $\mathcal{S}$, increasing h, or reducing l. Our solution to this problem was a novel MapReduce design based on external bucket sort [87] that not just coped very well during the actual experiment, but also possessed excellent scalability to much larger crawl sizes. Specifically, analysis in [87] showed that with a fixed I/O rate of 100 MB/s and 8 GB of RAM, this technique could sustain crawls of $\mathcal{D} = 8$T pages at $\mathcal{S} = 4$K pps, which was 3-4 orders of magnitude faster than the best crawlers in the literature at that time [105], [122].

2.3.2 Ranking

The second challenge with the early IRLbot involved BFS being stuck in various script-generated traps, typically after the first 1-2B downloaded pages. One pesky example was a script that replied to any GET request with links to 1K random hostnames, all resolving back to the same server through a wildcard DNS domain. Due to their locality in the BFS queue, millions of these URLs were clustered in close proximity to each other, generating massive backlog in the IP-politeness heap and stalling the crawler. After another level of BFS, this script was poised to pollute the queue with a *billion* useless URLs. Even avoidance of dynamic links, as proposed by certain crawlers, would not have helped since all generated URLs pointed to static `.html` pages.

Our finding that BFS was unusable in practice went against common experience in the field [106]. One possible reason for this discrepancy was the scale at which IRLbot operated compared to other crawlers; however, there is another explanation. Specifically, ability of BFS (and other types of frontier management) to avoid spam is determined not only by the *size* of the crawl, but also its *depth*. This is conceptually illustrated in Fig. 2.2, where crawls starting from node Z work great for $\mathcal{D} \leq 3$ downloaded pages, but then get stuck in spam after two levels of BFS. On the other hand, a larger seed set $\{W, X, Y, Z\}$ keeps the crawl shallower (i.e., closer to reputable pages) and reaches much larger coverage with $\mathcal{D} = 12$ before getting bogged down.

To better characterize crawls and their predisposition to accumulate spam, define m to be the size of the seed set. Then, the average depth of a crawl can be estimated by $\log_{lp}(\mathcal{D}/m)$, where lp is the average number of globally unique links generated by each crawled page. IRLbot always seeded off a single page (i.e., $m = 1$), while other crawlers commonly started from a large variety of initial pages – entire dumps of `dmoz.org`, highly ranked pages from prior crawls, and those returned by commercial search engines [22], [27], [37].

When designing IRLbot, it eventually became clear that non-BFS prioritization was absolutely necessary. However, we also realized that it was infeasible to apply classifiers to 3-TB graphs (i.e., 200x RAM size) or even compute PageRank's 332-GB vector of probabilities (i.e., 20x RAM size) in real-time. Instead, IRLbot's approach was to propose that ranking be performed at the granularity of *pay-level domains* (PLDs)[3], which ensured finiteness of resources being ranked, achieved excellent scalability (i.e., complexity $O(1)$ per inserted URL as crawl size $\mathcal{D} \to \infty$), and made rank manipulation expensive as now spammers had to control (e.g., register, hijack) a large number of domains [87].

2.3.3 Admission Control

The final challenge in IRLbot involved converting ranking information into download budgets (e.g., allocated bandwidth to different sites, page priority) and then scalably enforcing these decisions for the various pending pages/domains. This, however, turned out quite difficult due to the huge size of the frontier (i.e., 4 TB, 250x RAM size), which contained a massive number of hosts (641M) and domains (89M). Assigning rank to each page and keeping the frontier in sorted order, as suggested in prior work, while sustaining high insertion rates (i.e., 20K/sec), was simply impossible.

IRLbot's methodology for dealing with the frontier was a novel domain-based admission controller called BEAST [87] that split incoming URLs into an exponentially increasing number of queues. As domains started to exceed allocated budgets, their pending pages were pushed into queues further in the future, allowing higher-ranked domains to be crawled without hinderance. This was a scalable form of external-memory WRR (weighted round-robin) scheduling at the domain level, which we showed possessed $O(1)$ complexity per incoming URL.

Additional testing of BFS involved crawls seeded from a prominent advertising/spam directory with the goal of understanding how long it took for the crawler to get bogged down in millions of co-hosted sites and grind to a halt due to its politeness. This technique produced an unexpected side-effect as spammers eventually developed a distaste for IRLbot and started colluding in delay attacks against its IP, which involved dragging out HTTP connections by serving the page very slowly and/or silently discarding DNS requests to force a 30-timeout on our end. Since IRLbot attempted to concurrently crawl millions of seemingly unrelated sites, many of which were under spammer control, their cooperative attacks stalled the crawler for extended periods of time. However, enabling domain ranking and BEAST put an end to this problem.

---

[3] An Internet domain that requires registration at a gTLD or cc-TLD registrar, e.g., `amazon.co.uk` [87], [126].

2.3.4 Software and Hardware

IRLbot is a multi-threaded implementation written entirely in C++. It was designed from the ground up at Texas A&M University to rival commercial web crawlers, but using modest server hardware. Besides algorithmic challenges addressed in [87], we faced a number of technical problems in achieving high crawling rates.

With download rates peaking at 3K pps in our 2007 experiment, the first fundamental bottleneck was parsing HTML content at sufficient speed and avoiding major CPU contention with other parts of the crawler. In our early designs, we examined previous crawling papers and found no implementation that was shown to deliver this level of performance. In fact, prior literature rarely disclosed any benchmarks related to parsing or specified the nature of their implementation. Two exceptions were Polybot [122], which reported 400 pps of 13 KB each (i.e., 41 Mbps) on a "typical workstation," and Ubicrawler [17], which reported 600 pps of unknown size.

After considering various options, including several open-source parsers, we ended up writing our own version based on the Boyer-Moore algorithm. Using the front

Table 2.1: Parsing speed of http://whitehouse.gov using a single core of a 2.6 GHz AMD Opteron 2218.

Parser	Language	Speed
Html-parser 1.6 [48]	Java	26 Mbps
Majestic12 3.0.9 [35]	C#	540 Mbps
Tokenizer 10.10.07 [49]	C++	640 Mbps
Wget 1.10.2 [56]	C	1.3 Gbps
El-kabong 0.3.2 [45]	C	1.8 Gbps
IRLbot/3.0	C++	2.6 Gbps

page of `whitehouse.gov` (which was 31 KB at that time and contained 115 links) as benchmark, Table 2.1 shows the performance of IRLbot in comparison to other open-source parsers. While speed was one objective, another reason for having a custom parser was the ability to tune its accuracy on malformatted HTML. Performing a study of common discrepancies between Trident (Internet Explorer), Mozilla Gecko (Firefox), and the four non-Java parsers in Table 2.1, we identified 12 errors most commonly encountered in HTML download by IRLbot and made sure that our parser handled these cases correctly. While it generally inspected each byte no more than once, run-away comments and tags sometimes required a second pass over certain parts of the document.

The second fundamental bottleneck in a web exploration of such high performance is the networking subsystem. IRLbot used the Winsock API and ran between 10-15K threads in steady-state. To improve performance, we used the poorly-documented `ConnectEx`-`DisconnectEx` function pair that allowed IRLbot to reuse sockets without closing and rebinding them. This doubled TCP connect performance of our server from 10K/sec to 20K/sec and significantly reduced the impact of the TCP/IP stack on other parts of the crawler.

Another bottleneck was in DNS lookups, where `gethostbyname` saturated at a measly 1.5K queries/s with 100% utilization of the CPU. Writing our own DNS client that operated directly over UDP improved this number by a factor of 9 to 13.5K lookups/s. However, the next bottleneck was our lab's DNS server whose BIND 9.3.2 gradually grew cache to infinity, ignoring cache-size directives in the configuration file, and became really slow. We ended up finding several bugs in BIND and fixing them to allow its cache to operate at top speed.

Due to the huge number of concurrent sites being crawled, we were unable to maintain open connections to each host for the default 40 seconds between page downloads and thus did not utilize persistent features of HTTP/1.1. IRLbot requested compression for all files (including robots.txt) and decoded responses using a standard zlib library [93], which proved quite robust to truncation and various garbage transmitted to the crawler. Its decoding performance was not high, but since only a small fraction of pages was compressed, this never became a bottleneck.

We did not experience any crashes in 41 days, but a few updates were implemented during the crawl, which required checkpointing and restarting.

The hardware consisted of a 2.6 GHz dual-core, dual-socket AMD Opteron 2218 host with 16 GB of DDR2-667 RAM running Windows Server 2003 x64. Storage was provided by two 3ware 9550SX PCI-X cards, each with twelve 750-GB Seagate drives in RAID-5. The cards were then combined using software RAID-0 into a single $22 \times 750 = 16.5$ TB drive. While the system was capable of reading at 1 GB/s, only 100 MB/s of this rate was needed during the crawl. Due to IRLbot's efficiency of external-memory algorithms, this RAID configuration should be viewed not as a way to scale I/O speed, but rather to provide enough storage for various data structures needed during the crawl.

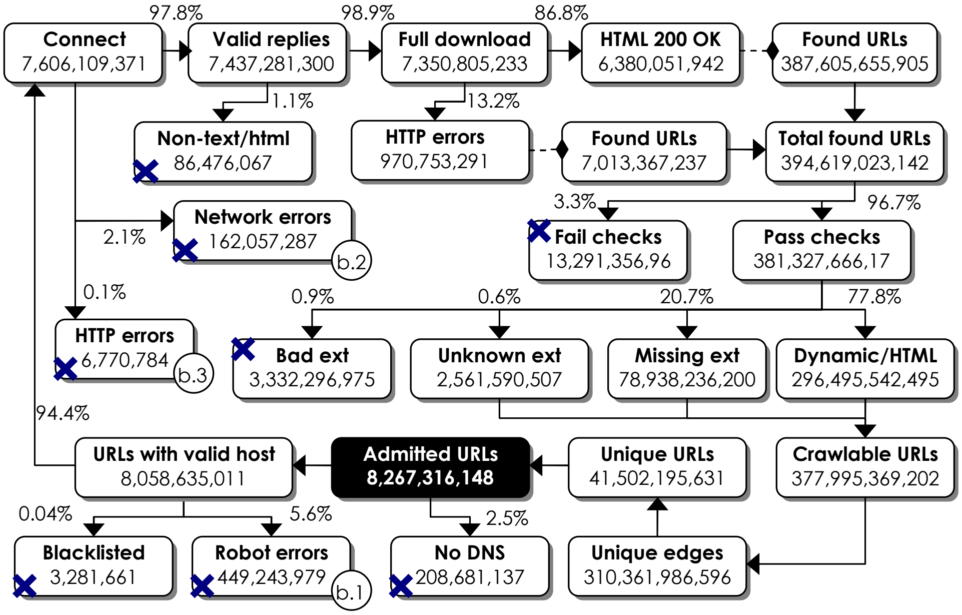
2.3.5 Discussion

For future web-crawling research to provide credible results, we believe that one should engage in experimentation that aims to simultaneously surpass prior crawls in all five metrics introduced earlier in this section – $\mathcal{S}/h$, $\mathcal{D}/h$, q, l, and average depth $\log_{lp}(\mathcal{D}/m)$ – where the first four attest to the crawler's scalability and the last one to its ability to avoid spam. We can also add to this list several auxiliary parameters – the number of crawled hosts, domains, and IPs – since they control the size of various caches, DNS and `robots.txt` workload, complexity of politeness rate-limiting, and Internet coverage. Another relevant metric is the average page size, which together with $\mathcal{S}$ determines the download bandwidth and performance of the network stack.

As we show in the remainder of the chapter, IRLbot is an implementation of this vision using $m = h = 1$, maximum q, and unrestricted l, with $\mathcal{S}$ and $\mathcal{D}$ determined by forces outside our control (e.g., university bandwidth). In 2007, IRLbot seeded off a *single* page `http://tamu.edu`, went to maximum depth 31, and managed to juggle all 41B URLs in the frontier (including every dynamic link) using a single server with 16 GB of RAM, producing the fastest and largest documented crawl to date.

2.4 Page-Level Analysis

We next explain our proposed methodology for documenting large-scale crawls. This section underscores the importance of collecting extensive statistics and meticulously logging various failure conditions, many of which are routinely omitted from prior studies.


97.8%
98.9%
86.8%
Connect
7,606,109,371
Valid replies
7,437,281,300
Full download
7,350,805,233
HTML 200 OK
6,380,051,942
Found URLs
387,605,655,905
1.1%
13.2%
Non-text/html
86,476,067
HTTP errors
970,753,291
Found URLs
7,013,367,237
Total found URLs
394,619,023,142
3.3%
96.7%
Network errors
162,057,287
b.2
2.1%
Fail checks
13,291,356,96
Pass checks
381,327,666,17
0.1%
0.9%
0.6%
20.7%
77.8%
HTTP errors
6,770,784
b.3
Bad ext
3,332,296,975
Unknown ext
2,561,590,507
Missing ext
78,938,236,200
Dynamic/HTML
296,495,542,495
94.4%
URLs with valid host
8,058,635,011
Admitted URLs
8,267,316,148
Unique URLs
41,502,195,631
Crawlable URLs
377,995,369,202
0.04%
5.6%
2.5%
Blacklisted
3,281,661
Robot errors
449,243,979
b.1
No DNS
208,681,137
Unique edges
310,361,986,596

(a) URL cycle

Error type	URLs affected
(b.1) Robots	**449,243,979**
Disallowed	296,966,591
Network error	106,638,856
HTTP error	24,221,293
Forbidden	20,621,185
Loop	612,160
Size over 64KB	183,894
(b.2) Network	**162,057,287**
Connect fail	124,752,717
Receive fail	36,534,906
Slow download	421,427
Page over 4MB	338,872
Send fail	9,365
(b.3) HTTP	**6,770,784**
Bad HTTP response	4,139,148
Decompression fail	1,110,272
Bad HTTP status	682,613
Invalid base URL	593,941
Bad chunking	242,858
Header over 64 KB	1,952

(b) discarded URLs

Figure 2.3: IRLbot crawl analysis.

2.4.1 Admitted URLs

To address the vagueness of prior URL statistics, we propose that they be presented under a unified umbrella, which we call the *URL cycle*. Fig. 2.3 shows its basic structure. Over a period of six weeks, IRLbot pulled $\mathcal{A} = 8.2$B URLs from admission control (the shaded box in part (a) of the figure) and attempted to crawl them. Approximately 2.5% of these pointed to hosts without DNS entries and were immediately discarded. Out of the remaining 8B links, 0.04% were thrown out due to manual blacklisting in response to complaints and an additional 5.6% were dropped during the `robots.txt` phase.

Part (b.1) of the figure shows a detailed breakdown of these errors. This includes 296M URLs prohibited by disallow directives and 106M mapping to hosts with various network errors during the download of robots.txt. The remaining URLs in (b.1) belonged to servers that were unable to correctly complete the HTTP exchange during robots.txt (24M), refused to provide it altogether (20M), entered redirection loops longer than 7 hops (612K), or served files larger than 64 KB (183K), all of which was treated by IRLbot as indication that the website did not welcome crawlers.

While IRLbot attempted redirects on `robots.txt` back-to-back, normal URLs were handled differently. To avoid wasting bandwidth on spam that frequently employed lengthy sequences of redirects, IRLbot treated each 301/302 as a new link (i.e., sent it for regular uniqueness verification and then admission control). This ensured that redirects had to pass spam-related budget enforcement before being attempted again, which made the retry latency dependent on the current rank of the corresponding domain and its URL backlog.

2.4.2 Crawled URLs

After passing the robot phase, $\mathcal{C} = 7.6$B URLs continued through the cycle and were issued non-robot connection requests. This resulted in 162M network errors and 6.7M HTTP failure conditions. The breakdown of the former is shown in part (b.2) of the figure, where the largest category (124M) was caused by connect failures, among which the three most common reasons were timeouts (103M), connection refused (14.7M), and destination unreachable (5.4M). Each `connect()` that failed with a timeout was retried three times back-to-back. Analysis shows that the second attempt increased the number of successful connections by only 0.1% and the third by 0.05%. This suggests that issuing repeated connection requests with high temporal correlation may not be worth the effort in practice; however, organizing the failed URLs into a separate data structure and retrying them later (e.g., at exponentially increasing intervals) might be worth considering in future crawls as a way to overcome temporary server outages.

The second largest category in (b.2) were 36M failed receive calls, which predominantly consisted of TCP timeouts (31M) and connection resets from the remote server (4.9M). In the next category, 421K URLs were aborted when their host either did not provide any data for over 60 seconds or dragged out the page download beyond 180 seconds, which were common spammer tactics aimed at stalling IRLbot. The opposite technique was to serve “infinite” streams of data, which we terminated at 4 MB, resulting in 338K additional URLs being discarded. Despite this limit, the largest page the parser dealt with (after decompression) was a whopping 884 MB. Finally, 9K URLs were reset by the remote peer while IRLbot was sending the GET request.

The most common HTTP failure in (b.3) was the missing status line in the response, which affected 4.1M URLs. Sometimes attributed to ancient HTTP/0.9 servers, this condition might also be indicative of other services running on the contacted port and various firewall/IDS misconfigurations. The second most common error type in (b.3) was failed decompression (1.1M), with gzip corruption responsible for 1M URLs and unknown/bogus encoding type for the other 0.1M cases. Finally, 682K URLs in (b.3) had an invalid HTTP status code (i.e., above 505 or below 100), 593K contained an unparsable base URL, 242K violated the chunking syntax or exceeded 4 MB after unchunking, and 1.9K contained HTTP headers over 64 KB.

Going back to Fig. 2.3(a), $\mathcal{R} = 7.4$B valid replies produced $\mathcal{O} = 86.5$M objects with status 200 OK and content-type other than `text/html`. To build as massive a webgraph as possible and push IRLbot to its scalability limits, we were only interested in downloading HTML pages. Considering that certain non-HTML files were extremely large (e.g., DVDs, ISOs), IRLbot aborted their connections as soon as the HTTP header was received. This curtailed the download to an average of 8.3 KB per object and limited the total wasted bandwidth to just 718 GB. Without header peeking, the crawler would have had to fall back on the 4-MB maximum page size, which could have allowed these 86.5M objects to consume 346 TB in the worst case.

2.4.3 Downloaded URLs

For the remaining $\mathcal{D} = 7.3$B URLs, 60% of which were dynamic (i.e., contained a ?), the HTTP response was fully downloaded by IRLbot. The breakdown of HTTP status codes among this group is shown in Table 2.2, including similar statistics from three other Internet-wide crawls that supply this information. Successful pages (200 OK) accounted for $\mathcal{H} = 6.3$B responses, or approximately $q = 87\%$ of $\mathcal{D}$, and errors for $\mathcal{E} = 970$M pages. Mercator included non-HTML objects in $\mathcal{D}$ and thus exhibited smaller q (e.g., around 58% for the two crawls in the table), while Polybot did not

Table 2.2: HTTP status codes of downloaded URLs

Code	IRLbot	Mercator		Polybot
	$\mathcal{D} = 7.3$B	$\mathcal{D} = 76$M	$\mathcal{D} = 819$M	$\mathcal{D} = 139$M
	$\mathcal{H} = 6.3$B	$\mathcal{H} = 45$M	$\mathcal{H} = 473$M	$\mathcal{H}$ unknown
200	86.79%	87.03%	88.50%	87.42%
302	7.49%	3.33%	3.31%	4.37%
404	3.56%	7.43%	6.46%	5.32%
301	1.12%	1.12%	–	2.08%
500	0.35%	0.11%	–	0.07%
403	0.28%	0.43%	–	0.35%
400	0.10%	0.09%	–	–
401	0.09%	0.30%	–	0.30%
406	0.08%	0.11%	–	–
Other	0.12%	0.06%	1.73%	0.09%

provide enough information to infer this value. Interestingly, half of IRLbot's 200 OK pages (i.e., 3.2B) and 47% of errors (i.e., 456M) were chunked by the server, indicating some type of dynamically assembled content.

The second most-popular code in Table 2.2 (i.e., 302 temporary redirect) was encountered twice as frequently as in previous studies, which was expected following the rise of CDN redirects and various domain-parking/spam activity that was not as prevalent earlier. Not-found 404 errors were much less common in IRLbot than in [64], [105], which might be explained by the peculiarities of each parser, URL-validation logic in each crawler, and emergence of automated link-verification software for webmasters.

To avoid pulling non-html pages, IRLbot transmitted the "Accept: text/html" header with all non-robot.txt requests, which the server should reject with "406 Not Acceptable" if the MIME type does not match the one requested by the client. Combining Table 2.2 and Fig. 2.3 we obtain that IRLbot attempted to download 92.6M non-HTML pages, out of which 6.1M returned with status code 406 and the remaining $\mathcal{O} = 86.5$M with 200 OK. This shows that Internet servers universally ignore the Accept field and that its usage amounts to a disappointing 6.6% reduction in aborted pages.

Considering that 4.4B downloaded URLs (i.e., 60% of $\mathcal{D}$) were dynamic, it follows that at least 700M (i.e., 16%) of them were *not* chunked, possibly due to caching.

2.4.4 Links

Parsing `a-href`, `frame-src`, and `meta-refresh` tags, as well as HTTP "Location:" fields in the 7.3B downloaded responses, IRLbot produced a total of $\mathcal{K}_1 = 394$B links shown in Fig. 2.3(a), with 1.7% coming from non-200 pages. The most prolific HTML page contained 4.6M links and the most verbose error page 110K. Note that unlike some of the prior work [105], we completely ignored `img` tags and did not consider them part of the URL cycle.

To avoid hitting obviously bogus pages, IRLbot tested links for correctness of syntax and canonized them by lower-casing host names, removing username/password, and re-writing directories containing single or double dots (i.e., `/../` and `/./`). To handle common typos and mistakes, IRLbot also removed trailing/leading whitespace, decrypted occasional HTML markup (e.g., `&lt;`), and %-encoded prohibited symbols. In the end, 3.3% of the links were discarded due to invalid syntax or excessive length, where anything longer than 1.2 KB was considered unreasonable.

Out of the remaining $\mathcal{K}_2 = 381$B links, 3.3B pointed to a static page with one of 694 prohibited non-HTML extensions (e.g., office files, music/video). Interestingly, usage of a pretty extensive blacklist reduced the URL workload by only $\epsilon = 0.9\%$ and the number of aborted pages by an estimated $\epsilon\mathcal{R} = 64$M. Given 8.3 KB per aborted page, this translates into 534 GB of saved bandwidth, or a mere 0.37% of the total. This number seems small enough that in future crawls it might be simpler to drop extension filtering and handle *all* non-HTML objects in the download phase.

Returning to Fig. 2.3(a), the remaining URLs were considered suitable for crawling, which included 2.5B static links with an unknown extension, 78B static links without an extension, and 296B links that were either dynamic or indicative of HTML. Condensing 377B crawlable links by removing same-page duplicates and replacing URLs with 64-bit hashes, IRLbot constructed a 3-TB webgraph with $\mathcal{K} = 310$B edges and $\mathcal{U} = 41$B unique nodes, the latter of which was fed into admission control, completing the cycle in Fig. 2.3(a).

Treating links found in error pages as integral byproduct of crawling, it can be estimated that each good HTML page injected $\mathcal{K}_1/\mathcal{H} = 61.7$ URLs into the system, with $\mathcal{K}_2/\mathcal{H} = 59.7$ of them valid. However, only $l = \mathcal{K}/\mathcal{H} = 48.6$ were locally unique. We can now determine the probability that a locally unique link is globally unique, i.e., $p = \mathcal{U}/\mathcal{K} = 0.13$, and the number of URLs injected into admission control per crawled page, i.e., $lp = \mathcal{U}/\mathcal{H} = 6.58$. This allows us to estimate IRLbot's average crawl depth as $\log_{lp}(\mathcal{D}/m) = 12$, where $m = 1$ is the number of seed nodes. For comparison, the same metric in ClueWeb09 [22], [37] with its $m = 33$M and $\mathcal{D} = 1$B is only 1.8.

2.4.5 Summary

We next introduce several novel metrics that help one quantify the type of downloaded material. The first measure is *conversion rate*

$$r = \frac{\mathcal{H} + \mathcal{O}}{\mathcal{A}}, \tag{2.1}$$

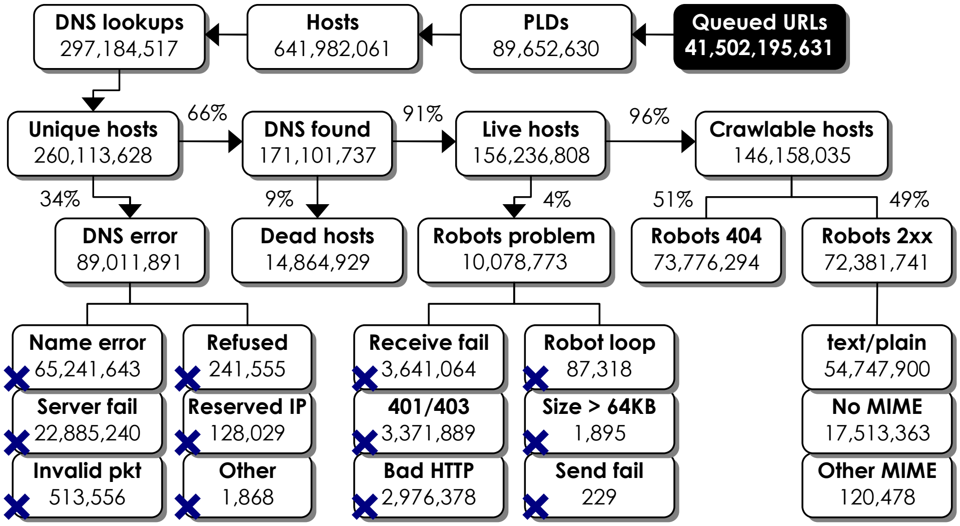
which is the fraction of attempted URLs that yield error-free (i.e., 200 OK) content of any type. The second one is *HTML download density*

$$d = \frac{\mathcal{H}}{\mathcal{H} + \mathcal{O}}, \tag{2.2}$$

which is the fraction of error-free URLs that are HTML pages. Note that crawls with small r are full of errors, while those with small d are mostly non-HTML objects. For IRLbot, we obtain $r = 78\%$ and $d = 98.6\%$. Applying this to its frontier with $\mathcal{U} - \mathcal{A} = 33$B pending URLs shows that about $rd = 77\%$ of them (i.e., 25B) are expected to be 200 OK HTML pages.

With the exception of [64], [105], which exhibited $r = 81-85\%$ and $d = 65-69\%$, prior studies do not provide sufficient information to compute either metric. One reason for IRLbot's lower conversion rate compared to Mercator [64], [105] was 2.5 times more DNS errors per attempted URL, which is not surprising given several orders of magnitude larger pool and higher aggregate expiration rate of crawled domains over IRLbot's 41 days vs Mercator's 17. Another reason was the 1.4-times-higher probability to encounter a link disallowed by robots.txt, which also makes sense given an increase in webmaster awareness to protect sensitive dynamic content (e.g., shopping carts) through robots.txt. IRLbot's higher HTML density stemmed from no attempts to parse or download `img` tags, which in prior crawlers [64], [105] accounted for 26-30% of all discovered links.

To perform a self-check and verify that text/html classification was reasonably correct during the crawl, define $\mathcal{H}_{out}$ to be the number of 200 OK HTML pages with at least one valid outgoing link in `a-href`,`frame-src`, or `meta-refresh` tags. It is expected that the majority of HTML pages should contain some number of valid



DNS lookups 297,184,517
Hosts 641,982,061
PLDs 89,652,630
Queued URLs 41,502,195,631
Unique hosts 260,113,628
66%
DNS found 171,101,737
91%
Live hosts 156,236,808
96%
Crawlable hosts 146,158,035
34%
DNS error 89,011,891
9%
Dead hosts 14,864,929
4%
Robots problem 10,078,773
51%
Robots 404 73,776,294
49%
Robots 2xx 72,381,741
Name error 65,241,643
Refused 241,555
Server fail 22,885,240
Reserved IP 128,029
Invalid pkt 513,556
Other 1,868
Receive fail 3,641,064
Robot loop 87,318
401/403 3,371,889
Size > 64KB 1,895
Bad HTTP 2,976,378
Send fail 229
text/plain 54,747,900
No MIME 17,513,363
Other MIME 120,478

Figure 2.4: DNS and robots.txt requests.

links, which can be assessed using our final measure we call *link presence probability*:

$$p = \frac{\mathcal{H}_{out}}{\mathcal{H}}. \tag{2.3}$$

Analysis of the IRLbot dataset shows $p = 91\%$; however, none of the previous papers are detailed enough to infer this metric.

2.5 Server-Level Analysis

We now deal with network statistics. While prior crawls provide some limited host-related information, there is almost no discussion of various IP-level interactions, experienced errors, robot downloads, or consumed bandwidth. We use IRLbot data to present our approach for streamlining this type of exposition.

2.5.1 DNS and Robots

Our first topic is crawler interaction with remote hosts and their authoritative DNS servers. We present the proposed model of this breakdown in Fig. 2.4. IRLbot's 41B discovered URLs belonged to 89M pay-level domains (PLDs) and 641M sites. The PLD of each host was determined by consulting our list of 1761 domain suffixes, which we constructed by manually examining the registration structure of each available cc-TLD at that time.[4] To avoid unnecessary overhead, IRLbot issued DNS queries only for those hosts whose URLs passed the budget enforcer. This resulted in 297M DNS queries for 260M unique hosts, where repetition was caused by expiration of previously pulled records.

Interestingly, only 171M hosts (66%) had a valid DNS entry and the rest led to DNS errors. Among 89M errors in the figure, 65M were caused by non-existent A records and 22M by expired or unreachable domains. The more obscure errors included 513K hosts whose DNS provided no IP in the response or complained that our request was invalid, 241K that refused to answer the query, 128K with an IP that was either reserved (e.g., 10.x, 127.x, 192.168.x, class E) or belonged to the multi-cast address space, and 1.8K with timeouts or truncated responses.

Among the hosts with valid DNS, 156M were live during the attempted downloads of `robots.txt`. Out of these, 10M failed to provide a legitimate `robots.txt`, which prevented IRLbot from knowing which parts of the website should be excluded and resulted in the entire host being treated as non-crawlable. A breakdown of these errors in the figure includes 3.6M receive errors, 2.9M HTTP problems, and 87K robot-redirect loops. Additionally, there were 3.3M hosts that refused to provide the robots file with 403-forbidden or 401-unauthorized. While this was likely caused by misconfiguration, we followed the robot standard and excluded these hosts from future visits.

Among the remaining 146M hosts in the figure, approximately half (73M) did not use `robots.txt` and one-third (54M) served it with the correct `text/plain` MIME type. Additionally, 17M hosts provided robot files with no content-type and 120K used a non-`text/plain` type. The last two cases were often seen on servers that sent HTTP filler (e.g., custom error messages, redirects to default pages, ads) instead of proper errors, which we interpreted as equivalent to not having any crawling restrictions.

---

[4] There is now a Mozilla project with similar goals [103].

The 54M legitimate `robots.txt` files originally occupied 115 GB (i.e., 2.2 KB on average); however, after retaining only the directives that applied to either all crawlers or IRLbot specifically, the entire dataset shrunk to 1.5 GB (i.e., 28 bytes/host) and contained just 101M entries (i.e., 1.85 entries/host). This indicates that the entire collection can be easily cached in RAM rather than on disk as done by IRLbot.

Further analyzing robot files, we found that 1,421,150 (2.6%) contained the `crawl-delay` parameter restricting the frequency of visits to the site. While this extension was not officially standardized at the time, IRLbot and most commercial search engines had supported it for years. Nevertheless, website adoption of this parameter was still apparently scarce. The discovered crawl delay was spread among 53 unique values, all contained between 1 to 255 seconds. The three most popular delays were 20 sec (39%), 60 sec (17%), and 10 sec (12%), with 63% below 20 sec and 85% below 1 minute.

In terms of Internet coverage, 171M sites with a valid DNS entry mapped to $5,517,743$ unique IPs, all of which were probed by IRLbot during attempted downloads of `robots.txt`. However, a more balanced characterization of a crawl includes only hosts with 200-OK HTML content. In that case, analysis shows that $\mathcal{H} = 6.3$B pages resided on $117,576,295$ sites, $33,755,361$ PLDs, and $4,260,532$ IPs.

2.5.2 Bandwidth

In the outbound direction, IRLbot transmitted approximately 23 GB of DNS traffic and 33 GB of robot requests. GET packets consumed an additional 1.8 TB in HTTP headers and 1.1 TB in TCP/IP overhead. Combining these with DNS and robots, the 3 TB of outbound traffic (i.e., 6.7 Mbps sustained) becomes noticeable. Inbound bandwidth was split across 37 GB of DNS responses, 254 GB of `robots.txt` files (including repeated requests), 718 GB of aborted objects, and 143 TB of fully downloaded URLs. A breakdown of the last category is shown in Fig. 2.5.

Starting with 7.3B HTML pages on top of the figure and following the path on the right, observe that 16% of all 200-OK pages arrived compressed and accounted for 6.6 TB (i.e., 6.3 KB/page). After decoding, they ballooned to a hefty 33 TB (i.e., 31.5 KB/page), showing a perfect 5 : 1 compression ratio. The other 5.3B OK pages arrived uncompressed and consumed 128 TB as also shown in the figure. These pages were quite a bit smaller, averaging 24 KB. Not including HTTP headers, both types of 200 OK pages consumed 134 TB of download bandwidth, or 93.8% of the total.

HTTP errors on the left side of Fig. 2.5 experienced half the likelihood of being compressed, slightly lower deflate ratios (4.7 : 1), and significantly smaller average page size, which ranged from 1.3 KB for compressed to 2.4 KB for uncompressed responses. Interestingly, error pages did not just end with the HTTP header; instead, many servers were compelled to stuff additional data after the header, including bizarre cases when it was impossible for the browser to display them. Even with this extra content, 970M HTTP errors accounted for only 1.5% of the final traffic. TCP/IP overhead (4 TB) and HTTP headers across all responses (2.5 TB) were actually more prominent.

We next compute the elusive average page size. From the perspective of band-

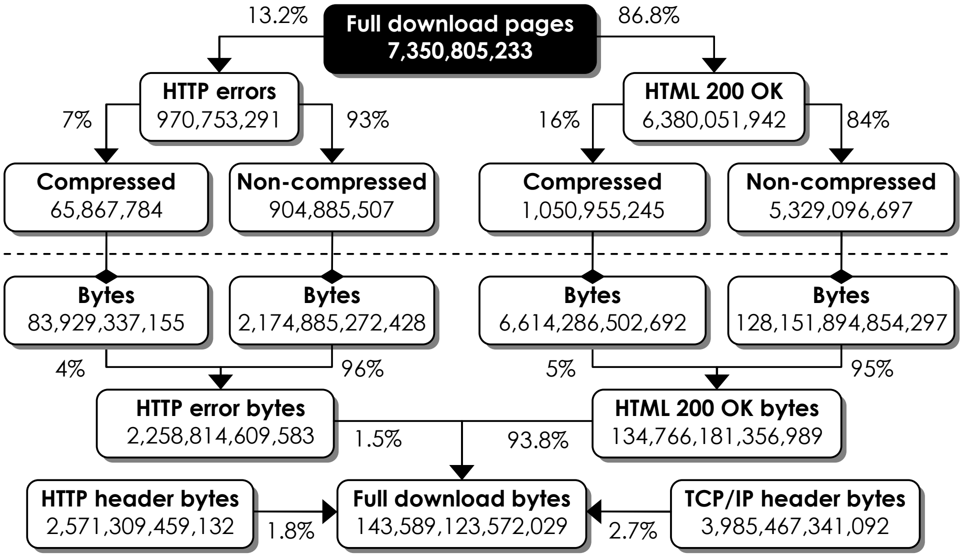

Full download pages
7,350,805,233
13.2%
86.8%
HTTP errors
970,753,291
7%
93%
HTML 200 OK
6,380,051,942
16%
84%
Compressed
65,867,784
Non-compressed
904,885,507
Compressed
1,050,955,245
Non-compressed
5,329,096,697
Bytes
83,929,337,155
Bytes
2,174,885,272,428
Bytes
6,614,286,502,692
Bytes
128,151,894,854,297
4%
96%
5%
95%
HTTP error bytes
2,258,814,609,583
1.5%
93.8%
HTML 200 OK bytes
134,766,181,356,989
HTTP header bytes
2,571,309,459,132
1.8%
Full download bytes
143,589,123,572,029
2.7%
TCP/IP header bytes
3,985,467,341,092

Figure 2.5: Downloaded HTML pages.

width provisioning, 143 TB of HTML in Fig. 2.5 are needed to obtain $\mathcal{H} = 6.3$B error-free pages, which results in 22.4 KB of network overhead per HTML page. If we focus on the average number of bytes per connection (i.e., including error pages), this number becomes 19.3 KB. However, from the content-analysis perspective, a more relevant metric is $128 + 33 = 161$ TB of HTML that went through the parser, which produces 25.3 KB on average. Prior work rarely specifies this value, but even when it does (e.g., 13.3 KB in [122]), it is unclear which of the methods above was used to compute it.

Across the entire experiment, IRLbot averaged 2132 attempted connections/s, $\mathcal{S} = 2061$ full downloads/s, $\mathcal{S}q = 1789$ error-free HTML pages/s, and 320 Mbps of inbound and 7 Mbps of outbound bandwidth.

2.6 Extrapolating Crawls

Given the large number of documents crawled by IRLbot, and even more discovered, one may wonder about the growth of various datasets and their finiteness as the crawl continues beyond the already-seen portions of the web. We next examine this question in more detail.

2.6.1 Stochastic Model

Given a web graph $G = (V, E)$ of the entire Internet, any crawl can be viewed as a stochastic process $\{(X_n, Y_n)\}$, where $n = 1, 2, \ldots$ is discrete time, $X_n \in V$ is the crawled page that generated link n, and $Y_n \in V$ is the URL it points to. We assume this process excludes invalid URLs, ignores same-page duplicates, and terminates after finding $N \leq |E|$ links. As the crawl progresses, we are interested in the behavior of discovery rates for new URLs, hosts, and/or PLDs. To cover all of these under a common umbrella, define indicator variable Q_n to be 1 if link (X_n, Y_n) satisfies some uniqueness condition and 0 otherwise. For example,

$$Q_n = \begin{cases} 1 & Y_n \text{ not seen before} \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

defines a non-stationary stochastic process of URL uniqueness. Then, the expected number of links L_N satisfying this condition in a crawl of size N is:

$$E[L_N] = \sum_{n=1}^{N} E[Q_n] \approx \int_1^N p(t)dt, \tag{2.5}$$

where N is assumed to be very large and $p(t) = P(Q_t = 1) = dL_t/dt$ is the growth rate of unique nodes at time t.

It is normally expected that $p(t)$ starts off high for small t; however, as the crawler starts exhausting the web, L_t should begin experiencing saturation and thus $p(t)$ should eventually decay to zero. In this regard, two questions are in order. First, what general model does $p(t)$ follow? Second, can one estimate the number of crawled pages C_N that produce a given value L_N? For example, Google reported in 2008 reaching 1T unique nodes in the webgraph [6]. How many more pages does IRLbot have to crawl to hit the same target?

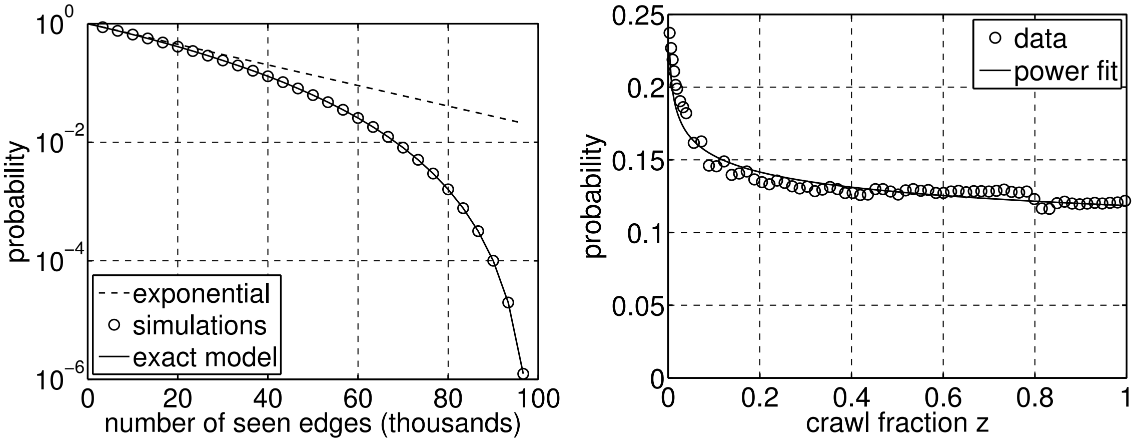
To build intuition, we answer the first question using a toy model of the web and predicate function (2.4). Assume the web is a finite digraph with a constant in/out-degree d and suppose the crawl implements a uniformly random shuffle on E. Then, (X_t, Y_t) points to a unique page if and only if none of Y_t's other $d-1$ in-links has been discovered in $[1, t-1]$, i.e.,

$$p(t) = \Big(1 - \frac{t-1}{|E|}\Big)^{d-1}. \tag{2.6}$$

For $t \ll |E|$, which is a common operating range of interest, Taylor expansion reduces (2.6) to $e^{-\lambda t}$, where $\lambda = (d-1)/|E|$. Fig. 2.6(a) shows simulations in comparison to (2.6) in a random graph with $|E| = 100$K edges and $d = 5$ (i.e., $|V| = 20$K nodes). It thus can be expected that in some cases $p(t)$ may exhibit an exponential tail, although real graphs are typically more complicated and require modeling work beyond the scope of this chapter. See [3] for details.

For the second question, using (2.6) in (2.5) and solving for N produces the number of edges that must be seen for a given value of L_N. Dividing the result by the fixed out-degree d yields the number of needed pages C_N in the crawl. For the last question, assume that $e^{-\lambda t}$ is fit to $p(t)$ in some initial range $t \in [1, t_0]$. If d is known, we immediately obtain $|V| \approx (d-1)/(d\lambda)$. If d is unknown and large, then $|V| \approx 1/\lambda$. Finally, in more general graphs G or when d is unknown, but small, estimation of $|V|$ requires the total number of edges $|E|$ and the entire curve $p(t)$:

$$|V| \approx \int_1^{|E|} p(t) dt; \tag{2.7}$$



(a) toy graph (b) IRLbot ($\alpha = 0.12$)

Figure 2.6: Model verification for $p(t)$ and URL discovery rate $\tilde{p}(z)$ in IRLbot.

however, if $|E| = \infty$, then $|V|$ can be obtained from the empirical $p(t)$ without any additional parameters.

When $p(t)$ is only known in a small interval $[1, t_0]$, integration of the extrapolated tail in the toy example provides an *upper* bound on $|V|$ as seen from Fig. 2.6. As modeling $p(t)$ is significantly more complex for degree-irregular G and crawls that do not draw edges uniformly from E, it is unclear whether similar conclusions hold in the general case.

We next provide a methodology to first build and then extrapolate curves $p(t)$ using real crawls. This will answer both questions posed earlier in a more realistic setting.

2.6.2 Data Extraction

We start by introducing an algorithm for estimating function $p(t)$ in a discrete set of points $t_1, t_2, \ldots, t_k$ using a given crawl dataset. Our discussion centers on URL uniqueness, but almost identical procedures apply in other cases (e.g., hosts, PLDs). Note that points $\{t_i\}$ may be spaced non-uniformly (e.g., at exponentially increasing distances), depending on the desired parameters of the plot. Define bin $b_i = [t_i - \Delta, t_i + \Delta]$ to be some Δ-neighborhood of t_i that contains enough discovered links for the law of large numbers to hold.

Assume that page-crawl timestamps $\tau_1, \tau_2, \ldots$ are embedded in the webgraph with each source node. For every link (j, k), we first determine bin i into which timestamp τ_j falls and increment the corresponding number of seen URLs s_i for that bin. Since the number of bins is usually small (e.g., 50-100), these counters can be kept in RAM. We then map (j, k) to a tuple consisting of k's hash h_k and the crawl timestamp of the source page j, i.e., (h_k, τ_j). After all tuples are sorted by h_k, the reduce step retains the smallest timestamp for each seen URL, i.e., $(h_k, \tau_j^1, \tau_j^2, \ldots) \rightarrow (h_k, \min(\tau_j^1, \tau_j^2, \ldots))$. Scanning the final result, we obtain the number of globally unique links u_i discovered in each bin i, which produces $p(t_i) \approx u_i/s_i$.

This computation on the IRLbot dataset requires sorting 310B tuples (i.e., 3.7 TB assuming 4-byte timestamps) and produces 41B tuples (i.e., 492 GB) as output. While none of this fits in RAM, IRLbot uses disk-based algorithms that tackle this problem using a single host in a few hours.

2.6.3 URLs

Assume $\mathcal{K}$ is the number of links in the *already-crawled* portion of the web. Then, let $z = t/\mathcal{K}$ be time normalized to this crawl and $\tilde{p}(z) = p(z\mathcal{K})$ be the corresponding uniqueness function. Using equally spaced bins and $\mathcal{K} = 310$B, Fig. 2.6(b) shows that IRLbot's $\tilde{p}(z)$ is a close fit to a power-law function $\beta z^{-\alpha}$, where $\alpha = 0.12$ and $\beta = 0.11$. Interestingly, this decay rate is significantly slower than predicted by (2.6), indicating that the degree distribution of G and crawl order (e.g., bias towards popular nodes) have a noticeable impact on the resulting curve.

We can now offer a crude model for estimating the number of crawled pages C_N

at which IRLbot would hit Google's $L_N = 1$T unique URLs. To do this, we must first determine the number of links N needed to generate L_N globally unique nodes. Defining $r = N/\mathcal{K}$ and re-writing (2.5) in terms of normalized time, we get:

$$E[L_N] \approx \mathcal{K} \int_0^r \tilde{p}(z)dz, \tag{2.8}$$

where the lower limit of the integral $1/\mathcal{K}$ is approximated with a zero. As we specifically aim for cases with $r > 1$, we can split the integral into two segments, i.e., with $z \in [0, 1]$ representing the already-crawled pages and $z \in [1, r]$ being the extrapolated portion:

$$E[L_N] \approx \mathcal{U} + \mathcal{K} \int_1^r \tilde{p}(z)dz, \tag{2.9}$$

where $\mathcal{U} = 41$B is the number of unique nodes in the crawled dataset. For the Pareto tail $\tilde{p}(z) = \beta z^{-\alpha}$:

$$E[L_N] \approx \mathcal{U} + \frac{\mathcal{K}\beta(r^{1-\alpha} - 1)}{1 - \alpha}, \tag{2.10}$$

which in turn leads to:

$$r \approx \Big(1 + \frac{(1-\alpha)(E[L_N] - \mathcal{U})}{\mathcal{K}\beta}\Big)^{\frac{1}{1-\alpha}}. \tag{2.11}$$

For $\tilde{p}(z)$ in Fig. 2.6(b) and $E[L_N] = 1$T, this model suggests $r = 40$ times more discovered edges, which produces $N = r\mathcal{K} = 12$T links in the webgraph and $C_N = N/l = 256$B crawled pages. For 30T unique nodes seen by Google in 2012 [123], we obtain $r = 1918$, $N = 592$T links, and $C_N = 12$T crawled pages. Using their current 20B pages/day crawl rate [123], this amounts to 50 months worth of crawling @ 41 Gbps. Development of network stacks that can keep up with quad 10-Gbps adapters and scaling the crawler to this rate would make a challenging future project.

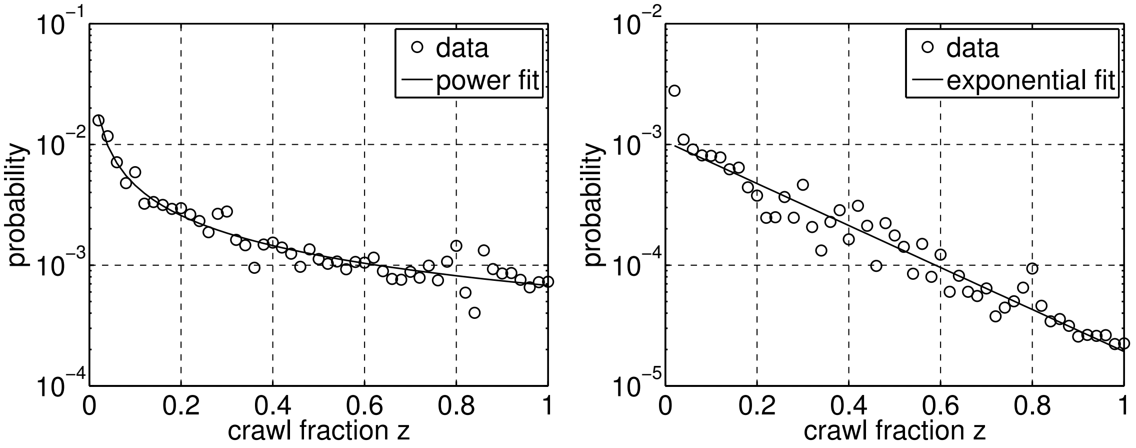
The tail of $\tilde{p}(z)$ in Fig. 2.6(b) is heavy enough, i.e., $\alpha < 1$, that its integral becomes unbounded as the number of seen links $N \to \infty$. While this result was anticipated knowing that scripts could generate arbitrary amounts of unique URLs, the model clearly confirms that IRLbot was on track to experience this problem firsthand and estimates the expected growth rate of $E[L_N]$ in (2.10) as $\Theta(N^{0.88})$. This again cautions against attempting to download every possible unique URL and underscores the importance of prioritizing the frontier.

2.6.4 Hosts and PLDs

Applying the same methodology to the host graph, we obtain curve $\tilde{p}(z)$ shown in Fig. 2.7(a). While it drops more dramatically over the same range (i.e., by a factor of 15 instead of just 2), it still follows a power-law function, where now $\alpha = 0.79$ and $\beta = 0.0008$.

Invoking (2.10) with $r = 40$ and $\mathcal{U} = 641$M unique nodes in the crawled portion leads to 2B extrapolated hosts, which is only 3 times larger than seen by IRLbot so far. For $r = 1918$, this number scales up to 5.2B, but still remains quite reasonable. As before, $\alpha < 1$ predicts an infinite number of hosts, but their growth rate $\Theta(N^{0.21})$ is significantly slower than for unique URLs. Understanding this scaling behavior is quite useful in future designs of site-related data structures, their processing algorithms (e.g., ranking, DNS caching), and storage provisioning.

In contrast to the previous two curves, the PLD-uniqueness probability in Fig. 2.7(b) exhibits a different shape with a much more aggressive decay, dropping by a factor of 122 over the course of the crawl. A curve-fit suggests an exponential



(a) host ($\alpha = 0.79$) (b) PLD ($\lambda = 4$)

Figure 2.7: Host/PLD discovery rate $\tilde{p}(z)$ in IRLbot.

tail $\mu e^{-\lambda z}$ with $\lambda = 4$ and $\mu = 0.0011$, which reminds of the approximate toy model considered earlier in this section, but with an extra parameter μ. Solving the integral in (2.9) leads to:

$$E[L_N] \approx \mathcal{U} + \frac{\mathcal{K}\mu}{\lambda}(e^{-\lambda} - e^{-\lambda r}), \tag{2.12}$$

which produces $E[L_\infty] - \mathcal{U} = 1.6$M PLDs in addition to the 89M already discovered, regardless of future crawl size. IRLbot's spam avoidance has relied precisely on this fact, i.e., that spammers could not obtain control over an infinite number of PLDs.

2.7 Internet-Wide Coverage

We next examine how to quantify crawl coverage of the available web space.

2.7.1 Basic Properties

Crawl coverage may be measured by 1) the collection of URLs from which 200 OK HTML content was obtained; or 2) the constructed graph of the visible web. Note that the latter consists of the former combined with the nodes in the frontier, HTTP

Table 2.3: Internet coverage of existing crawls

Dataset	Crawled (200 OK)				Web		Host		PLD		TLD	
	pages	hosts	PLD	TLD	nodes	edges	nodes	edges	nodes	edges	nodes	edges
AltaVista [19]	–	–	–	–	271M	2.1B	–	–	–	–	–	–
Polybot [122]	121M	5M	–	–	–	–	–	–	–	–	–	–
Google [14]	–	–	–	–	1.3B	20B	13M	395M	–	–	–	–
Mercator [20]	429M	10M	–	–	–	18.3B	–	–	–	–	–	–
WebFountain [43]	1B	–	–	–	4.8B	37B	20M	1.1B	–	–	–	–
WebBase [28]	98M	51K	–	–	–	4.2B	–	–	–	–	–	–
ClueWeb09 [37]	1B	–	–	–	4.8B	7.9B	–	–	–	–	–	–
IRLbot	6.3B	117M	33M	256	41B	310B	641M	6.8B	89M	1.8B	256	46K
UbiCrawler `.uk` [17]	105M	114K	–	1	105M	3.7B	114K	–	–	–	1	1
IRLbot `.uk`	197M	3M	1.2M	1	1.3B	9.5B	5M	54M	1.5M	18M	1	1
TeaPot`.cn` [147]	837M	17M	790K	1	837M	43B	17M	–	790K	–	1	1
IRLbot `.cn`	209M	3.3M	539K	1	1.1B	12B	8.4M	103M	711K	20M	1	1

errors, and the links connecting them together. We include errors in the webgraph as they provide valuable information about redirects (301/302), dead nodes (404/50x), forbidden URLs (401/403), and parents of crawled pages. These might be useful for merging duplicate pages, spam detection, general page ranking, and back-tracing the crawl tree, which may pass through errors, in case of complaints.

Table 2.3 shows a snapshot of available information about the major crawls in the literature. Interestingly, some papers discuss only the crawled pages (e.g., [122]), others only the web graph (e.g., [14], [19]), while some do both (e.g., [20], [43]), but often using a small subset of the possible metrics of interest. Comparison is further complicated by various missing information and unspoken assumptions. For example, Mercator includes `img` tags in the webgraph, while other crawlers typically do not. WebBase considers HTTP errors and robot files as *crawled* pages, while others usually omit `robots.txt` from the totals and include errors only in the webgraph. UbiCrawler removes the frontier and all dangling nodes (i.e., with out-degree zero) from the webgraph, while other datasets normally retain them.

The bottom line is that accurate comparison of previous crawls is difficult; however, given the exhaustive level of detail provided earlier in this chapter, IRLbot results should be straightforward to interpret.

2.7.2 Observations

The first outcome from the table is the emergence of new crawl classification. If a dataset exhibits a high ratio R of pages to hosts, we call its crawl strategy *narrow*. This is exemplified by WebBase, which usually covers $30 - 50$K unique hosts from a pre-defined list and downloads on average $R \approx 2000$ pages per site. Another example is UbiCrawler [17] whose periodic crawls of `.uk` exhibit $R \approx 1000$. If ratio R is low, we call the corresponding strategy *wide*. This category in the table includes Polybot with 24 pages/host, Mercator with 42, TeaPot with 49, and IRLbot with 54.

The second classification relates to the growth in the number of discovered URLs as a function of crawl size. As discussed earlier, this is a crucial metric that determines the workload of the crawler pipeline. We call a dataset *sparse* if the number of links l per downloaded HTML page is small and *dense* otherwise. The former group's sole representative in the table is ClueWeb09 with 7.9 links/page, while the latter group consists of WebFountain with 37, Mercator with 42.7 (including `img` tags), WebBase with 42.8, and IRLbot with 48.6. Among the country-limited graphs, UbiCrawler `.uk` exhibits $l = 35.2$ (excluding dangling nodes), IRLbot `.uk` 48.2, TeaPot `.cn` 51.6, and IRLbot `.cn` 57.

Further examination of ClueWeb09 with its peculiar 7.9 links per page revealed that 72% of the crawled 200 OK HTML pages had *zero* out-degree. Knowing that the first number should have been much higher and the second much lower (e.g., 9.6% in WebBase, 9% in IRLbot), we conducted additional analysis, which uncovered that ClueWeb09 neither crawled dynamic pages, nor included them in the webgraph. From our data, this should have reduced the frontier by an estimated $1 - 7.9/48.6 = 84\%$, or possibly even more if page link density increased between 2007 and 2009.

Since narrow and wide crawls are driven by entirely different objectives, their coverage cannot be directly compared to each other. Similarly, it makes little sense to compare sparse and dense webgraphs. Additionally, *focused* crawls (such as `.uk` [17] and `.cn` [147] in the table) aim to exhaust all pages within a given TLD; however, this is done by following only links with both source and destination contained in the target TLD, which non-focused crawlers do not do. Due to these vast differences, IRLbot can be viewed as compatible in objectives and methodology only with the first five crawls in the table, whose dates precede IRLbot's by $3 - 8$ years and whose statistics unfortunately consist mostly of dashes.

Table 2.4: Commercial datasets

Dataset	Date	Pages
Google	1/2008	$30,756,383,801$
Yahoo	1/2008	$37,864,090,287$

2.7.3 TLD Coverage

Besides the raw totals in Table 2.3, another important aspect of Internet-scale crawling is allocation of budgets to individual domains. Since no prior methods have been developed for measuring this and given that comparison of different crawls has been largely limited to graph-theoretic metrics of the webgraph (e.g., size of various bow-tie components [19], [147]), our aim in this section is to develop a novel approach for understanding how much of crawler bandwidth is spent in what parts of the Internet.

We leverage *site queries* (i.e., strings in the form of "site:domain") that can be submitted to popular search engines to restrict the outcome to a particular domain. In the result page, both Google and Yahoo (now part of Bing) offer an estimated count of how many pages from that domain are contained in their index. We have verified that these counts are exact for small domains and have no reason to doubt their ballpark accuracy for larger domains. Running site queries for all gTLDs and cc-TLDs allows one to obtain not just the index size of a search engine, but also its distribution of pages between various top-level domains. The results are summarized in Table 2.4, which shows that Google's self-reported index contained 30.7B pages, while Yahoo's contained 37.8B at that time.

In order to compare the coverage within each TLD, we designate one of the sets as the *base* and sort all domains in the descending order of the number of pages in the base dataset. Furthermore, instead of using raw page counts, which are functions of

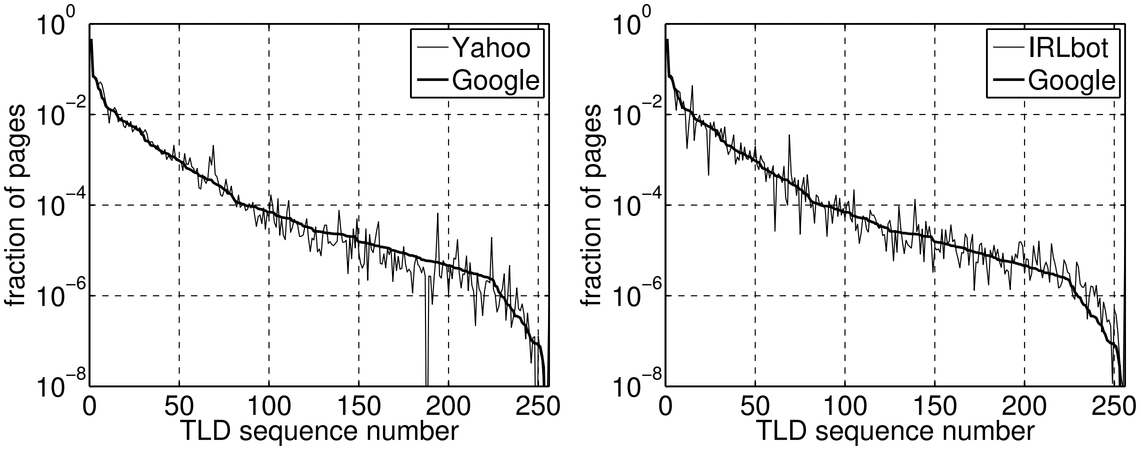
Table 2.5: Google-ordered top-10 TLD List

TLD	Google	Yahoo	IRLbot	WebBase	ClueWeb
.com	46.7%	38.3%	43.3%	31.2%	54.8%
.net	6.9%	7.7%	6.9%	2.2%	6.7%
.de	6.6%	6.8%	7.4%	3.8%	3.8%
.org	5.5%	6.3%	6.6%	17.8%	6.6%
.cn	3.7%	4.6%	3.3%	0.2%	5.6%
.jp	3.4%	5.2%	1.2%	1.7%	3.2%
.ru	2.3%	4.6%	3.3%	0.6%	0.1%
.uk	2.2%	3.0%	3.1%	4.9%	1.7%
.pl	1.6%	1.9%	1.3%	0.2%	0.3%
.nl	1.4%	1.4%	2.0%	0.5%	0.1%
TLDs	255	256	256	174	254

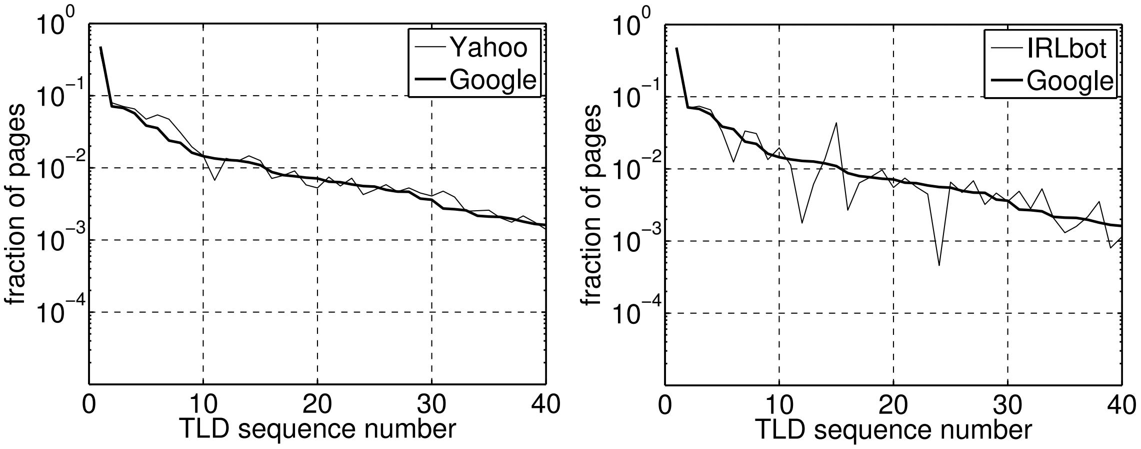
crawl size, we are more interested in *fractions* of each crawl allocated to each TLD. To highlight this better, Table 2.5 shows the top-10 list using Google as the base (for the three academic crawls, we use only HTML 200 OK pages).

We include a WebBase crawl that took place concurrently with IRLbot's and ClueWeb09 (both already detailed in Table 2.3) to highlight the fact that individual crawl policy may purposely favor skewed allocation of resources across domains, leading to a drastically different Internet coverage from that of other crawlers. In this case, the difference occurs because WebBase was interested only in a handful of sites, while ClueWeb09 targeted only static pages in 10 specific languages.

Fig. 2.8 plots the entire TLD curve using Google as the base. Yahoo's deviation at the beginning of the plot in part (a) is noticeably smaller than that of IRLbot in part (b); however, eventually the two curves exhibit random oscillations of similar magnitude. A closer look at the 40 most-popular domains in subfigures (c)-(d) reveals that IRLbot's biggest discrepancy appears in three points – `.edu` (#12), `.gov` (#24), and `.info` (#15) – where the first two are under-crawled and the third one is over-crawled.



(a) Yahoo (all) (b) IRLbot (all)



(c) Yahoo (top 40) (d) IRLbot (top 40)

Figure 2.8: TLD coverage (Google order).

The former case can be explained by IRLbot's budget function that favored TLDs with many individual domains. Both `.edu` and `.gov` contained a small number of unique PLDs, which despite their high ranking were given a relatively low aggregate budget in comparison to other domains. The issue with `.info` can be traced to the large number of $0.99/year spam PLDs hosted there at the time, which conceivably were either removed from Google's index or significantly throttled down during crawling using techniques that were not available to IRLbot (e.g., content analysis, ranking from prior crawls, user click behavior).

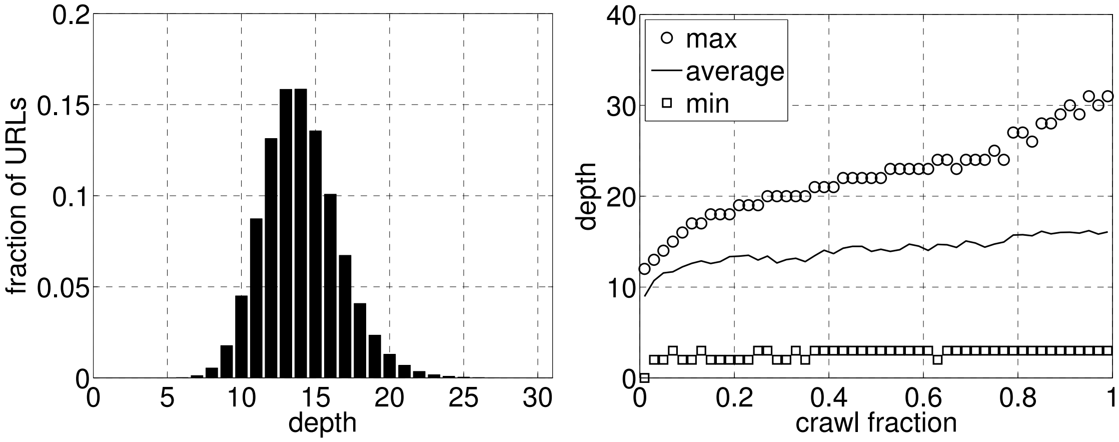
In general, performing analysis of TLD coverage helps one detect over/under-represented parts of the web in crawl data, identify spam regions, and tune budget-allocation policies, all of which are beneficial tools for future crawler development.

2.7.4 More on Budgets

We next shift focus to a related issue – the effect of IRLbot's PLD budgeting mechanism on crawl penetration into individual domains – which is the underlying cause for the discrepancy in `.edu` and `.gov` seen in Fig. 2.8. Any web crawl can be viewed as a forest (i.e., union of trees) with root nodes consisting of the seed URLs and children of each node v being the *globally unique* links discovered in v. Unlike the majority of crawlers in the literature that start from millions of white-listed pages, IRLbot seeded off a single page `http://tamu.edu`. This was done to increase the depth of the crawl, obtain maximum exposure to spam traps, and simplify analysis of the forest later.

Considering the root at depth 0, Fig. 2.9(a) shows a histogram of depth for all 7.3B URLs downloaded by IRLbot. The furthest page from the root was 31 clicks away and 58% of the downloaded material was located at depth 12-15, which aligns well with our estimate $\log_{lp}(\mathcal{D}/m) \approx 12$ earlier in the chapter. While IRLbot went pretty deep, it turns out that our linear budget function [87] favored new PLDs over crawling many pages within the same domain. As a result, much of the depth was acquired *across domain boundaries* and was not maintained internally within individual PLDs.

Fig. 2.9(b) shows the max, min, and average depth of downloaded pages at different stages of the crawl. Interestingly, pages at depth 3 were still being crawled even at the very end of the experiment, which suggests that the budget enforcer was



(a) histogram (b) time series

Figure 2.9: Crawl depth distribution.

not allowing enough pages from the domains adjacent to (and including) `tamu.edu` to pass through. Analysis of the data confirms this suspicion as only 30K pages out of 1.7M in `tamu.edu` were downloaded by that time. Another example is `blogspot.com` from which we obtained only 347K pages out of 362M reported by Google.

This indicates that IRLbot's rather conservative (i.e., linear) budget function requires careful examination and tuning. Results above suggest that significantly more aggressive, i.e., non-linear, budget allocation for highly ranked PLDs might be beneficial in future crawler designs. This is a topic of ongoing investigation.

2.8 Related Work

Traditional academic crawlers are mostly limited to 100M pages [17], [28], [42], [96], [114], [122], [142], while more recent efforts [37], [147] report approaching 1B pages, but without disclosing any implementation or crawling details. The most extensively documented research crawler in the industry is Mercator [64], [105], whose largest published experiment issued 1B connection requests and downloaded 784M responses, out of which 429M were HTML pages [20]. Long-term archiving projects have reported 3B pages in 5 years [67], 14B pages in 3 years [65], and 150B pages since 1997 [70].

The remaining crawlers are mostly commercial. The WebFountain project [58] at IBM at one point accumulated over 5.6B URLs [59], discovering 50M hosts [55] and 37B edges [43]. Microsoft was reported [116] to possess in 2006 a graph with 5B nodes and 370B links. In 2008, Google was indexing 30B pages [87] and seeing links to 1T unique URLs [6]. In late 2010, Yahoo disclosed crawling 150B pages and using graphs with 5T edges [110]. In 2012, Google was crawling 20B pages/day and keeping track of 30T unique URLs [123].

2.9 Conclusion

This chapter presented new IRLbot implementation details, proposed a novel methodology for documenting large-scale crawls, and used it to deliver a massive amount of previously undocumented information about the IRLbot experiment. We also derived a model for extrapolating the growth rate of unique nodes (e.g., pages, hosts, and PLDs) as a function of crawl size, confirming the colloquial notion that the space of URLs and hostnames is infinite, and estimated the number of remaining PLDs in larger IRLbot crawls. We finally proposed several methods for assessing Internet-wide crawl coverage, examined the budget function of IRLbot, and suggested avenues for improvement.

3. RANDOMIZED DATA STREAMS

3.1 Introduction

Large-scale distributed systems (e.g., Google, Facebook) operate on *streams* of key-value pairs that arise from disk/network-based processing of massive amounts of data. Due to the enormous size of input, streaming is the commonly used computational model, where the arriving items are scanned sequentially and processed one at a time. While MapReduce [41] is one application that by design employs streaming, other types of jobs can be modeled under the same umbrella. For example, caching can be viewed as computation on streams, where the frequency of duplicate items determines performance (i.e., hit rate). Large-scale graph-processing algorithms that output edges/nodes in bulk (e.g., BFS search) can also be reduced to streams, where performance may be determined by the size of the frontier (i.e., pending nodes), bias in the observed degree, and/or discovery rate of new vertices.

It is common to replace keys with their hashes and apply computation that outputs data in random order, either by design (e.g., reversing edges in graphs) or as byproduct of some previous computation (e.g., sorting by a different key in an earlier stage of MapReduce). This results in real workload consisting of *randomized* streams, in which keys are shuffled in some arbitrary order. Understanding statistical properties of these streams is an important area of research as it leads to better characterization of MapReduce, caching, graph exploration, and more general streaming. However, existing analysis is not just scattered across many fields [12], [13], [26], [40], [73], [80], [81], [89], [106], [144], but is also lacking in its ability to accurately model the stochastic properties of random streams.

In this work, we first formalize one-dimensional (1D) streams as discrete-time processes $\{Y_t\}_{t\geq 1}$, where each item Y_t observed at time t is unique (i.e., previously unseen) with some probability $p(t)$ and duplicate otherwise. Given the frequency distribution of keys, we first derive $p(t)$ and obtain the number of unique keys observed by time t. Then, we extend our modeling framework to two-dimensional (2D) streams $\{X_t, Y_t\}_{t\geq 1}$, where random variables X_t and Y_t are correlated due to the nature of the workload (e.g., adjacent nodes in a graph).

To demonstrate the usefulness of the derived results, we apply them to obtain the overall miss rate of Least Recently Used (LRU) cache in both simulated and real streams. Next, we analyze MapReduce computation, where we obtain the amount of data generated by each sorted run as a function of RAM size and consequently, the total I/O overhead of the reduce phase. This has recently emerged as an important problem [13], [80], [144], with no prior closed-form analysis. Finally, we apply our 2D stream models to characterize Breadth First Search (BFS) crawls on directed random graphs and develop a number of interesting results (e.g., degree distribution of seen/unseen/crawled nodes, frontier size), all functions of crawl time t. We also present comparative analysis between BFS and a number of other crawling methods using these results.

3.2 Literature Review

Although we examine a number of problems (i.e., LRU performance, MapReduce overhead and crawl characteristics) under one single framework (i.e., data streams), the existing literature on these problems is spread across multiple fields.

There has been much work [26], [40], [52], [73] on deriving the cache miss rate under various replacement policies (e.g., LRU, LFU, random, FIFO) and specific distributions of key frequency. For example, [73] provides an asymptotic analysis of LRU for Zipf and Weibull distributions. Another set of results [26] relates the to the miss rate of individual items. In contrast, our LRU model captures the overall miss rate across all available keys and further generalizes [26] using certain intermediate results. Furthermore, our model is not limited to any specific distribution of item frequency.

Since their introduction, both MapReduce [41] and its open-source implementation Hadoop [141] have received a great deal of academic and industry attention. Their performance is analyzed in [13], [80], [90], and [144]. The authors in [13], [80], [90] offer models for the size of sorted runs produced from MapReduce computation and those in [90] extend the models further to multi-pass merging. But they all assume a constant multiplicative factor that converts the size of input to the number of keys in the sorted runs, which requires training/prior knowledge about the data. Furthermore, this assumption requires the input-output relation to be linear, which is not true in general. In contrast, we propose exact models for the size of sorted runs and the total disk overhead, which rely only on statistical stream properties.

Crawl characteristics of BFS and its sampling bias have been studied extensively in the literature [12], [81], [89], [106]. A characterization of degree bias as a function of crawl fraction and a method for its correction are presented in [81]. This method's main limitation is that it considers only undirected random graphs. Furthermore, this approach requires a modified BFS where every edge is constructed at crawl time by selecting both ends randomly. Such modification fails to preserve source-destination edge pairing inherent in graph exploration. In contrast, our analysis deals with directed graphs and does not require any modification to BFS algorithm.

3.3 Randomized 1D Streams

We start by introducing our terminology, assumptions, and objectives. We then develop a stochastic model for the uniqueness probability $p(t)$ and the size of the unique set.

3.3.1 Terminology

Assume a collection V of n unique keys. Suppose $\mathcal{I}(v)$ for each $v \in V$ is the number of times v appears in the input stream. Using graph-theory terminology, we often call v a *node* and $\mathcal{I}(v)$ its *in-degree*. Define $T = \sum_{v \in V} \mathcal{I}(v)$ to be the length of the stream and assume that the stream is randomly shuffled. Then, realizations of the stream can be viewed as a 1-dimensional stochastic process $\{Y_1, \ldots, Y_T\}$, where Y_t is the random key in position $1 \leq t \leq T$. For simplicity of presentation, let random variable $\mathcal{I}$ have the same distribution as the in-degree of the system:

$$P(\mathcal{I} < x) = \frac{1}{n} \sum_{v \in V} \mathbf{1}_{\mathcal{I}(v) < x}. \tag{3.1}$$

As the stream is being processed, let $S_t = \bigcup_{i=1}^{t} \{Y_i\}$ be the set of keys *seen* by time t and suppose $U_t = V \setminus S_t$ contains the *unseen* keys at t. Then define $p(t) = P(Y_t \in U_{t-1})$ to be the probability that key Y_t has not been seen before time t, which is the central metric for assessing performance of streaming algorithms. This includes crawl characterization, cache analysis, and MapReduce computation. To develop a tractable model for the uniqueness probability $p(t)$, we must place certain constrains on the appearance of keys in the stream, as discussed next.

3.3.2 Stream Residuals

We start by defining a special class of streams that commonly occur in practice (e.g., MapReduce data processing, DNS queries, workload arrival to web servers), where the keys are hashed to random values before being streamed from storage into the program and there is no (or little) correlation between adjacent items. Define

$Z(v,t) = \mathbf{1}_{\{Y_t=v\}}$ to be an indicator of v being in position t, i.e.,

$$Z(v,t) = \begin{cases} 1 & Y_t = v \\ 0 & \text{otherwise} \end{cases}, \tag{3.2}$$

and consider the following.

Definition 1. *A stream is called to possess* Uniform Residuals *(UR) if the probability of seeing v at time t is proportional to the number of remaining copies of v in the stream:*

$$P(Y_t = v | Y_{t-1}, \ldots, Y_1) = \frac{\mathcal{I}(v) - \sum_{\tau=1}^{t-1} Z(v,\tau)}{T - t + 1}. \tag{3.3}$$

Note that $\{Y_t\}_{t\geq 1}$ is not a Markov chain since Y_t at every step depends on the entire history of the process. To understand (3.3) better, define $H(v,t)$ to be the number of times v is seen in $[1,t]$:

$$H(v,t) = \sum_{\tau=1}^{t} Z(v,\tau), \qquad t = 1, 2, \ldots, T, \tag{3.4}$$

where $H(v,0) = 0$ and $H(v,T) = \mathcal{I}(v)$. While $H(v,t)$ is a sum of Bernoulli random variables, it is tempting to speculate that it is binomial; however, this is false since each $Z(v,t)$ depends on prior values $Z(v,1), \ldots, Z(v,t-1)$ and:

$$\sum_{\tau=1}^{t} Z(v,\tau) \leq \mathcal{I}(v), \tag{3.5}$$

which makes set $\{Z(v,t)\}_t$ non-iid.

Now, letting $R(v,t) = \mathcal{I}(v) - H(v,t)$ be the *residual degree* of v and unconditioning (3.3) by taking expectation over all sample paths, we get a more intuitive definition of UR streams:

$$P(Y_t = v) = \frac{\mathcal{I}(v) - E[H(v, t-1)]}{T - t + 1} = \frac{E[R(v, t-1)]}{T - t + 1}, \tag{3.6}$$

which shows that the probability to encounter v is proportional to its expected number of residual copies in the interval $[t, T]$. The rest of the analysis assumes that streams under consideration exhibit the UR property.

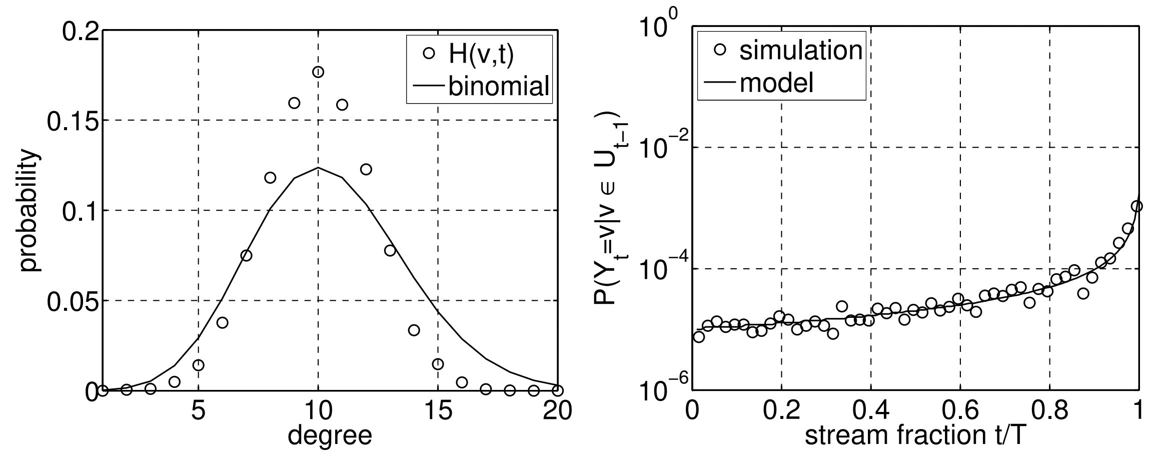
One interesting conclusion emerges from (3.6). Observe that the residual degree of an unseen key v always equals its total degree $\mathcal{I}(v)$. Therefore, the probability of discovering such nodes is:

$$P(Y_t = v | v \in U_{t-1}) = \frac{\mathcal{I}(v)}{T - t + 1}. \tag{3.7}$$

3.3.3 A Few Words on Simulations

Throughout this section, we simulate MapReduce streams by first establishing sequence $\{\mathcal{I}(v)\}_{v \in V}$ under a given distribution. We use Zipf $\mathcal{I}$ with different shapes α as an approximation to in-degree of the web [19] and binomial as a model of degree in $G(n, p)$ random graphs. Then, we repeat each key v exactly $\mathcal{I}(v)$ times, assign it a random hash based on its position in the stream, and then sort the result by the hash, which gives us one realization of the stream. Changing the seed to the hash function, we execute this process a number of times to generate many sample paths of the system. In the next section, we use streams produced by crawls over random graphs and focus on a single sample-path. Finally, the last section of the paper employs real (non-simulated) input.

To put this discussion to use, Fig. 3.1(a) illustrates the distribution of $H(v, t)$ using a node with $\mathcal{I}(v) = 20$ and $t/T = 0.5$. Notice that the binomial fit has a much



(a) distribution of $H(v,t)$ (b) model (3.7)

Figure 3.1: Simulation with Zipf $\mathcal{I}$ with $\alpha = 1.2, E[\mathcal{I}] = 10, n = 10\text{K}$.

higher variance than the true distribution, which is a consequence of the dependency in (3.5). Fig. 3.1(b) verifies that (3.7) is indeed applicable to randomly sorted streams, where we track a random node with $\mathcal{I}(v) = 1$.

3.3.4 Single Node

Our examination begins with properties of a fixed node v, which we will need later for the more advanced results. Define *stream fraction* $\epsilon_t = t/T$ and consider the following result.

Lemma 1. *Assume t and x are integers such that $0 \le x \le t \le T$. Then, the following holds:*

$$\prod_{\tau=0}^{t-1}\left(1 - \frac{x}{T-\tau}\right) = \prod_{\tau=0}^{x-1}\left(1 - \frac{t}{T-\tau}\right) \approx (1-\epsilon_t)^x. \tag{3.8}$$

Proof. Expanding the left hand side of (3.8):

$$\prod_{\tau=0}^{t-1}\left(1-\frac{x}{T-\tau}\right)=\frac{\prod_{\tau=0}^{t-1}(T-x-\tau)}{\prod_{\tau=0}^{t-1}(T-\tau)}=\frac{\prod_{\tau=x}^{x+t-1}(T-\tau)}{\prod_{\tau=0}^{t-1}(T-\tau)}.$$

Since $t \geq x$, there are $t-x$ terms in each product that cancel. We thus get:

$$\begin{aligned}\prod_{\tau=0}^{t-1}\left(1-\frac{x}{T-\tau}\right)&=\frac{\prod_{\tau=0}^{x-1}(T-\tau-t)}{\prod_{\tau=0}^{x-1}(T-\tau)}=\prod_{\tau=0}^{x-1}\frac{T-\tau-t}{T-\tau}\\&=\prod_{\tau=0}^{x-1}\left(1-\frac{t}{T-\tau}\right).\end{aligned}\tag{3.9}$$

For $1 \leq x \ll T$ and $x \leq t$, the right-hand side of (3.9) is approximately:

$$\prod_{\tau=0}^{t-1}\left(1-\frac{x}{T-\tau}\right)\approx\left(1-\frac{t}{T}\right)^{x},\tag{3.10}$$

which is the desired result. □

In our application of (3.8), t usually represents time and x the random degree of a node, where $x \ll T$ holds. Under these conditions, Lemma 1 is important in its ability to create a simple, yet very accurate, approximation to a product of millions (if not billions) of terms. In fact, when $x = 1$, the $(1-\epsilon_t)^x$ term in (3.8) is exact. Leveraging this observation, we obtain the following result.

Theorem 1. *The expected residual degree of v at time t is:*

$$E[R(v,t)]=(1-\epsilon_t)\mathcal{I}(v).\tag{3.11}$$

Proof. Recalling that $Z(v,t)=1$ if Y_t hits v, notice that the residual degree at t is given by:

$$R(v,t) = R(v,t-1) - Z(v,t). \tag{3.12}$$

Using $E[Z(v,t)] = P(Y_t = v)$ and (3.6):

$$\begin{aligned} E[R(v,t)] &= E[R(v,t-1)] - \frac{E[R(v,t-1)]}{T-t+1} \\ &= E[R(v,t-1)]\left(1 - \frac{1}{T-t+1}\right). \end{aligned} \tag{3.13}$$

Expanding (3.13) down to $t = 1$ and noting that $R(v,0) = \mathcal{I}(v)$, we get:

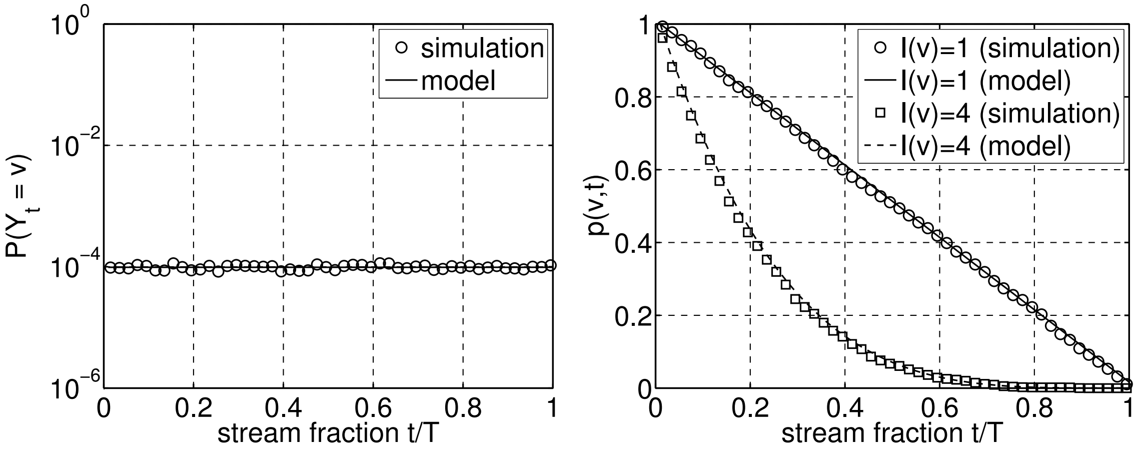
$$E[R(v,t)] = \mathcal{I}(v)\prod_{\tau=0}^{t-1}\left(1 - \frac{1}{T-\tau}\right), \tag{3.14}$$

which after invoking Lemma 1 with $x = 1$ produces (3.11). $\square$

Substituting (3.11) into (3.6), Theorem 1 shows that the probability of v being hit at t is time-invariant:

$$P(Y_t = v) = E[Z(v,t)] = \frac{\mathcal{I}(v)}{T}, \tag{3.15}$$

which is sometimes called the *Independent Reference Model* [97]. This result is confirmed in Fig. 3.2(a), which shows that $P(Y_t = v)$ indeed stays constant throughout the stream and its value is a function of $\mathcal{I}(v)$, but not the time. Thus, nodes with high degree are more likely to be seen irrespective of which portion of the stream is being examined. Interestingly, (3.15) shows that the distribution of $Z(v,t)$ does not depend on t and thus $\{Z(v,t)\}_t$ is a set of *identically-distributed* random variables; however, from (3.5), we know they are dependent.



(a) model (3.15) (b) model (3.16)

Figure 3.2: Verification under Zipf $\mathcal{I}$ with $\alpha = 1.2, E[\mathcal{I}] = 10, n = 10\text{K}$.

The next result derives the likelihood for a given node v to remain unseen throughout the entire interval $[1, t]$.

Theorem 2. *The probability that v is still unseen at time t:*

$$p(v,t) = P(v \in U_t) \approx (1-\epsilon_t)^{\mathcal{I}(v)}. \tag{3.16}$$

Proof. For a node to stay unseen at t, it must be unseen at $t-1$ and also not hit at t. Therefore, using (3.7):

$$\begin{aligned} p(v,t) &= P(Y_t \neq v, v \in U_{t-1}) \\ &= P(Y_t \neq v | v \in U_{t-1}) P(v \in U_{t-1}) \\ &= \left(1 - \frac{\mathcal{I}(v)}{T-t+1}\right) p(v, t-1). \end{aligned} \tag{3.17}$$

After expanding the recurrence in (3.17) to $t = 1$, we get:

$$p(v,t) = p(v,0)\prod_{\tau=0}^{t-1}\left(1 - \frac{\mathcal{I}(v)}{T-\tau}\right), \tag{3.18}$$

which produces (3.16) after invoking Lemma 1 and using $p(v,0) = 1$. $\square$

To verify (3.16), we use a Zipf stream and select two random keys: one with degree 1 and the other with degree 4. Then, we observe the corresponding values of $p(v,t)$ in simulations and compare them with model (3.16) in Fig. 3.2(b), which shows that the model is accurate. As expected from (3.16), $p(v,t)$ is linear for the degree-1 node and nonlinear for the degree-4 node.

Armed with Theorem 2, we can solve the opposite problem from that in (3.7) to produce a useful result that will help us later establish how the degree of newly discovered nodes varies with time.

Theorem 3. *Conditioned on the fact that node v is hit at time t, the probability that v was unseen at $t-1$ is:*

$$P(v \in U_{t-1}|Y_t = v) \approx (1-\epsilon_t)^{\mathcal{I}(v)-1}. \tag{3.19}$$

Proof. Using Bayes' theorem:

$$P(v \in U_{t-1}|Y_t = v) = \frac{P(Y_t = v|v \in U_{t-1})p(v,t)}{P(Y_t = v)}. \tag{3.20}$$

Applying substitutions from (3.7), (3.15), and (3.16):

$$P(v \in U_{t-1}|Y_t = v) \approx \frac{T}{T-t+1}(1-\epsilon_t)^{\mathcal{I}(v)}. \tag{3.21}$$

Observing that $T/(T-t+1) \approx (1-\epsilon_t)^{-1}$, we immediately get (3.19). $\square$

3.3.5 Uniqueness Probability

We now are ready to obtain the main result of this section.

Theorem 4. *The probability that the t-th key in the stream Y_t refers to a previously-unseen node is:*

$$p(t) = P(Y_t \in U_{t-1}) \approx \frac{E[\mathcal{I} \cdot (1-\epsilon_t)^{\mathcal{I}-1}]}{E[\mathcal{I}]}. \tag{3.22}$$

Proof. Partitioning the probability space into n mutually exclusive events, we get:

$$\begin{aligned} p(t) &= \sum_{v \in V} P(Y_t \in U_{t-1}, Y_t = v) \\ &= \sum_{v \in V} P(Y_t = v | v \in U_{t-1}) P(v \in U_{t-1}). \end{aligned} \tag{3.23}$$

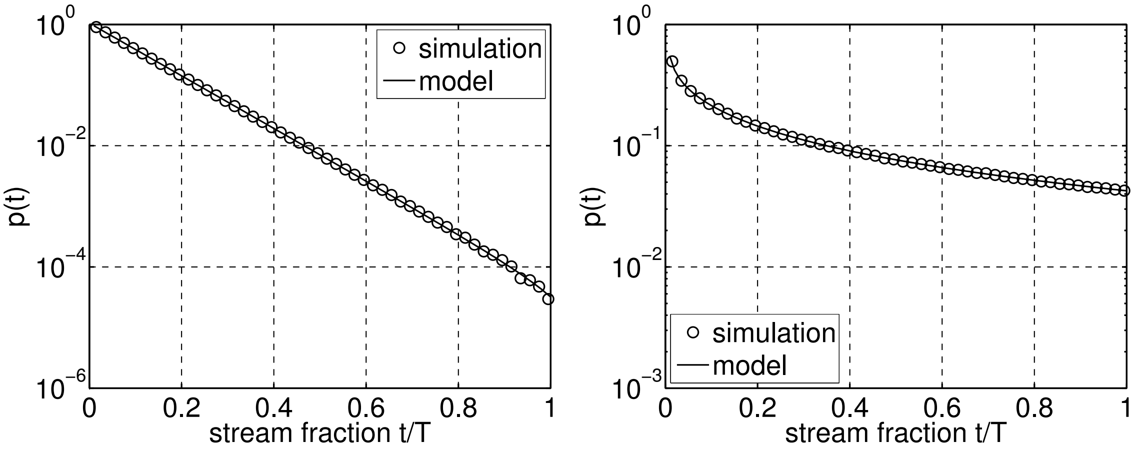
Using (3.7) and (3.16) in (3.23), we get:

$$\begin{aligned} p(t) &\approx \sum_{v \in V} \frac{\mathcal{I}(v)}{T-t+1} (1-\epsilon_t)^{\mathcal{I}(v)} \\ &\approx \frac{1}{T} \sum_{v \in V} \mathcal{I}(v) (1-\epsilon_t)^{\mathcal{I}(v)-1}, \end{aligned} \tag{3.24}$$

which leads to (3.22) after using $\sum_{v \in V} f(\mathcal{I}(v)) = nE[f(\mathcal{I})]$ (by definition of expectation), $T = nE[\mathcal{I}]$, and $(t-1)/T \approx \epsilon_t$. $\square$

To perform a self-check, we analyze $p(t)$ for two special cases. First, assume constant degree $\mathcal{I}(v) = d \geq 1$ for all $v \in V$. In this case, (3.22) simplifies to:

$$p(t) \approx (1-\epsilon_t)^{d-1}, \tag{3.25}$$



(a) binomial $\mathcal{I}$ (b) Zipf $\mathcal{I}$ ($\alpha = 1.2$)

Figure 3.3: Verification of (3.22) under $E[\mathcal{I}] = 10, n = 10\text{K}$.

which is the probability that none of the Y_t's remaining $d-1$ appearances have fallen into the interval $[1, t)$. Second, consider a stream of *unique* items, i.e., $\mathcal{I}(v) = 1$ for all $v \in V$. In this scenario, (3.22) produces the correct $p(t) = 1$ for all t.

We compare Theorem 4 against simulations in Fig. 3.3. It is clear that in both cases the model is accurate in the entire range $[1, T]$. Also observe that $p(t)$ in the Zipf case (b) demonstrates a heavier tail, which is a consequence of many low-degree nodes that stay undiscovered until the very end. On the other hand, the binomial curve quickly finds the majority of the nodes and $p(t)$ decays to zero exponentially fast. To explain this intuition better, we next analyze $p(t)$ as $t \to T$. Rewriting (3.22) by splitting between degree-1 nodes and all others:

$$p(t) \approx \frac{P(\mathcal{I} = 1) + E[\mathcal{I} \cdot (1 - \epsilon_t)^{\mathcal{I}-1} | \mathcal{I} > 1] P(\mathcal{I} > 1)}{E[\mathcal{I}]},$$

where the second term decays to 0 as $\epsilon_t \to 1$ and we get:

$$\lim_{t \to T} p(t) \approx \frac{P(\mathcal{I} = 1)}{E[\mathcal{I}]}, \tag{3.26}$$

which means that the last point in the curve is solely determined by the number of degree-1 nodes in the stream. Since $E[\mathcal{I}] = 10$ and nearly 40% of the nodes in the Zipf stream are degree-1, we immediately obtain $p(T) = 0.04$, which agrees with Fig. 3.3(b). On the other hand, the binomial stream contains fewer than 0.1% unique nodes and thus exhibits $p(T) < 10^{-4}$. Our experiments with a number of other in-degree distributions (e.g., uniform) and graph structures (e.g., hypercube) provide similarly accurate $p(t)$ results.

3.3.6 Set of Unique Nodes

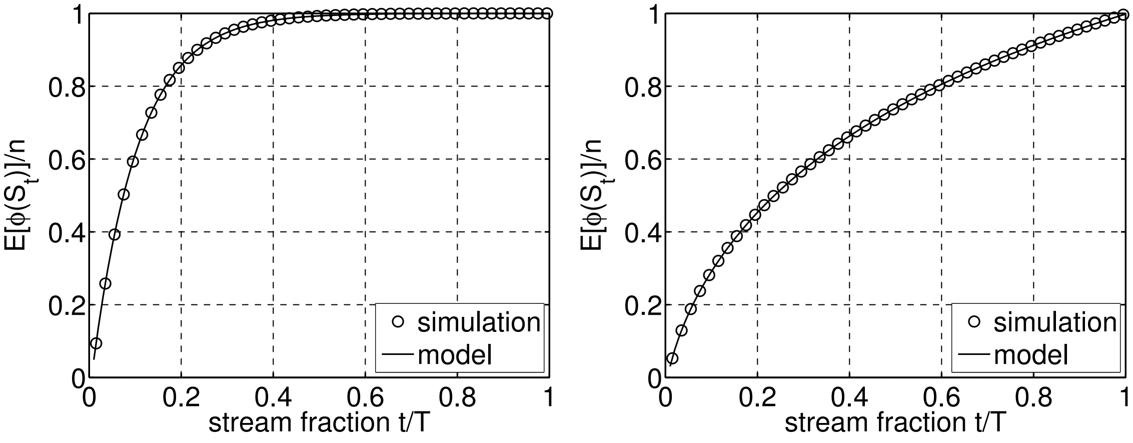
Modeling the size of S_t allows characterization of cache performance and estimation of hash-table sizes required to store unique items as the stream is being processed. We come back to these issues later. In the meantime, we define $\phi(A)$ to be the size of set A and present the following result.

Theorem 5. *The expected size of the seen set at time t is:*

$$E[\phi(S_t)] \approx nE\left[1 - (1 - \epsilon_t)^{\mathcal{I}}\right]. \tag{3.27}$$

Proof. Since the expected size of the unseen set is given by a summation of probabilities that each individual node is still unseen at t, we have:

$$\begin{aligned} E[\phi(U_t)] = \sum_{v \in V} P(v \in U_t) \approx \sum_{v \in V} (1 - \epsilon_t)^{\mathcal{I}(v)} \\ \approx nE[(1 - \epsilon_t)^{\mathcal{I}}]. \end{aligned} \tag{3.28}$$



(a) binomial $\mathcal{I}$ (b) Zipf $\mathcal{I}$ ($\alpha = 1.2$)

Figure 3.4: Verification of (3.27) under $E[\mathcal{I}] = 10, n = 10$K.

Recalling that $E[\phi(S_t)] = n - E[\phi(U_t)]$, we get (3.27). □

We compare this result against simulations in Fig. 3.4 and observe that it is accurate. Note that the seen set grows more rapidly in the binomial case (a). For example, it accumulates 84% of the keys in the first 20% of the stream, while Zipf in subfigure (b) discovers only 45% of the nodes by that time. The reason is that $\phi(S_t)$ is essentially the integral of $p(t)$, which in the binomial stream grows much quicker in the beginning. This can be seen by contrasting the two curves in Fig. 3.3 in the range $\epsilon_t \in [0, 0.2]$.

3.4 Randomized 2D Streams

This section formalizes 2D streams and uses them for analyzing graph traversal algorithms (e.g., BFS, DFS). We start with terminology and assumptions, which are followed by the main results.

3.4.1 Terminology and Assumptions

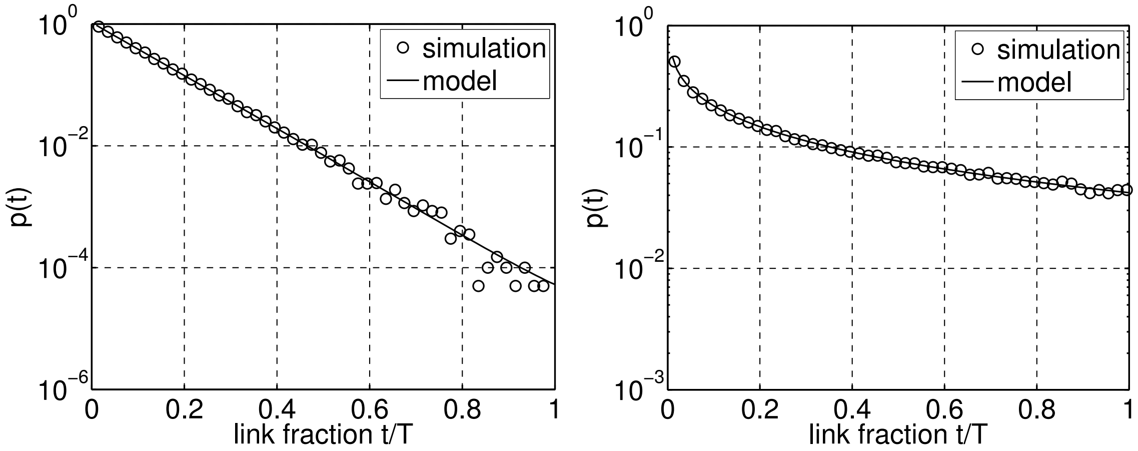
We start by describing a family of graphs where assumption (3.3) holds for common search algorithms. Consider a simple directed random graph $G(V, E)$, where V is the set of n nodes, E the set of T edges. To ensure uniform residuals, we generate the in and out-degree sequences $\{\mathcal{I}(v)\}_{v\in V}$ and $\{\mathcal{O}(v)\}_{v\in V}$ according to the corresponding distributions such that:

$$T = |E| = \sum_{v\in V} \mathcal{I}(v) = \sum_{v\in V} \mathcal{O}(v), \tag{3.29}$$

and then use so-called *configuration models* [100] to create random edges in the system. We describe the process next.

We use time-varying multi-sets $\mathcal{D}_{in}(t)$ and $\mathcal{D}_{out}(t)$ in the construction process. For each $v \in V$, we insert $\mathcal{I}(v)$ copies of the node into initial set $\mathcal{D}_{in}(0)$ and $\mathcal{O}(v)$ instances of it into $\mathcal{D}_{out}(0)$. At every step $1 \leq t \leq T$, a directed edge is formed between randomly selected nodes $x \in \mathcal{D}_{out}(t-1)$ and $y \in \mathcal{D}_{in}(t-1)$, both of which are then removed from the corresponding sets. Note that the constructed graph G is a random instance among all possible graphs that can be constructed from a given degree sequence/distribution.

Next, we describe the crawl process. We consider algorithms that crawl/visit each node exactly once. Suppose at time t, node u is removed from the frontier F_t according to some exploration strategy. Then all $\mathcal{O}(u)$ out-edges of node u, i.e., $(u, v_1), (u, v_2), \ldots, (u, v_{\mathcal{O}(u)})$, are processed at *consecutive* time steps $t, t+1 \ldots t + \mathcal{O}(u) - 1$, which creates correlation between the visited edges (i.e., they all have the same source node). If neighbor v_i is previously unseen, it is appended to the frontier F_{t+i-1}; otherwise, it is discarded. The crawler selects the next node from the frontier



(a) binomial $\mathcal{I}$ (b) Zipf $\mathcal{I}$ ($\alpha = 1.5$)

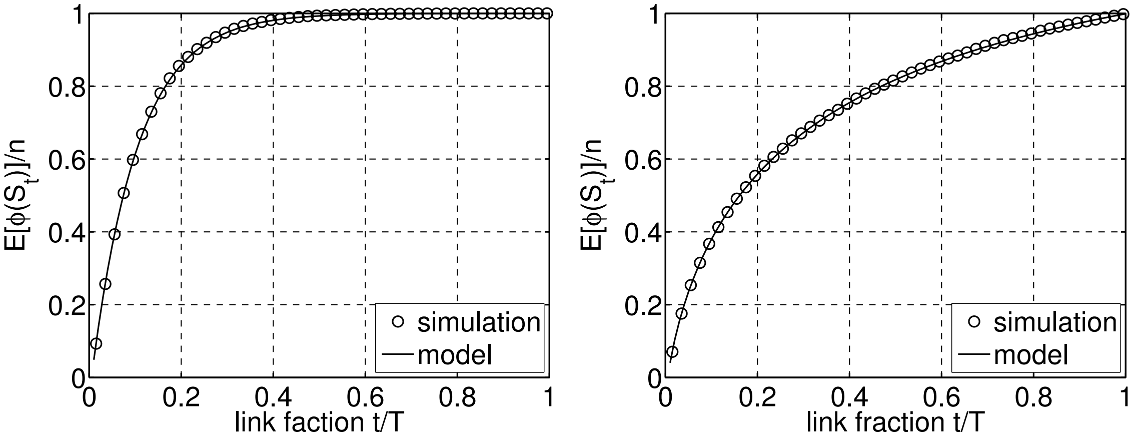
Figure 3.5: Verification of $p(t)$ in BFS crawls with $E[\mathcal{I}] = 10, n = 10\text{K}$.

at $t + \mathcal{O}(u)$ and the process repeats.

We can now define the stream of edges produced by graph crawling to be a 2D discrete-time stochastic process $\{(X_t, Y_t)\}_{t=1}^T$ on E, where X_t is the crawled node and Y_t the destination node (i.e., one of X_t's out-neighbors), both random variables. Randomness arises due to the stochastic nature of G, where each sample path $\{X_t, Y_t\}_{t=1}^T$ operates on a different instance of the graph. Defining $C_t = \bigcup_{i=1}^t \{X_i\}$ to be the set of already-crawled nodes by time t, frontier F_t can be expressed as $S_t \setminus C_t$. Before the crawl starts, both sets are empty, i.e., $C_0 = F_0 = \emptyset$. The choice of initial node X_0 does not affect the analysis and is omitted from the discussion.

3.4.2 Degree of the Seen Set

We start by examining the stream $\{Y_t\}_{t=1}^T$ of destination nodes produced by a crawler and verify that our earlier results are in fact applicable in this situation. Figs. 3.5-3.6 compare respectively models (3.22), (3.27) against BFS simulations on graphs with binomial and Zipf degree, where each graph contains the same number of nodes n and edges T. The figures show that both models are still very accurate.



(a) binomial $\mathcal{I}$ (b) Zipf $\mathcal{I}$ ($\alpha = 1.5$)

Figure 3.6: Verification of $E[\phi(S_t)]$ in BFS crawls with $E[\mathcal{I}] = 10, n = 10\text{K}$.

Armed with this confirmation, we draw our attention to inferring more advanced properties of the crawl process.

Let $\mathcal{I}(S_t)$ and $\mathcal{O}(S_t)$ denote respectively the number of in/out edges incident to the nodes in the seen set S_t, i.e.,

$$\mathcal{I}(S_t) = \sum_{v \in S_t} \mathcal{I}(v), \quad \mathcal{O}(S_t) = \sum_{v \in S_t} \mathcal{O}(v). \tag{3.30}$$

Then, we have the following result that helps understand the properties of the seen nodes and their average degree.

Theorem 6. *The expected number of in/out edges incident to the nodes in the seen set is respectively:*

$$E[\mathcal{I}(S_t)] \approx nE[\mathcal{I} \cdot (1 - (1 - \epsilon_t)^{\mathcal{I}})], \tag{3.31}$$

$$E[\mathcal{O}(S_t)] \approx nE[\mathcal{O} \cdot (1 - (1 - \epsilon_t)^{\mathcal{I}})], \tag{3.32}$$

where $\mathcal{O}$ is the random out-degree of a node in the system.

Proof. The total in-degree of the nodes in the seen set is:

$$\begin{aligned} E[\mathcal{I}(S_t)] &= \sum_{v \in V} \mathcal{I}(v) P(v \in S_t) \\ &\approx \sum_{v \in V} \mathcal{I}(v)[1 - (1 - \epsilon_t)^{\mathcal{I}(v)}], \\ &= nE[\mathcal{I}(1 - (1 - \epsilon_t)^{\mathcal{I}})], \end{aligned} \tag{3.33}$$

The out-degree case is derived similarly, which we omit for brevity. □

Interestingly, (3.31) can also be expressed as $p(t)T$, which shows that the in-degree of the seen set follows the same curve $p(t)$ scaled by the total number of edges T. The other model (3.32) captures correlation between in/out-degree in the system. If the two variables $\mathcal{I}$ and $\mathcal{O}$ are independent, the result expands to $nE[\mathcal{O}]E[1-(1-\epsilon_t)^{\mathcal{I}}] = E[\mathcal{O}]E[\phi(S_t)]$. This makes sense as it multiplies the size of S_t by the average out-degree in the graph. However, when the two degree variables are dependent, the formula no longer admits a simple expansion.

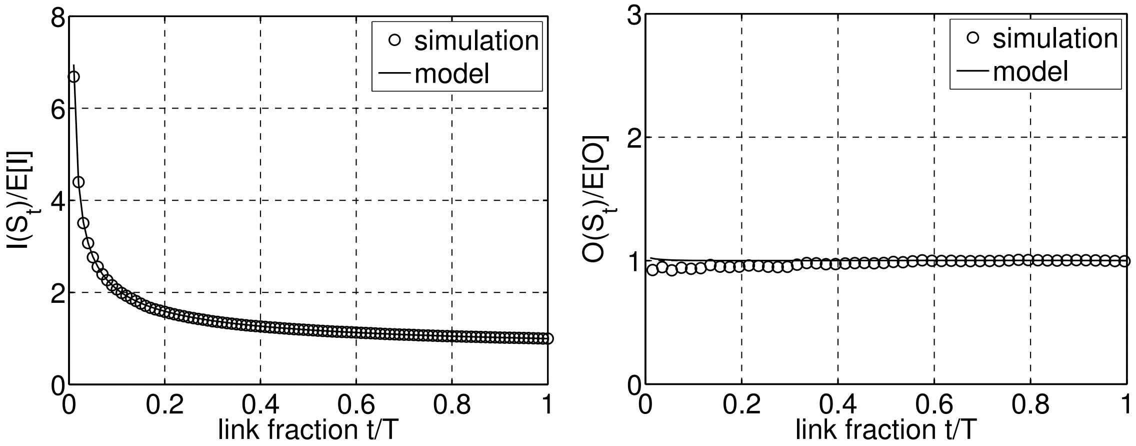
With this general knowledge, we can quantify the degree bias of the nodes placed into the seen set. Define the average in and out-degree of the nodes in S_t respectively as:

$$\bar{\mathcal{I}}(S_t) = \frac{E[\mathcal{I}(S_t)]}{E[\phi(S_t)]}, \quad \bar{\mathcal{O}}(S_t) = \frac{E[\mathcal{I}(S_t)]}{E[\phi(S_t)]}, \tag{3.34}$$

and consider the following result.

Theorem 7. *The average in/out-degree of the nodes in the seen set is:*

$$\bar{\mathcal{I}}(S_t) \approx \frac{E[\mathcal{I} \cdot (1 - (1 - \epsilon_t)^{\mathcal{I}})]}{1 - E[(1 - \epsilon_t)^{\mathcal{I}}]}, \tag{3.35}$$



(a) in-degree model (3.35) (b) out-degree model (3.36)

Figure 3.7: Verification with Zipf $\mathcal{I}$ under $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10\text{K}$.

$$\bar{\mathcal{O}}(S_t) \approx \frac{E[\mathcal{O} \cdot (1 - (1 - \epsilon_t)^{\mathcal{I}})]}{1 - E[(1 - \epsilon_t)^{\mathcal{I}}]}. \tag{3.36}$$

Proof. Observe that we can express the first metric as the total in-degree of the seen set normalized by the size of S_t:

$$E[\bar{\mathcal{I}}(S_t)] \approx \frac{E[\mathcal{I}(S_t)]}{E[\phi(S_t)]}, \tag{3.37}$$

which leads to (3.35) after applying (3.27) and (3.31). The proof of (3.36) is similar. □

In Fig. 3.7(a), we compare (3.35) against BFS simulations. The plot shows that the average in-degree in the seen-set starts at a much higher (i.e., 6 times in this example) value than $E[\mathcal{I}]$. Consistent with previous findings about BFS bias [12], [89], [106], the model confirms that BFS finds nodes with high in-degree earlier during the crawl. It also stochastically quantifies the amount of bias, which has not been shown before in the literature. To perform a self-check, notice that when $\mathcal{I}$

and $\mathcal{O}$ are independent, (3.36) simplifies to the average out-degree $E[\mathcal{O}]$, which is confirmed by Fig. 3.7(b).

3.4.3 Destination Nodes

Another related property is the degree distribution of destination nodes Y_t of the discovered edges as the crawl progresses.

Theorem 8. *The in-degree distribution of Y_t is given by:*

$$P(\mathcal{I}(Y_t) = k) = \frac{kP(\mathcal{I} = k)}{E[\mathcal{I}]}. \tag{3.38}$$

Proof. Using conditional probability:

$$\begin{aligned} P(\mathcal{I}(Y_t) = k) &= \sum_{v \in V} P(\mathcal{I}(Y_t) = k | Y_t = v) P(Y_t = v) \\ &= \sum_{v \in V} \mathbf{1}_{\mathcal{I}(v)=k} \frac{\mathcal{I}(v)}{T} = \frac{knP(\mathcal{I} = k)}{T}, \end{aligned} \tag{3.39}$$

which produces (3.38) after replacing $T = nE[\mathcal{I}]$. □

Note that (3.38) has been known in the literature [94] as the degree distribution of random walks on undirected graphs. The main benefit of this relationship is that bias-correction methods for random walks [81] can be applied to BFS in our scenarios. Specifically, using an observation of Y_t for $t \in [1, m]$, the following is an unbiased estimator of $P(\mathcal{I} = k)$:

$$\frac{\sum_{t=1}^{m} \mathbf{1}_{\mathcal{I}(Y_t)=k}}{k \sum_{t=1}^{m} 1/\mathcal{I}(Y_t)}. \tag{3.40}$$

Next we derive the expected degree of Y_t's.

Theorem 9. *The average in/out-degree of Y_t is independent of time and equals:*

$$E[\mathcal{I}(Y_t)] = \frac{E[\mathcal{I}^2]}{E[\mathcal{I}]}, \qquad E[\mathcal{O}(Y_t)] = \frac{E[\mathcal{I}\mathcal{O}]}{E[\mathcal{I}]}. \tag{3.41}$$

Proof. From the definition of expectation:

$$\begin{aligned} E[\mathcal{I}(Y_t)] &= \sum_{v\in V} E[\mathcal{I}(Y_t)|Y_t = v]P(Y_t = v) \\ &= \sum_{v\in V} \mathcal{I}(v)P(Y_t = v) = \sum_{v\in V} \mathcal{I}(v)\frac{\mathcal{I}(v)}{T} = \frac{E[\mathcal{I}^2]}{E[I]}. \end{aligned} \tag{3.42}$$

The case of $E[\mathcal{O}(Y_t)]$ is handled similarly. □

We verify the in-degree model (3.41) against simulations in Fig. 3.8(a). Since we normalize both the model and the observed values by $E[\mathcal{I}^2]/E[I]$, the simulation staying constant at 1 indicates that the model is accurate.

The next lemma will become useful shortly in the analysis of nodes being added to the seen set at every time step t.

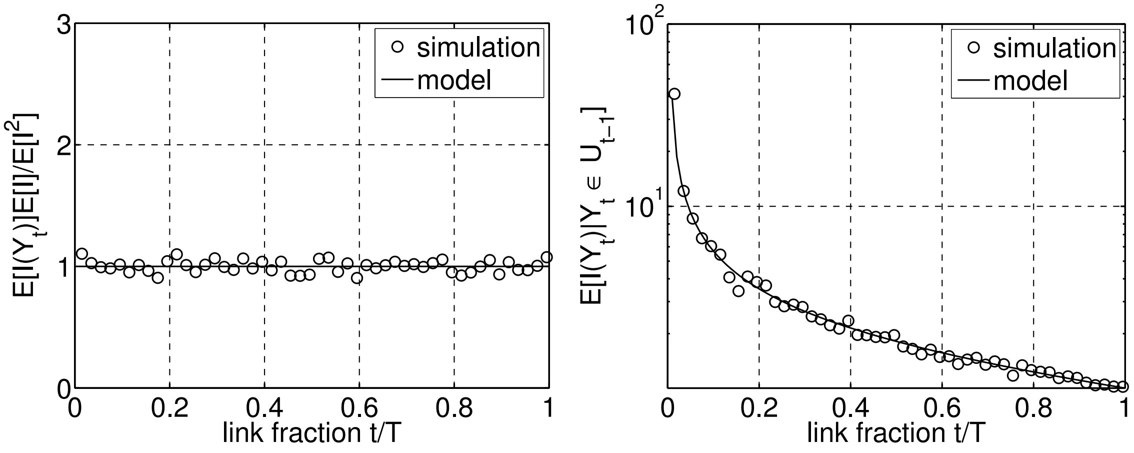
Lemma 2. *The probability that the discovered node at t is equal to v, conditioned on its being unseen, is:*

$$P(Y_t = v|Y_t \in U_{t-1}) \approx \frac{\mathcal{I}(v)(1-\epsilon_t)^{\mathcal{I}(v)-1}}{nE[\mathcal{I}(1-\epsilon_t)^{\mathcal{I}-1}]}. \tag{3.43}$$

Proof. Applying Bayes' rule and recalling (3.15):

$$\begin{aligned} P(Y_t = v|Y_t \in U_{t-1}) &= \frac{P(Y_t \in U_{t-1}|Y_t = v)P(Y_t = v)}{P(Y_t \in U_{t-1})} \\ &= \frac{P(v \in U_{t-1}|Y_t = v)I(v)}{p(t)T}. \end{aligned} \tag{3.44}$$

Applying (3.19), we get (3.43). □



(a) in-degree of Y_t (normalized) (b) in-degree of unseen Y_t

Figure 3.8: Verification of models (3.41) and (3.45) with Zipf $\mathcal{I}$ and $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10\text{K}$.

Now we are ready to derive the expected degree of the nodes that are moved from U_t into S_t as the crawl progresses. These are the nodes discovered for the first time at t.

Theorem 10. *The expected in/out-degree of the discovered node at* t*, conditioned on its being unseen, is given by:*

$$E[\mathcal{I}(Y_t)|Y_t \in U_{t-1}] = \frac{E[\mathcal{I}^2 \cdot (1-\epsilon_t)^{\mathcal{I}-1}]}{E[\mathcal{I}(1-\epsilon_t)^{\mathcal{I}-1}]}, \tag{3.45}$$

$$E[\mathcal{O}(Y_t)|Y_t \in U_{t-1}] = \frac{E[\mathcal{I}\mathcal{O} \cdot (1-\epsilon_t)^{\mathcal{I}-1}]}{E[\mathcal{I}(1-\epsilon_t)^{\mathcal{I}-1}]}. \tag{3.46}$$

Proof. We know that

$$E[\mathcal{I}(Y_t)|Y_t \in U_{t-1}] = \sum_{v \in V} \mathcal{I}(v) P(Y_t = v | Y_t \in U_{t-1})),$$

which produces (3.45) after applying (3.43). The proof of (3.46) is similar. □

We verify the in-degree model (3.45) against simulations in Fig. 3.8(b), which confirms the model. Interestingly, the expected degree of nodes added to S_t also starts at $E[\mathcal{I}^2]/E[\mathcal{I}]$ for $\epsilon_t = 0$ and then drops to $z = \min_{v\in V}\{\mathcal{I}(v)\}$ as $\epsilon_t \to 1$. To show the latter, we can apply L'Hôpital's rule to the ratio in (3.45) since both the numerator and denominator tend to zero. Thus, after some number of differentiations:

$$\lim_{\epsilon\to 1} \frac{\sum_{v\in V} \mathcal{I}(v)^2(1-\epsilon_1)^{\mathcal{I}(v)-1}}{\sum_{v\in V} \mathcal{I}(v)(1-\epsilon_1)^{\mathcal{I}(v)-1}} = \frac{z!zP(\mathcal{I}=z)}{z!P(\mathcal{I}=z)} = z. \tag{3.47}$$

However, unlike Fig. 3.7(a), which also begins at $E[\mathcal{I}^2]/E[\mathcal{I}]$, the decay rate of (3.45) is much faster.

In summary, the models in this section characterize a crawl process by quantifying various properties of Y_t where the current edge (X_t, Y_t) points to. We characterize the seen set and mathematically show how it is related to entire set of node V in the graph. The derived degree properties of Y_t are useful for forecasting the crawl experience throughout the process.

3.5 Applications

In this section, we examine a number of problems that deal with streams, where $p(t)$ is useful for measuring various quantities of interest. We first use it to analyze LRU cache performance, both on simulated and real streams. After that, we derive a stochastic model for the volume of data (also called *disk spill*) produced from MapReduce computation. Finally, we study the properties of X_t's in edge stream and obtain a number of metrics that apply to crawling web graphs. Note that, the first two (LRU cache and MapReduce) work with 1D streams, while the last one is for 2D streams.

In addition to simulated workloads, we use two real-world data sets for the experiments in this section. The first one is the host-level out-graph produced by IRLbot

[86], while the second one is a URL out-graph of WebBase [130], both dating back to June 2007. The former graph contains 641M unique nodes and 6.8B links, standing at a hefty 60 GB. The second contains 635M unique nodes and 4.2B links. It is smaller at 35 GB, but still larger than RAM size of our servers, which requires disk-based streaming and batch-mode processing.

3.5.1 Least Recently Used Cache

Here we briefly discuss the design of a Least Recently Used (LRU) cache and derive and accurate model for its hit/miss rate under a finite stream of input data. In addition, we also demonstrate how the existing well-known models do not translate directly to such finite samples, making our model the right choice in this case.

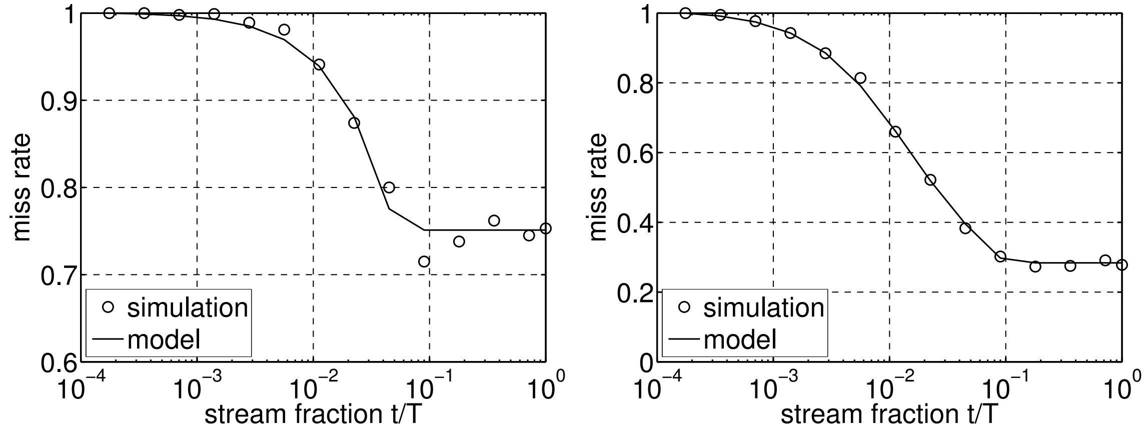
In an LRU cache, the items are kept in a linked list, where each item on a hit is placed in the front of the list. Whenever there is a miss, and the cache is full, the least recently used item, which is at the end of the list is evicted to make room, again putting the new item in the front of the list. In the following, we present and accurate model for the hit rate of such cache.

3.5.1.1 Proposed Model

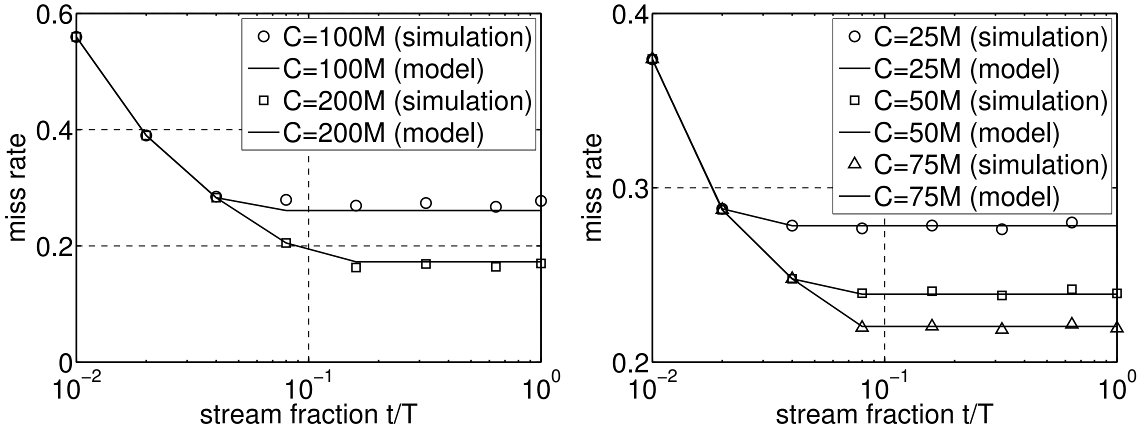
We start by formulating the miss rate of an LRU cache using the $p(t)$ model (4.3). Assume a cache of capacity C data items, fed by an input stream of length T. Suppose that the cache is full (i.e., contains C unique keys) after processing τ keys from the stream, where τ is a function of C. Defining by $m(t)$ the miss rate of LRU at time t, we get the following :

$$m(t) = \frac{E[D(1-\epsilon_{\min(t,\tau)})^{D-1}]}{E[D]}, \qquad (3.48)$$

where $\tau = s^{-1}(C)$.



(a) binomial stream (b) Zipf stream ($\alpha = 1.2$)



(c) G_A (d) G_B

Figure 3.9: Verification of miss rate model (3.48).

The above result can be explained by the fact that once the cache saturates, it experiences the same miss rate $p(\tau)$ for the rest of the stream.

Although, there is no closed-form inversion to $s(t)$ unless D follows a well-known distribution, we can perform this job numerically. We compare model (3.48) against random streams in Fig. 3.9(a)-(b), where the x-axis is drawn on a log-scale. After the cache is saturated, the miss rate in (a) stays 3 times higher than in (b), which shows the ineffectiveness of caching in a binomial stream, where items are constantly evicted from the cache. On the other hand, the Zipf stream contains a few high-degree nodes and the LRU policy keeps most of them in the cache.

We test model (3.48) on the IRLbot graph, where we stream all edges sequentially from disk and pass them through an LRU cache. Fig. 3.9(c) shows the result with two cache sizes – 100M and 200M keys. The WebBase case in Fig. 3.9(d) is run with $C = 25$M, 50M, and 75M items. These figures shows that the model is accurate for different types of input and cache sizes.

3.5.1.2 Existing Models

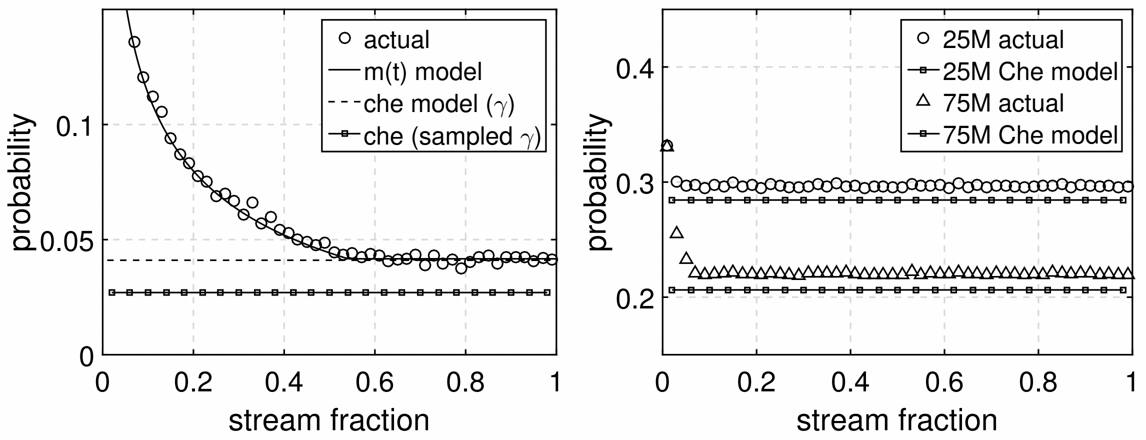
The authors in [26] derives a popular LRU hit/miss rate model given that the true popularity distribution of keys is known. Let that distribution be given by $\gamma(v)$ for $v \in V$. Now, the miss rate model in [26] for a cache of size C after the cache becomes full at t_C (i.e., after processing t_C items) is:

$$p_{miss}(t) = \sum_{v \in V} \gamma(v) e^{-\gamma(v) t_C}, \tag{3.49}$$

where $t_C = f^{-1}(C)$ and $f(t) = \sum_{v \in V}(1 - e^{-\gamma(v)t})$. Again, $t_C = f^{-1}(C)$ is obtained using numerical methods.

Now, the problem with this model (3.49) is that it requires the true popularity distribution $\gamma(v) \forall v \in V$ be known, which is impractical when we are sampling/observing the underlying process using a finite window. For instance, when the log of a web cache is collected, in many cases, the true distribution of the actual document popularity is not know. The same is true for different real workloads e.g., click stream, DNS requests, and to some extent to web crawls pointing to different web pages.

To demonstrate this point, in the following we setup an experiment where we



(a) zipf stream

(b) WebBase graph

Figure 3.10: Existing vs proposed LRU miss rate model on finite streams.

generate a stream of length $T = 10$K items, where the items all belong to a set of $|V| = 1$K items and their popularity distribution is Zipf distributed with $\alpha = 1.3$. Then, we run this stream through a cache of size $C = 333$ items (i.e., one third of the $|V|$). Fig. 3.10(a) shows the actual miss rate of this stream, along with model (3.49) using the true γ.

In addition, we also compute sampled distribution denoted by $\hat{\gamma}$ from the stream using $\hat{\gamma}(v) = d(v)/T$ for all $v \in V$, then compute the LRU miss-rate for the sampled distribution using (3.49) where $\gamma(v)$ is replaced by $\hat{\gamma}(v) = d(v)/T$. This model is shown in the Fig. 3.10(a) as well.

Observe in Fig. 3.10(a) that, first, our model (3.48) is indistinguishable from the actual miss rate, while che's model (3.49) is accurate (shown by the broken line) when the true distribution γ is available and after the cache has saturated (i.e., become full). On the other hand, if we compute the popularity distribution from the sampled stream to obtain $\hat{\gamma}$ and apply that to compute che's model, it gives us inaccurate result as shown by the boxed line in Fig. 3.10(a).

Furthermore, we repeat these experiments for the IRLbot graph and show the results in Fig. 3.10(b).

3.5.2 Random Replacement Cache

In the Random Replacement (RND) cache design, a random existing item is chosen for eviction upon a cache miss. Similar to First-In-First-Out (FIFO) cache, RND is an attractive choice for applications where lower complexity and higher speed is desired (e.g., ARM processors, Content Delivery Networks and high-speed routers [52], [113]. In [113], the authors showed that both RND and FIFO requires less memory accesses per packet than LRU and MRU, which is critical for sustaining line-rate caching is high-perforce routers. But such advantages come at the cost of lower hit-rate compared LRU, LFU, and other more complex caching methods.

While the performance (i.e., hit rate) of RND cache is a well-studied problem with accurate model for infinite (or huge) streams with known item popularity distributions, as we have shown for LRU, those models do not directly apply for finite streams (e.g., graph workloads) where the only available information is the stream itself. In this part of the chapter, we develop an accurate model for RND cache.

3.5.2.1 Proposed Model

To analyze the miss rate of RND cache, we first draw our attention to a newly proposed data structure called Incremental Hash Table (IHT), focus on the mechanics of operation there, derive an accurate for that, and then use that model to also model RND cache. This method works since the mechanics of the IHT is identical to that of an RND cache. Here, we just present the model and omit the proof for it. Please see Chapter 4, where we discuss IHT and derive its performance in more details. We start with how an IHT works and present its steady state model.

An IHT is very similar to a regular hash table, where there are a number of buckets and each bucket contains a subset of the keys determined by a hash function (e.g., division, bit-shift). IHT also maintains an eviction pointer (EP) which initially points to the first bucket and does not move until the IHT is full. At that point, if a new (currently non-existent in the hash table) arrives, the entire bucket pointed by the EP is evicted and the freed-up slots are used to put the new key in the right bucket. After that, EP advances to the next bucket. This process continues throughout the rest of the input workload and the EP evicts and advances whenever the hash table becomes full and new keys are incoming.

Note that we assume that the keys in the workload are replaced by its fixed length hash that is random and uniform, and IHT uses this numeric value of the key to determine its hash location. As a result, there is a randomness introduced between the workload and the bucket number of its key, making the two independent and random. Therefore, the dynamics of EP and its eviction is un-related to the order of the actual keys and thus random. This leads to the conclusion that the operation of IHT and RND cache are identical. We also verified in our test experiments that their behavior (i.e., miss rate) is same.

The following theorem describes IHT's and also RND cache's miss rate.

Theorem 11. *The miss-rate of a IHT denoted by* $q_R(t)$ *is given as follows:*

$$q_R(t) = \begin{cases} p(t) & \text{for } t \leq s^{-1}(C) \\ \dfrac{s(T_\infty)}{T_\infty} & \text{otherwise} \end{cases}, \tag{3.50}$$

where $\tau = s^{-1}(C)$*, and* T_∞ *is obtained using the knowledge of hash table size* C *and by numerically inverting the following:*

$$C = \frac{1}{T_\infty} \int_0^{T_\infty} s(t)dt. \tag{3.51}$$

3.5.2.2 Existing Model

The existing model for RND cache is well known due to [52] is as follows:

$$p_{miss} = 1 - \sum_{v \in V} \frac{\gamma^2(v) t_C}{1 + \gamma(v) t_C}, \tag{3.52}$$

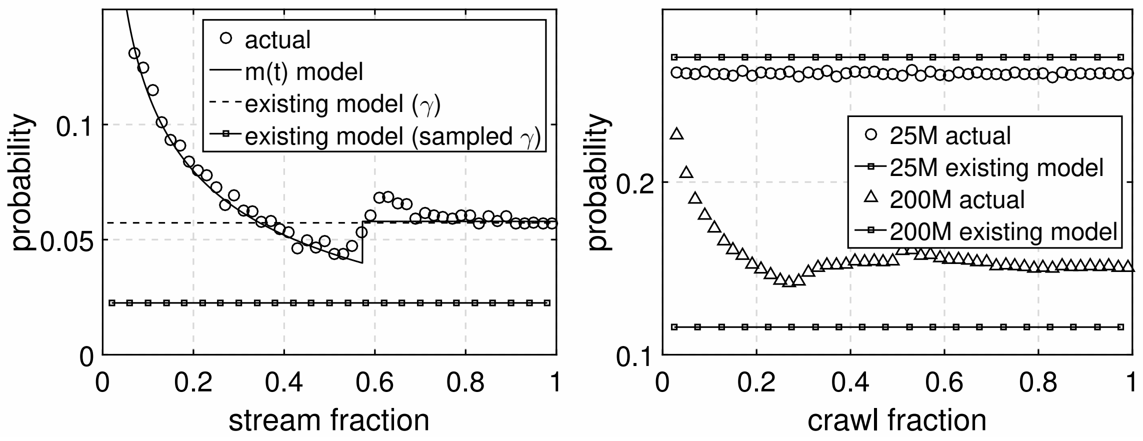
where $t_C = g^{-1}(C)$ and

$$g(t) = \sum_{v \in V} \frac{\gamma(v) t}{1 + \gamma(v) t}. \tag{3.53}$$

It is well-established that the above model is accurate on infinite streams and known popularity distributions. To find out the accuracy of this model we compare this existing model (3.52) against our proposed model (3.50) on two finite graph streams – a random Zipf ($\alpha = 1.3$) and a WebBase web graph, both examples of finite streams and show the results in Fig. 3.11. Note that the only way to obtain the popularity distribution in these cases is deriving those from the workload itself. Observe in both part (a) and (b) that our model is more accurate when the the popularity distribution is obtained from the graph. On the other hand, if the true distribution is known, then the existing model works accurately as well. In part (b) the error between the actual results and the existing model is even more noticeable.

3.5.3 MapReduce

The MapReduce programming paradigm consists of two phases – *map* and *reduce*. In the former phase, input data are processed using a user-provided parsing function that outputs a stream of key-value pairs. It is then sorted and combined in the reduce phase, as illustrated in Fig. 3.12(a). Our work below analyzes the reduce phase whose details are given in Fig. 3.12(b). Specifically, the reduce phase begins



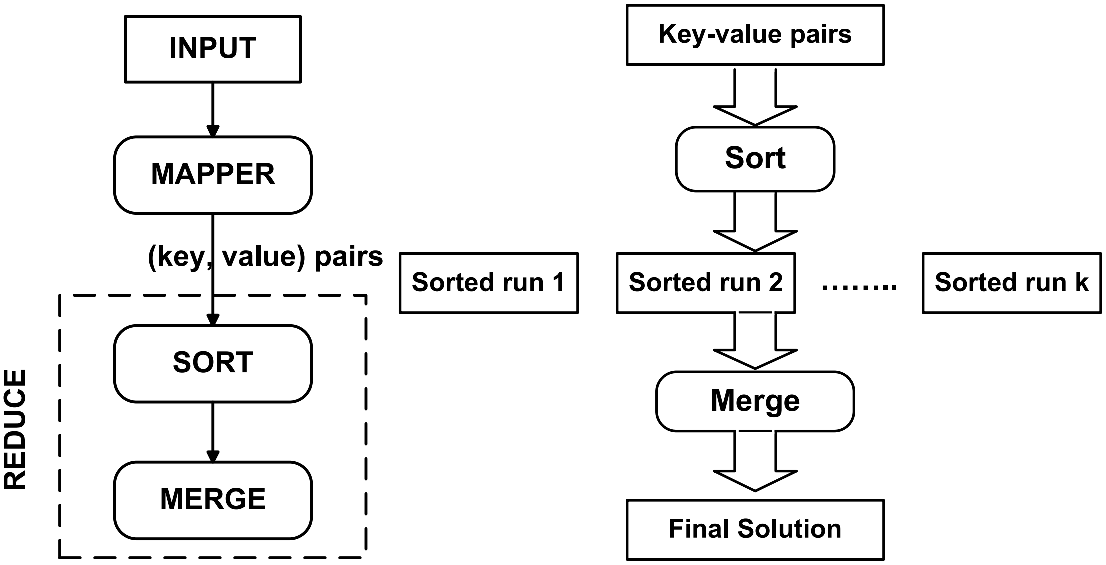
(a) zipf stream ($\alpha = 1.3$)

(b) WebBase graph

Figure 3.11: Existing vs proposed RND cache miss rate model on finite streams.

by producing $k \geq 1$ *sorted runs*, each representing a portion of input that can fit in RAM. The sorted runs are then written to secondary storage after eliminating duplicates through some *combiner* function $\theta(.)$. We assume that key-value pairs (v, a) and (v, b) are combined into a single result $(v, \theta(a, b))$, where all keys and values are fixed-size scalars. After sorting is finished, the runs are streamed back to RAM and combined using k-way merge. The final output is another stream sorted by the key, in which each node v appears exactly once.

Assume that keys and values occupy K and D bytes, respectively. The main memory can store at most r key-value pairs and $r \ll T$. The MapReduce computation goes through $k = \lceil T/r \rceil$ cycles, each of which loads r records from the stream to memory, sorts the result, and eliminates duplicates by running the combiner. The size of each sorted run is given by the number of unique keys $q(r)$ out of the r loaded during the cycle. Unfortunately, there is no accurate model in the literature to compute this number. Most of the existing methods [13], [90] sidestep this problem by assuming some known constant that converts r into the size of each run $q(r)$.


INPUT
MAPPER
(key, value) pairs
SORT
MERGE
REDUCE
Key-value pairs
Sort
Sorted run 1
Sorted run 2
……..
Sorted run k
Merge
Final Solution

(a) MapReduce (b) reduce phase

Figure 3.12: MapReduce with external sort.

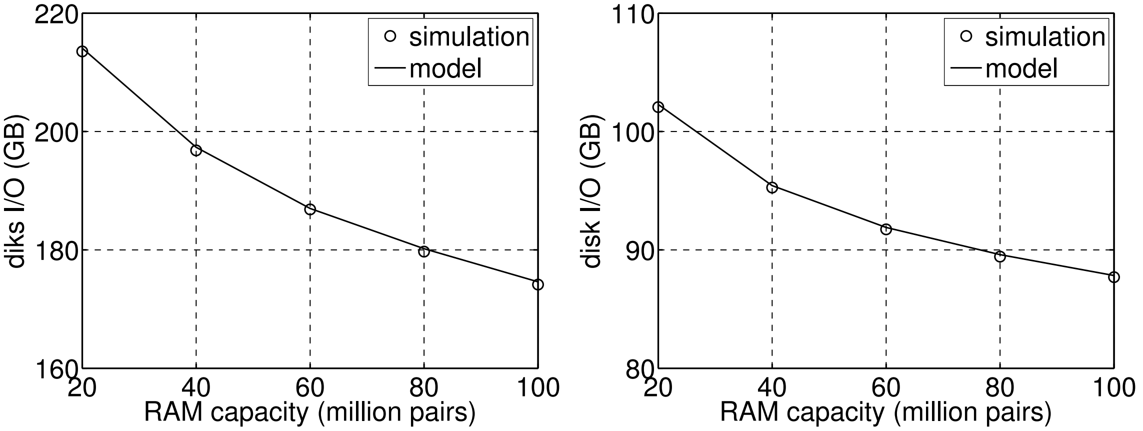
Since the order of the keys are random, the streams under consideration fall under our assumptions in the beginning of the paper. This allows us to use model (3.27) to compute the size of each sorted run. Observe that in a stream of size r, the number of unique keys is given by model (3.27) as:

$$q(r) = E[\phi(S_r)] = nE[1 - (1 - \epsilon_r)^{\mathcal{I}}], \tag{3.54}$$

which, unlike the assumption in [13], [80], [90], is far from linear in r. Since there are k sorted runs from k cycles, their total size is:

$$q = nk(K + D)E[1 - (1 - \epsilon_r)^{\mathcal{I}}]. \tag{3.55}$$

Note that the full input stream of size $(K + D)T$ is read in the beginning and n unique pairs are written at the end. Counting both read and write I/O for each



(a) IRLbot host graph (b) WebBase web graph

Figure 3.13: Verification of (3.56).

sorted run, we get the total amount of disk overhead as:

$$w = n(K + D)\left(1 + E[\mathcal{I}] + 2kE[1 - (1 - \epsilon_r)^{\mathcal{I}}]\right). \tag{3.56}$$

To verify (3.56), we use a MapReduce task that computes the earliest time $\delta(v) = \min\{t \geq 1 : Y_t = v\}$ each node v is seen in the stream. Therefore, each key-value pair consists of an 8-byte key hash for the node ID and an 8-byte timestamp representing the time of its earliest discovery. The combiner function is simply the minimum of the two values under consideration. We run the above computation on the IRLbot and WebBase graphs under varying RAM capacities. The results are shown in Fig. 3.13, which demonstrates that the model is accurate in both cases. For example, Fig. 3.13(a) shows that the disk I/O for processing the IRLbot graph is 215 GB with $r = 20$M pairs in RAM. Excluding 70 GB for the input and final output, the remaining 145 GB are for sorted runs. This demonstrates how sorted runs constitute a significant portion of MapReduce overhead.

Also observe that the results in (3.56) allow accurate modeling of the total completion time of MapReduce jobs (we omit the map phase as it is highly user-dependent and usually much faster than the reduce phase). Assume that the sorting and merging speed are respectively s and g keys per second. Furthermore, suppose that the disk I/O speed is d bytes per second. Then, the total time needed for the reduce phase is:

$$\Gamma = \frac{T}{s} + \frac{w}{d} + \frac{nk}{g} E[1 - (1 - \epsilon_r)^{\mathcal{I}}]. \tag{3.57}$$

Result (3.57) specifies the MapReduce runtime for a given RAM size r. In addition to analyzing latency, this model can help optimize the MapReduce architecture by suggesting proper selection of r for a given (s, d, g), or vice versa.

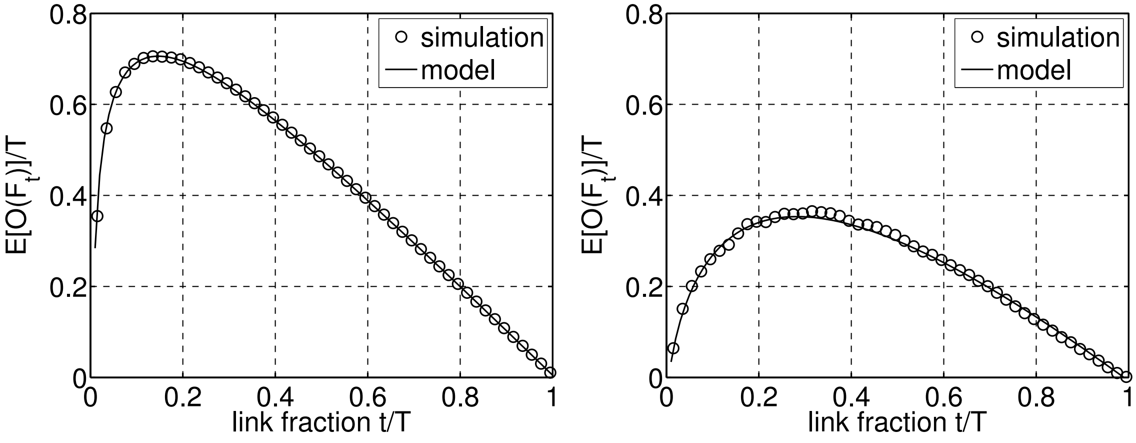
3.5.4 Frontier in Graph Traversal

Frontier size is a crucial metric for crawlers, because it indicates the amount of resources required for maintaining crawl queues. The larger the crawl queue, the more overhead it places on uniqueness check (to avoid re-crawling the same pages/nodes), prioritization (by sorting the nodes based on some importance metrics), etc. Here, we develop models of frontier size for a number of crawling strategies and present a comparison among them.

Recall that the frontier at time t is denoted by F_t. Similar to (3.30), define the combined out-degree of nodes in F_t as:

$$\mathcal{O}(F_t) = \sum_{v \in F_t} \mathcal{O}(v). \tag{3.58}$$

This can also be written as the number of out-edges emanating from the seen



(a) correlated degree

(b) uncorrelated degree

Figure 3.14: Verification of frontier out-degree (3.60) with Zipf $\mathcal{I}$ and $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10\text{K}$.

nodes minus the total out-degree of the nodes moved into the crawled set:

$$\mathcal{O}(F_t) = \sum_{v \in S_t} \mathcal{O}(v) - \sum_{v \in C_t} \mathcal{O}(v). \tag{3.59}$$

Taking an expectation of (3.59), we get:

$$E[\mathcal{O}(F_t)] = nE[\mathcal{O}(1 - (1 - \epsilon_t)^{\mathcal{I}})] - t. \tag{3.60}$$

We examine in simulations two cases of degree dependence: a) $\mathcal{O} = \mathcal{I}$ with a positive correlation; and b) $\mathcal{O}$ and $\mathcal{I}$ drawn independently from the same Zipf distribution. Fig. 3.14 compares model (3.60) against the observed values from BFS simulations, showing that the model is accurate. Observe that the curve in Fig. 3.14(a) is heavily skewed to the left, reaching its maximum value (i.e., 70% of T) after seeing just 15% of the stream. On the other hand, $E[\mathcal{O}(F_t)]$ in Fig. 3.14(b) peaks after processing 30% of the stream, but at a significantly smaller value (i.e., 38% of T). After the peaks, both curves drop to zero with a linear slope, just as predicted by the model. For exploration of large graphs of unknown size (e.g., the web), the shape of these curves provides insight into *crawl coverage* (i.e., percentage of links processed thus far) – positive slopes in (3.60) indicate that most links are yet to be seen.

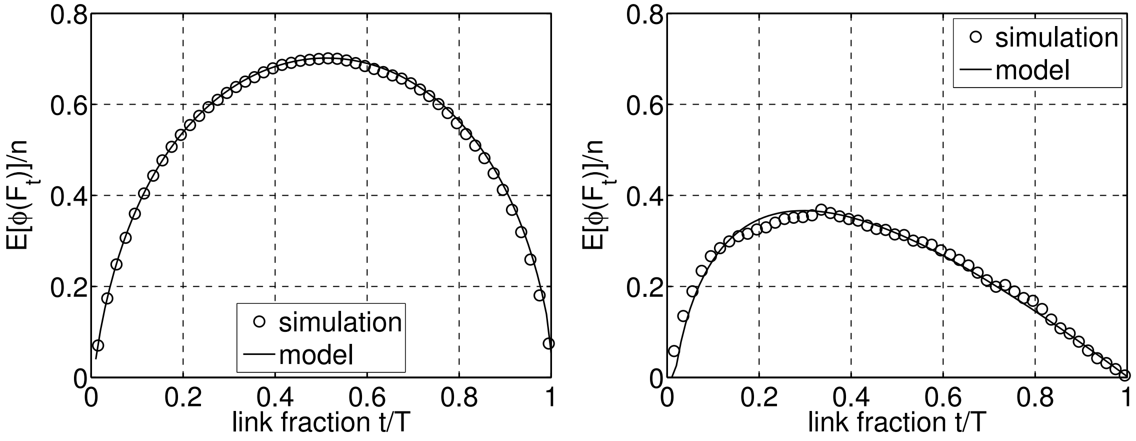
We finally arrive to the most interesting metric of this section – size of the frontier. Observe that $\phi(F_t)$ increases by addition of previously unseen nodes, which happens with probability $p(t)$, and removal of crawled nodes. Whenever a node X_t is removed from the frontier for crawling, all of its $\mathcal{O}(X_t)$ outgoing edges are processed before the next removal. Therefore, the rate of node removal from F_t is $1/\mathcal{O}(X_t)$. Assuming $E[1/\mathcal{O}(X_t)] \approx 1/E[\mathcal{O}(X_t)]$, the size of the frontier at t is:

$$E[\phi(F_t)] \approx E[\phi(F_{t-1})] + p(t-1) - \frac{1}{E[\mathcal{O}(X_{t-1})]}. \tag{3.61}$$

Quantity $E[\mathcal{O}(X_t)]$ depends on the crawl strategy that orders the frontier and the degree-properties of the graph. We next examine a number of such cases and compute the corresponding $E[\mathcal{O}(X_t)]$'s. First, we consider BFS with an arbitrary degree distribution. Note that after v is inserted into the BFS queue at t, all $\mathcal{O}(F_t)$ out-edges currently pending in F_t will be processed before v is crawled. Therefore, we can estimate the average time between the first discovery of a node at t and its crawl as $E[\mathcal{O}(F_t)]$. Using (3.46), this leads to:

$$\begin{aligned} E[\mathcal{O}(X_{t+E[\mathcal{O}(F_t)]})] &= E[\mathcal{O}(Y_t) | Y_t \in U_{t-1}] \\ &= \frac{E[\mathcal{I}\mathcal{O}(1-\epsilon_t)^{\mathcal{I}-1}]}{E[\mathcal{I}(1-\epsilon_t)^{\mathcal{I}-1}]}. \end{aligned} \tag{3.62}$$

Applying (3.62), we compute $E[\mathcal{O}(X_t)]$ for all t iteratively. Note that (3.62) skips



(a) correlated degree (3.62) (b) uncorrelated degree (3.63)

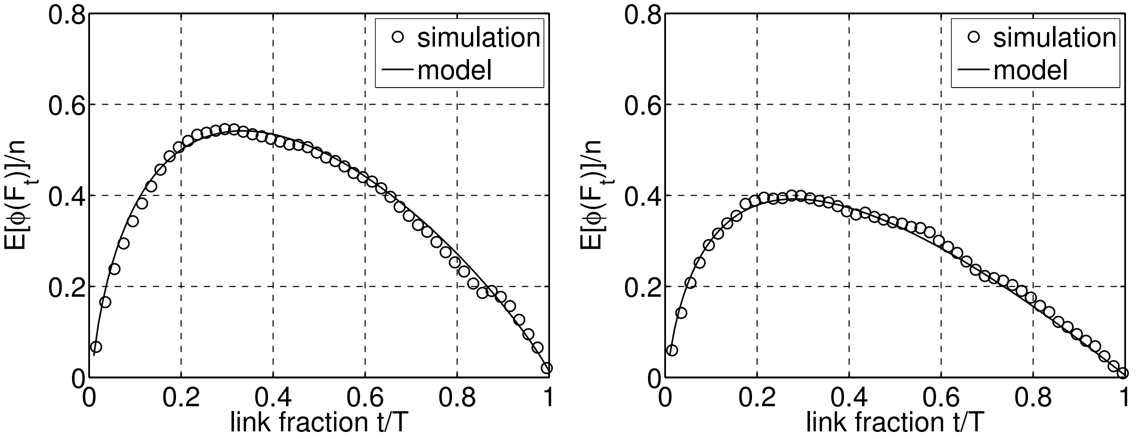
Figure 3.15: Verification of the frontier size in BFS with Zipf $\mathcal{I}$ and $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10$K.

values for some t's, which are interpolated from the neighboring values. We then use $E[\mathcal{O}(X_t)]$ in (3.61) to compute $E[\phi(F_t)]$ for all t. Fig. 3.15(a) compares this model against simulations and confirms its accuracy. Our second case involves uncorrelated in/out-degree, where we can estimate the size of the frontier by dividing its total out-degree by $E[\mathcal{O}]$:

$$E[\phi(F_t)] = \frac{E[\mathcal{O}(F_t)]}{E[\mathcal{O}]} = nE[1 - (1 - \epsilon_t)^{\mathcal{I}}] - \frac{t}{E[\mathcal{O}]}. \qquad (3.63)$$

Note that we get the same result as (3.63) after expanding the recursion in (3.61) and using $\sum_{\tau=1}^{t} p(t) = E[\phi(S_t)]$. In Fig. 3.15(b), we compare model (3.63) against BFS on the uncorrelated-degree graph. Observe that Fig. 3.15(b) closely follows Fig. 3.14(b) since they are scaled versions of each other.

Our third crawl method, which we call *Frontier RaNdomization* (FRN), picked candidates randomly from the frontier. In contrast to BFS, FRN avoids back-to-back hits against the same website and achieves better politeness guarantees during



(a) correlated degree

(b) uncorrelated degree

Figure 3.16: Verification of model (3.64) in FRN with Zipf $\mathcal{I}$ and $\alpha = 1.5, E[\mathcal{I}] = 10, n = 10\text{K}$.

large-scale web crawling [122]. Since any node in the frontier is equally likely to be picked, $E[\mathcal{O}(X_t)]$ in FRN is simply the average out-degree of the nodes in F_t. Hence,

$$E[\mathcal{O}(X_t)] = \frac{E[\mathcal{O}(F_t)]}{E[\phi(F_t)]}. \tag{3.64}$$

We use $E[\mathcal{O}(X_t)]$ from (3.64) in (3.61) to compute the $E[\phi(F_t)]$ model for all t iteratively. Then, we compare the result against FRN simulation in Fig. 3.16. Part (a) of the figure shows the correlated case, which is skewed to the left compared to Fig. 3.15(a) under BFS. Since both curves have the same $E[\mathcal{O}(F_t)]$, we can conclude that FRN still exhibits degree-bias for crawled nodes, but due to randomization of the frontier, the bias is less prominent compared to BFS. The uncorrelated case in Fig. 3.16(b) is, as expected, almost identical to the uncorrelated scenario in BFS.

Observe in Fig. 3.15(a) that BFS grows the frontier set to 70% of n. In contrast, FRN's frontier does not exceed 55% of n as shown in Fig. 3.16(a). This can be explained by the fact that BFS discovers unique nodes at a higher rate and grows its queues faster compared to FRN. As a result, FRN is not only more polite, but also more efficient in resource usage.

3.6 Conclusion

We proposed an accurate analytical framework for characterizing applications that process random data streams, including such properties as the probability of uniqueness for discovered keys, the number of unique values accumulated by a certain time t, the average degree of seen nodes, and the size of the frontier during crawls on large-scale graphs under three different strategies. We demonstrated that these models were applicable not just to synthetically generated streams, such as those produced by BFS on random graphs, but also real workloads stemming from LRU caching and MapReduce processing of IRLbot and WebBase graphs.

4. CLASSICAL EXTERNAL MEMORY ALGORITHMS

4.1 Introduction

The motivation behind Google's highly successful programming paradigm MapReduce is to provide a simple abstraction for large-scale information processing, which is a common problem in many enterprizes. To handle massive data sets, MapReduce splits the workload into manageable chunks and processes them individually, with an option to combine them at a later stage. Since both CPU and I/O overhead often directly depend on the size of these chunks, their analysis is required for obtaining usable MapReduce models.

There has been limited work towards characterizing the performance (i.e., runtime and resource usage) of MapReduce [80], [90], [144]. Their common limitation is to assume a known multiplicative factor γ that converts the size of input that fits in memory to that of the total disk I/O. However, the question of how γ depends on RAM size and workload characteristics, as well as deduplication algorithms and various data structures, remains open. This precludes reasoning about the required hardware resources, task runtime, and optimal design in various scenarios of interest. For example, given keys whose popularity follows the Zipf(α) distribution and RAM size R, how much I/O would be needed for a stream with T keys? How many reduce operations would be performed? What is the total runtime of the program? What values of α allow hash-table deduplication to beat quicksort? How does the overhead scale with R and α?

To our knowledge, none of the existing work can accurately model the MapReduce process or provide answers to these questions. Our goal below is to address this problem. Our angle is the repetition of keys that usually happen in typical MapReduce workloads (e.g., crawl stream of the Internet or other graphs, DNS and click logs, web cache logs), and the effect of this repetition on algorithm performance.

4.1.1 Contributions

Assume the input to MapReduce is a uniformly random stream of key-value pairs, usually much larger than RAM. As keys may repeat, the main purpose of MapReduce is to identify all duplicates and call a user-provided function to combine the corresponding values. We assume a *scalar* combiner, which does not increase the size of the value field, and focus on shared-memory systems with a single CPU. Modeling this scenario is a prerequisite to understanding the more complex cases. In fact, as discussed at the end of the chapter, these models can be extended to multi-core and cluster environments with little additional effort.

First, define the *uniqueness probability* $p(t)$ to be the likelihood to encounter a previously unseen key in a given location t of the stream. Based on the $p(t)$ model in [3], our first contribution is to develop models for merge-sort deduplication, where we obtain the amount of disk I/O, the number of reduce operations, and total runtime. Our model takes into account reduction overhead as the keys travel up the merge tree. Our second contribution is to analyze hash-based sort, drawing attention to its better compaction of the keys and possibly faster operation than merge-sort.

After that, we focus on the classical *replacement selection* method, which is the key part of this chapter. Disk I/O of replacement selection has been studied extensively in literature in case of all-unique keys (i.e., no duplicate presence of keys). We study the traditional two-min-heap based design with de-duplication, proposed originally in [112], and present accurate disk I/O models for it.

Our final contribution is to implement and compare the studied designs against each other using real-world data sets. Results show that the proposed hash-table-based replacement selection produces the least amount of I/O. We also demonstrate that our models are not limited to single threaded implementation, which has been our experiment platform for this chapter. Rather they can be easily extended to multi-core and parallel implementation with little modification.

4.2 Related Work

The body of related work spans two fields. First, we examine MapReduce-related literature to extract existing performance models. Then, we move on to the more classical replacement selection literature.

4.2.1 MapReduce Performance

A Hadoop task requires setting a large number of parameter values. But setting these parameters requires domain knowledge or access to large number of traces, which are difficult to obtain. In the following, we discuss some such papers who attempt at automating this process. Note that this will require building models of runtime based on data and task properties, making these papers relevant to our work here.

In [135], the aim is to select these parameters automatically to fulfill various objectives (e.g., optimal resource usage, shortest run time, lowest cost). The authors first formulated the total runtime as a weighted function of various parameters, where the weights are obtained from some training runs on a small cloud setup. Then, this function is used as the objective function of optimization problems, which can be subjected to various cost or resource constraints (e.g., task deadline, memory limitations). The authors trained the runtime function for a number of MapReduce computations (i.e., WordCount, PageRank, Join - all part of Hadoop distribution) on a 16-node cluster. The trained models are accurate for tests in the same setup. But the models are not tested against cloud with different settings, making the formulations arguable. Furthermore, the proposed optimization is not tested against real computations.

Work in [90] is one of the few to characterize MapReduce completion time taking CPU usage and disk I/O (mapper and reducer internal spills and final output as a result of multi-pass merge phase) into consideration. The authors in this paper propose an analytic model for intermediate data of a MapReduce job and then a method for optimizing Hadoop parameters based on this model. They also present a number of hash based techniques that improve the performance of the basic Hadoop implementation. They demonstrate that pipelining, which is first proposed in Hadoop Online [38], does not reduce overall completion time significantly. This paper is an important first step towards characterizing I/O cost in MapReduce and has much applicability in external merge-sort based application. However, the authors make some simplifying assumptions that make their model inaccurate. The parameters map and reduce selectivity, which are task and workload dependent, are just assumed to be known. Furthermore, the authors assume that when a number of files are merged together, the size of the output file is just the size of all the files added together. They used this assumption while computing the volume of the multi-pass merge volume, leading to inaccurate results. This can be improved by using the merging model presented later in this chapter.

Authors in [8] propose a number of techniques to automate the parameter selection process, while those in [138] present a simulation framework called MRPerf that explores the design space and forecasts the resulting runtime of a MapReduce job. Then, they demonstrate that this simulation framework is able to identify some bottlenecks in a medium-size Hadoop cluster, removing which results in a 28% performance gain. Similarly, [60] presents a MapReduce simulator called MRSim, which is a discrete event based simulator and is capable of capturing the job behavior (e.g., completion time, hardware utilization) accurately.

The objective of [144] is to help tune the number of mappers and reducers. The authors first present a simplistic model for the time taken by a single mapper, a reducer, data transfer between a pair. Then, the total time is derived by assuming that all mappers and reducers work in parallel. Then, the optimal number of map and reduce tasks are chosen by simply differentiating the total time equation. The authors train this model in a 13-node cluster for the WordCount task and show that their model closely resembles actual run times. However, since machine learning is used for training and testing is performed on the same setup, many major factors of performance may have been overlooked by the model. In addition, the model is not detailed enough to capture algorithmic details of the map and reduce algorithms which may significantly affect performance. Furthermore, the model assumes that volume of intermediate and output data are known, which is not true in general. Our model presented later in this chapter that derives the intermediate data can complement the models presented in this paper and make them accurate.

In [63], the authors attempt to find optimal resource configuration for MapReduce by modifying the Hadoop framework to collect job profiles, and training a classifier based on the existing profiles. The collected profiles include various task level timing and resource usage information. Then, the authors use a cost-based optimizer (cost specified as either run-time, or resource usage) to search through the space of configuration parameters using a number of techniques and find out the optimal one for a particular task, input, and cluster configuration. This method is then compared against a rule-based optimizer constructed using various thumb rules and suggestions from Hadoop experts on a number of Hadoop data sets. The results demonstrate that this method produces configurations that lead to faster job completion. The authors also show that the run-time predicted by their system closely follow that of the simulation. The advantage of this work is that it considers many parameters of the configuration space (e.g., mapper memory, whether or not to compress intermediate data) in addition to just the number of mappers and reducers, which many of the previous papers limited their analysis to. However, this paper still requires previous job traces and training on those. In our method, we eliminate the requirement for training completely.

The authors of [68] present a benchmark suite to correctly and universally evaluate a Hadoop setup and see its implications. This suite contains both synthetic and real-world workloads that are representative of a large class MapReduce workloads. Then, the authors evaluate a large number of MapReduce jobs from the benchmark to explore their speed (i.e., job running time), throughput (i.e., the number of tasks completed per minute), the HDFS bandwidth, system resource utilizations, and data access patterns.

To keep our analysis simple in this chapter, we start with a custom shared-memory implementation and model that first. Therefore, existing literature regarding shared-memory MapReduce designs are of great interest to our work. We cover a number of those that are well-known in the following. Note that in the end of this chapter, we also extend our analysis to multi-core and cluster MapReduce designs.

The paper [115] presents the Phoenix API, a shared memory implementation of MapReduce that uses shared memory for data transfer between cores (on/off chip, the latter for multiple processors). The objective of this paper is evaluate the effectiveness of MapReduce model in shared memory systems. In this framework, processor cores act as nodes where map and reduce tasks are allocated. The available memory is separated into two-dimensional grid where each cell corresponds to a pair of map and reduce tasks. The map task, after parsing and applying map function to the input, produces key-value records and put each record in a cell determined by the key. Each cell contains a hash table where the keys with corresponding values are inserted. This cell-wise division enables concurrency and automatic shuffle of intermediate data to reduce workers. This framework has its own buffer management, fault recovery, concurrency, and locality management schemes. Phoenix API is then evaluated on a number of benchmarks that all fit in RAM. Experiments on CMP and SMP show desired (sometimes super-linear) speedups, with some exceptions in SMP where interconnection bus is saturated.

The authors in [129] introduces Phoenix++, a shared memory MapReduce and the successor of Phoenix [115]. The authors identify number of MapReduce problem types for which the Phoenix framework work sub-optimally. In these cases, programmers have to write extra code to solve this problem. Phoenix++ exposes more control to the programmer by allowing the use of different data structures depending on the task type. Moreover, in the original Phoenix, the combiner function is applied only at the end of map phase, which leads to memory pressure. Phoenix++ reduces this pressure by applying the combiner function for every emitted value, resulting in a more compact memory footprint. The authors then demonstrated that Phoenix++ achieves 4.7x speedup on the average over Phoenix on a number of tasks. Furthermore, Phoenix++ code is much more modular and compact (less lines of code) compared to Phoenix.

The objective in [136] is to identify the majors performance factors in a shared memory MapReduce and derive their relation with the completion time of the map and the reduce phases. First, the authors identify that the order of unique keys and their frequencies affect performance. Based on that, they derive performance models for a number of types of key arrangements (i.e, skewed and uniform) which correspond to the worst- and best-case performances respectively. These models give detailed consideration to specific hashing and sorting techniques used in different shared memory frameworks (Phoenix and Metis). The derived models are expressed as weighted combination of various functions of the key performance factors (e.g., number of map and reduce threads, hash buckets, ratio of unique to total keys). Rather than using machine learning techniques, the authors determine the weights of these functions from test runs. Then, they compare these models against simulations, finding them to be accurate in some cases. In other cases, the simulations and model show the same dynamics, but in different scale. In the end, the authors present a set of experiments using real data sets. They show that how various phenomena observed in these experiments can be explained by their models. However, the derived models are verified only in simulations, not in real data sets. Furthermore, the models assume that each key has the same frequency (making each key equally likely), which does not hold in general. In our models, we do not make any such assumptions.

To analyze MapReduce runtime in cluster, there has been a body of work that has approached it as a job scheduling problem and even proposed different scheduling mechanisms to improve runtime. We investigate some of those in the following, focusing mainly on those that deal with the amount of intermediate data shuffled between the nodes.

In [13], the authors applied Divisible Load Theory (DLT), a well known theory in distributed processing, for deriving an optimal scheduling strategy for MapReduce programs such that the completion time is minimized. The authors assume that each node contains only one mapper and at most one reducer. This generalizes to the general MapReduce because all map tasks assigned to a node can be consolidated to a single mapper - each task starting sequentially after one another. In addition, reduce task (if any) in a node starts after completion of the map task (thus eliminating pipelining). Assuming reducer execution time roughly constant, the objective is to minimize mapper computation and mapper-to-reducer transmission. The authors provide the following heuristic for optimum scheduling: processor P_i first transfers data to reducer j and then signals P_{i+1} to transfer its data to j. Then P_i will work on reducer $j+1$ given it has already received signal from P_{i-1} for$j+1$. According to this schedule, the authors compute the optimal size of data allocated at each mapper i and consequently the completion time of the computation. However, the formulated runtime model uses a constant map output to input data size ratio, which is not known beforehand.

The authors in [145] investigate a crucial scheduling problem with the Hadoop MapReduce platform, where the universal assumption is that all members of the cluster are homogeneous. According to their experiments in Amazon EC2, this assumption is not usually true. As a result, some speculative execution of stragglers become a complicated problem, since the strategy for that is also based on this homogeneity assumption, thus leading to even worse performance compared to non-speculative execution. The same is true for other cloud setups since it is very common for those to house different types/generations of machine to build the cluster. To solve this problem, the authors propose a new job scheduling strategy without any assumption of homogeneity. The key aspects of their scheduling strategy, which they term Longest Approximate Time to End (LATE), are: prioritizing task to speculate, selecting a fast node to run on and controlling/capping speculation to prevent thrashing and consequent performance degradation. They demonstrate that this new strategy shortens the response times of some MapReduce jobs by a factor of 2 in large EC2 clusters.

For a Hadoop task with a given deadline, it is difficult to decide the minimum amount of resources (mappers and reducers) to finish the job on time. To solve this problem, the authors in [137] derives the theoretical upper and lower bounds of various map and reduce tasks under a Earliest Deadline First (EDF) scheduler using Makespan theorem. The models for these bounds use the job profile as input and provides the minimal number of mappers and reducers to be used so that the job can be completed with in the deadline. The authors also derived an average job completion time by simple averaging of the upper and lower bounds. They tested these models against simulation on a 66-node cluster and found that the upper bound model can always guarantee completion before deadline, while the average time model gives completion time closest to the deadline. They also tested their system with multiple jobs and under different load condition on the cluster. They demonstrated that the predicted number of map and reduce workers can guarantee deadline completion when the load is less than 100%. The interesting aspect of this paper is that the authors used the theoretical upper bound of task time to estimate the resource requirements for meeting the deadline. This is a good use of the theoretical bounds of run time. But, in practice, actual runtime can be far off from the theoretical upper bound. This paper only mention what percentage of jobs exceeded the deadline, but does not report how the actual runtime deviated from the bound. Furthermore, job profiles, on which the models are built upon, are not easy to obtain.

In [111], the authors compare MapReduce against parallel DBMS from a number of angles (e.g., performance, development complexity, scalability). First, the authors presented qualitative comparison between MapReduce and parallel DBMS in terms of schema and indexing support, development effort, data distribution, execution and scheduling strategy, scalability, and fault tolerance. They argued that although MapReduce shows better scalability and fault tolerance compared to parallel DBMSs, it lacks in the other cases. Then, the authors compared Hadoop, an open source MapReduce implementation, against DBMS-X and Vertica, two commercial parallel DBMS distributions, for a number of MapReduce computations (i.e., grep, select, aggregation, and join) in a 100-node cluster. The experiments demonstrated that Hadoop required much less data loading time (to HDFS) compared to both DBMS, but was outperformed during the actual task executions. This is attributed to the slow start-up of Hadoop tasks and the more disk scanning in the absence of index.

MapReduce, although being a simple data processing model, does not directly support relational queries (e.g., join). MapReduce programmers can still implement such queries by running map and reduces stages on two different sources and then merging the reduced results, but at the cost of increased programming complexity and burden on the programmer side. To solve this problem, the authors in [143] augment the MapReduce model by including a final merge stage, which enables MapReduce to support various relational queries. Similar to the original MapReduce, this new programming model contains user-provided *mapper* and *reducer* functions. In addition, it contains a number of other functions (e.g., partition selector, processor, merger, configurable iterators) to fully support various join algorithms as part of the standard. Although this framework bridges the gap between the MapReduce and RDBMS, the contribution of this paper is intuitive, if not trivial.

The authors in [38] proposes a modified MapReduce framework that supports online aggregation and early returns. This changes the tradition batch processing style of MapReduce by pipelining results between mappers and reducers of a single job, or between multiple dependent jobs. This is done by eagerly pushing results produced from the map tasks directly to the reducers tasks, and still writing results to local disks for fault tolerance purposes. But before pushing data to the reducers, each mapper perform some aggregation using the combiner function. This communication of data is polite to the network and resilient to the case when all map and reduce task are not running concurrently (i.e., there is not enough slot). Pipelining between two jobs is achieved using more sophisticated schemes, while the task scheduler is modified to encourage co-scheduling of dependent jobs in the same physical machine. The authors then experimented this modified the Hadoop implementation of MapReduce and experimented it on a 60-node Amazon EC2 cluster for a number of jobs. They demonstrated that in some cases, in addition to providing early and near accurate results, their approach achieves up to 25% reduction in job completion time. Furthermore, since reduce tasks are activated earlier in this new framework, better system utilization is ensured. However, not all the experiments show significant speed up. In addition, the new framework complicates the basic MapReduce design with different scheduling and fault-tolerance policies, and redundant computations by the reducers, offsetting the gain achieved by better system utilization.

In [72], the authors present Microsoft's Dryad, a programming paradigm for parallel processing of large scale data. Although MapReduce is a very simple programming model, it is criticized for sub-optimal performance. Dryad, on the other hand, scales by utilizing multiple CPU cores and multiple machines in a cluster, and thus shows much better performance compared to MapReduce implementations (e.g., Hadoop). In this framework, a computation is specified as a graph using a graph description language, where "vertices" represent sequential computation and the "channels" are for communication among them. Here, the channel is an abstraction for either shared memory, TCP pipe, or file in secondary storage. The scheduling system is in charge of optimal job placement such that results can be transferred between tasks without writing to disk, which ensures better performance. Data splitting, job co-location in nodes, pipelining, and even the graph modification during the runtime (based on node failure on job completion) are also done as part of the framework. Furthermore, this framework is resilient against node/task failures. The authors then compared the performance of Dryad against a commercial SQL server in a 10-node cluster and showed that Dryad outperforms the SQL server when run using a single node. In addition, the performance of Dryad scales almost linearly as they used more an more machines in that cluster. Then, to test scalability, they run a query-log mining task on a 1800-node cluster, which took 11 minutes to process a 10TB data set. Although, this paper discusses various aspects, policies and mechanisms of Dryad in details, it lacks in experiments to demonstrate its performance. Comparisons with parallel DBMS and shared memory MapReduce implementations are expected.

In [34], the authors adapt the well-known MapReduce framework to scale a class of machine learning algorithms to take advantage of mainly multi-core architectures. The learning algorithms that fit in the Statistical Query model, can be written as a special *summation form*, and thus are candidates for such parallelization. Examples include linear regression, k-means, logistic regression, naive Bayes, SVM, ICA, PCA, gaussian discriminant analysis, EM etc, where the authors demonstrate mostly linear speed-up with the number of available processors.

MapReduce was originally designed for non-iterative computation, which can adapted to iterative ones (e.g., page rank) by running multiple iterations. But this is sometimes wasteful because there are some useful information that could be retained between iterations and save on the total runtime. The authors in [44] present such an iterative runtime based on MapReduce, but modified targeting such iterative applications. They call this framework and the corresponding runtime environment "Twister". The authors demonstrate that their runtime extends the applicability of the basic MapReduce to a large class of iterative algorithms e.g., clustering, machine learning, and computer vision.

In [61], the authors propose a MapReduce framework to run efficiently on GPUs that would hide the GPU internals from the programmers and provide a familiar MapReduce interface. The authors evaluated the proposed framework on NVIDIA

G80 GPU and demonstrated that it performs up to 16X faster compared to Phoenix, the state-of-the-art for multi-core CPUs.

In [69], the authors evaluate Hadoop to be implemented on virtualized environments, specially in commercial/non-commercial cloud setups. Such implementations suffer from the lack of access to full hardware, but enjoys the flexibility and fault tolerance of the virtualized environment. In this paper, the authors point out a number of issues that need to considered in such deployment.

Authors in [82] present an extension to the original MapReduce framework that has very minimal overhead, but can identify and mitigate skew among the nodes in the cluster. This extension, which is completely transparent, automatically detect idle nodes, some straggling jobs that are overdue, partition that task to retain the original job ordering, and reassign that to the idle node. Experiments on a number of real-world workloads suggest that this tool can significantly reduce the total job completion time in the presence of skewed behavior of the cluster.

The problem of automatic resource provisioning of a Hadoop cluster is addressed in [83], where the proposal is a two-phase machine learning and optimization-based technique that learns from a training set of jobs, matches the resource-usage patterns of the current job with the known ones, and then automatically sets/adapts the parameters of a Hadoop deployment. The proposed method, which the authors call AROMA, is implemented in a set of virtualized HP ProLiant blade servers, and tested on diverse MapReduce jobs, and shown to be very effective in their automatic provisioning.

When a MapReduce job is deployed in cloud environments, where there is no dedicated resource and node failure/unavailability is a common place, the result is poor predictability and performance. The authors in [91] presents a new model where the existing volunteer computing behavior is supplemented with a small degree of dedicate resource allocation. This new technique, termed MOON, is tested on an emulated 60-node volunteer-computing cluster where this method shows 3-fold reduction improvement in completion time.

MapReduce is very flexible for simple data flow operations. But for complicated multi-level data flows, it require manual arrangement of multiple MapReduce jobs and often reinvent the wheel for such jobs as *joins*, for which the declarative language like SQL is much better suited. To avoid this problem and achieve the best of both, authors of [53] presents a new high-level language construct called Pig. User programs written in Pig are complied to a sequence of MapReduce jobs and executed in Hadoop. The paper also compares Pig jobs against raw MapReduce jobs written directly.

Although there are many well-known techniques for *join* in DBMS, its implementations in MapReduce are often sub-optimal and inefficient. Since join is used in many cases for log processing and mining, its efficient implementation is necessary. Therefore, the authors in [15] present a number of well join methods in MapReduce, compare them on a 100-node cluster and show the results using each method.

A new scheduling strategy based on workload characteristics is proposed in [95], MapReduce tasks with complementary resource usage are co-located to the same nodes. Task are characterized based on various sampling techniques complemented by different static analysis techniques (e.g., Java byte code analysis). Experiments on local and Amazon clusters show upto 17% improvements in throughput while processing diverse co-existing workloads.

Existing resource allocation in MapReduce happen at the granularity of slots, which are mainly fixed-size, static portion of nodes. The authors in [121] argue that such fixed and coarse-grained scheduling does not work well in cases each MapReduce job shows varying behavior in their resource usage. Therefore, the authors propose a coordinated and more fine-grained management of resources such that dynamic reconfiguration of the nodes are possible by acquiring resources from other nodes. This method, which they call MROrchestrator, also identifies bottlenecks in the computation and remove those by borrowing idle resources. The authors show that their technique result in up to 38% reduction in completion time and 25% increase in resource utilization on two 24-node native and virtualized Hadoop clusters.

In [134], the authors present a data warehousing solution on top of Hadoop. This solution supports queries in SQL-like declarative language called HiveQL and compiles them to MapReduce jobs on Hadoop. HiveQL supports type systems with tables that contain those primitive types (e.g., arrays and maps). Hive also includes a component called Metastore, which maintains schema and statistics about the data that are then used for query optimization. The authors also report that Hive is used in Facebook to maintain hundreds of tables containing 700TB of data.

Although Hive provides a SQL-environments of different MapReduce jobs, the problem of query optimization is handled rigorously in that platform. The assumption is that once the query is written in Hive, it is compiled into a sequence of jobs that are then mainly hand optimized by an expert. To solve this problems, the authors in [140] present a query optimization extension on top of Hive that generates efficient and optimized query plans, as demonstrated by their experiments.

Then, [4], [66], [32] all propose distributed data data warehousing model based on Hadoop, where the first one focuses on spatial/GIS data and corresponding queries. In addition, [127] compares a number of such high-level query languages out there on top of Hadoop.

Then, authors in [120] point out that due to the unstructured nature of data, it is common among different warehouses to contain repeated instances of the same data, thereby increasing the required storage and communication bandwidth. To mitigate this problem, the authors present a model where data duplication is detected at the file level, controlled degree of duplication is retained for performance and fault tolerance reasons, but most of the duplicates are removed for storage saving. In the end, duplicates of various datasets are tuned in a controlled manner based on their characteristics and analyzed using MapReduce, Hive, and Pig.

It is common for scientific and many business MapReduce jobs to be scheduled in batch like fashion working on the same data. The authors in [108] argue that there is opportunities to share data among such batch jobs which can enable faster completion time. The authors present a new framework, they call MRShare, that transforms a batch query into groups and running each group separately, where the jobs in each group leverage data sharing. The authors use optimization methods to automatically determine such grouping and demonstrate by experiment that their method leads to substantial savings.

To keep the low-level advantages of MapReduce intact and to enjoy higher-level data processing capability (e.g., join, sorting), the authors in [50] present a new framework called Tuple MapReduce. This framework results in better design and implementation of such high level query-type data query programs in MapReduce.

Hadoop was originally designed for fixed inputs, rendering it unusable for stream processing. The authors in [7] present a new MapReduce framework called M3 that is able to support continuous queries on continually arriving data through main memory (i.e., replaces HDFS) and by executing persistent Map and Reduce jobs that do not terminate.

In complicated/high-level tasks, usually SQL-like languages (e.g., Hive, Pig) are used to write the queries that are then translated to a sequence of MR jobs that each produce the data for the next job in the sequence that are read from disk. But, in [46], the authors argue that much of the intermediate data within jobs could be reused to save time for future jobs. To leverage this idea, they present a framework called ReStore that stores intermediate data and sometimes the full output and make them available to later computations. The authors report that they achieve significant speedup on quries from PigMix benchmark.

4.2.2 Replacement Selection

Replacement Selection is a classic sorting technique which has been analyzed since the fifties. Let the RAM is blig enough to store h items (key-value pairs) at a time. First, the author in [51] observes experimentally that sorted runs generated by this method on random data are close to $2h$, but without rigorous modeling. Then, the author in [102] proposes a model that views the run generation process as a plow clearing the falling snow on a circular track, and derives that first run is $(e - 1)h$ items long, while the other runs are around $2h$ items. The author in [79] also derives an accurate combinatorial model for these and shows how the length of each run (except the first) oscillates around $2h$.

Replacement selection with duplicate elimination was first proposed in [112], and later analyzed in [84]. The author in [84] is one of the first few to analyze the uniqueness probability and the seen set size while processing a stream of data, which is mainly synthetic Zipf streams. Interestingly, some of the derived models are similar to ours. The author describes a number of external memory methods (e.g., repeated scanning, repeated union with de-duplication, simple merge sort, replacement selection, and bucket sort) and derives models for each of them. But there are some limitations. First, while analyzing de-duplication-based method (e.g., hash table) he does not consider the overhead associate with the hash table. In addition, the used model for replacement selection is crude and in-accurate. Furthermore, there is no verification on actual/real-world data.

4.3 Random Stream Properties

This section discusses the general architecture of MapReduce, states our main objectives, and derives generic properties of random streams needed later in the chapter.

4.3.1 Terminology and Assumptions

Consider the reduce phase in Fig. 1.1, where the input stream consists of key-value pairs, with keys belonging to a finite set V of size n. Define $d(v)$ to be the number of times v appears in the stream and let $T = \sum_{v \in V} d(v)$ be the stream length. For graph workloads, streams arriving from the mapper commonly consist of links pointing to destination nodes. In such cases, $d(v)$ can be viewed as the in-degree of node v (i.e., how many times it is referenced by other nodes), T as the total number of edges in the graph, and T/n as the average in-degree.

For many real-world data (e.g., user clicks, DNS queries, webpage requests, search in large graphs, queries against a database index, nested MapReduce jobs), the reducer sees keys in random order, i.e., as some stochastic process $\{X_t\}_{t=1}^{T}$, where X_t is the random key, possibly accompanied by some value, in position $t \in \{1, 2, \ldots, T\}$. For simplicity of notation, let random variable D have the same distribution as the in-degree of the system. Then, its PMF (probability mass function) is given by:

$$p_x := P(D = x) = \frac{1}{n} \sum_{v \in V} \mathbf{1}_{d(v)=x}, \tag{4.1}$$

where $\mathbf{1}_A$ is an indicator of event A and $x = 1, 2, \ldots, T$. Note that the output of the reducer always has size n, i.e., one record for each $v \in V$.

We assume that the distribution of D is known a-priori. This knowledge may be available when the stream is queried multiple times for different purposes, or its properties can be predicted from other datasets. Additionally, D may be given by a well-known theoretical distribution in order to understand how the parameters of input affect the resulting behavior of the system. For example, if the in-degree becomes more heavy-tailed (i.e., power-law α gets smaller), does the overhead increase/decrease and by how much? Would doubling memory R have the same effect on runtime in streams with geometric in-degree as with Zipf?

As input pairs are being processed one at a time, let $S_t = \bigcup_{j=1}^{t} X_j$ be the set of keys seen by time t. Similarly, suppose set $U_t = V \setminus S_t$ contains the remaining (i.e., unseen) keys. Unless the workload is explicitly known to exhibit correlated occurrence of keys, it is reasonable to assume that the $d(v)$ copies of v are spread uniformly across the length of the stream. This is known as the *Independent Reference Model* (IRM) [40], [97], which makes the probability to encounter v at time t a constant:

$$P(X_t = v) = \frac{d(v)}{T}. \tag{4.2}$$

Thus, keys with higher frequency/degree are more likely to be seen, irrespective of which portion of the stream is being examined.

4.3.2 Uniqueness Probability

Define $p(t) = P(X_t \in U_{t-1})$ to be the probability that the key arriving at time t is unique (i.e., not encountered before).This is the primary metric of this section as it helps derive other quantities of interest. Defining $\epsilon_t = t/T$ to be the stream fraction, consider the next result.

The probability that the t-th key in the stream X_t has not been previously seen due to [3] is:

$$p(t) \approx \frac{E[D\,(1-\epsilon_t)^{D-1}]}{E[D]}, \tag{4.3}$$

Note that expectations involving D are easily computed by summing up over non-zero PMF values:

$$E[g(D)] = \sum_x g(x)p_x, \tag{4.4}$$

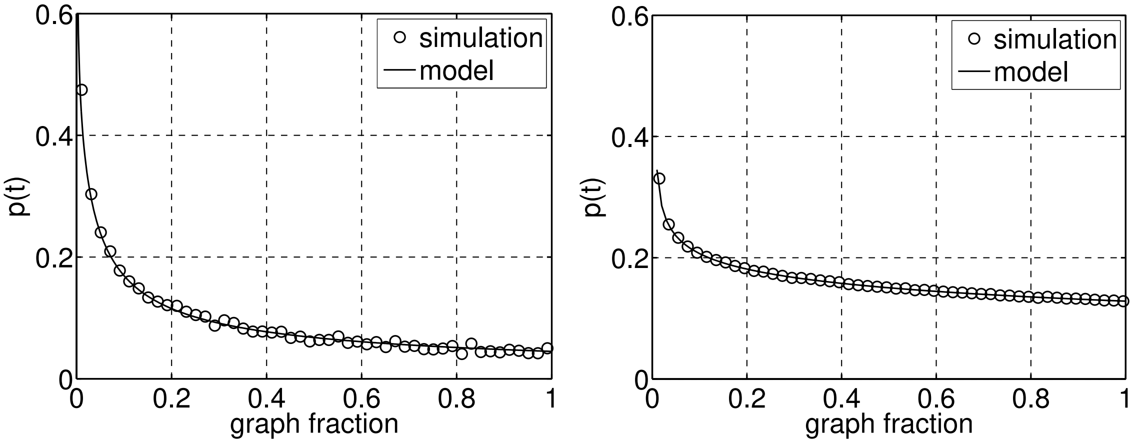
where $g(x)$ can be any function defined on the natural numbers.

Let $s(t)$ be the expected number of unique keys accumulated by time t. From [3], we have:

$$s(t) = \int_0^t p(y)dy \approx nE[1-(1-\epsilon_t)^D]. \tag{4.5}$$

First, we evaluate the $p(t)$ model on two real graph data sets – IRLbot host-level graph [71], [86], and WebBase web graph [131], where both are out-graphs and the crawls for both happened in Jun 2006. The former contains $n = 640$M nodes and $T = 6.8$B edges, occupying 55 GB. The latter contains 530M nodes and 4B edges, consuming 32 GB on disk. Both graphs are represented by adjacency lists ($x_i \to y_{i1}, y_{i2}, \ldots$), where y_{ij} is the j-th out-neighbor of x_i and all node IDs are 64-bit hashes. In the file, neighbors $\{y_{ij}\}$ appear in numerically ascending order and so do source nodes $\{x_i\}$. We use these graphs throughout the chapter for different experiments, in which the destination node of each edge, i.e., labels $\{y_{ij}\}$, are processed by sequentially scanning the graph.

Note that this format is commonly used for storing graphs (e.g., UKWeb [17]); however, due to the sorted nature of neighbor lists, it does not directly satisfy the



(a) IRLbot host graph

(b) WebBase web graph

Figure 4.1: Verification of (4.3) on real graphs.

IRM assumption. Furthermore, the properties of real graphs must be predicted by the model in a single pass rather than after averaging over multiple independent realizations. Thus, our next objective is to understand how much impact these issues have on the precision of the model. Fig. 4.1 shows a comparison between (4.3) and the actual uniqueness probability within 50 bins, each containing 2% of the keys. As it turns out, localized violations of the IRM and single-sample-path constraints do not preclude the model from remaining accurate. Armed with this verification, we next model MapReduce overhead.

4.3.3 Experimental Setup

As mentioned in the introduction, the value field in each key-value pair is assumed to be a scalar. This means pairs (v, a) and (v, b) are reduced using some combiner function $\theta(.)$ to a single tuple $(v, \theta(a, b))$, where $\theta(a, b)$ remains a scalar. To verify the derived models below, we use a simple MapReduce task – computing the in-degree of each node v using the IRLbot host graph/WebBase web graph out-graphs. The mapper emits pairs $(y_{ij}, 1)$ as it processes the edges and the combiner sums up the corresponding values, i.e., $\theta(a, b) = a + b$. Note that keys are 8-byte hashes of node IDs and values are 4-byte counters. For the experiments, we use a hardware platform with an Intel i7 4930K CPU running at 4.4 GHz and 32 GB of RAM.

In addition, we conduct different experiments by allowing varying RAM size for the program to use. To indicate how much RAM is allowed, we use a measure called *Normalized Input Size* (NIS), which is the ratio of input size to the allowed RAM size. Therefore, NIS value of 100 means that the RAM is big enough to fit 1% of the input stream. Note that the more the NIS, the smaller the RAM.

4.4 Basic Load-and-Sort

In this section, we model two basic comparison-sort-based MapReduce designs: merge sort and hash tables. Both these methods first load the data in chunks (due to limited space in RAM), sort them, and then write the sorted runs to the disk. Then, these sorted runs are merged together in the merge step to produce the final result in a single file. Such methods are usually termed *load-and-sort* techniques in existing literature [84].

4.4.1 Preliminaries

Suppose each key-value pair occupies c bytes (i.e., $c = K + D$). Let L be the number of tuples read from/written to disk during deduplication of the entire stream (i.e., the I/O involving only chunks) and R be the size of RAM. The rest of the chapter deals with deriving L for the various algorithms. Considering the T pairs on input, n on output, and L for the read/write during the reduce job, the total amount of I/O is:

$$W = c(T + n + L). \tag{4.6}$$

Note that prior work [13], [80], [90], [144] has sidestepped the issue of analyzing L by introducing a constant $a = L/T$ and converting input size T to that of disk chunks using aT. They also did not consider chunk shrinkage during repeated merge passes, which is an interesting problem in itself. The closes to our work is [84] that modeled L with more success, but in-accurately for a number of methods (e.g., hash table, replacement selection) due to not considering data structure overhead.

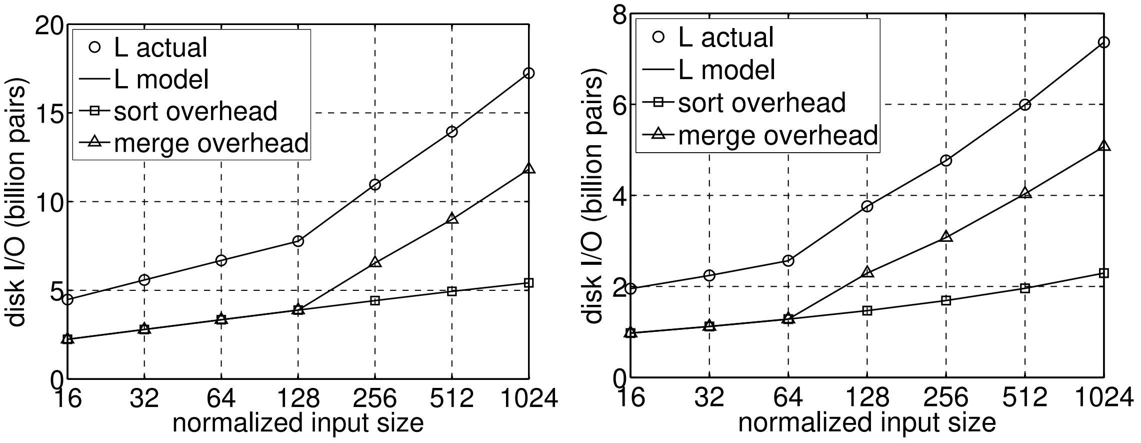
4.4.2 Merge Sort

We utilize a straightforward implementation of disk merge sort. We accumulate input pairs until the RAM is full, sort the result by key, remove duplicates via $\theta(\cdot)$, and write the result to disk as one chunk. This process continues for k cycles until the entire input is processed. After that, all sorted runs are b-way merged and duplicate keys are combined using $\theta(\cdot)$. Define $m = R/c$ to be the number of keys that fit in RAM and assume $T \gg m$ such that $k = \lceil T/m \rceil$ is approximately T/m. We start by handling the simplest case of $k \leq b$.

Theorem 12. *The expected amount of I/O during a single phase of merge-sort deduplication is:*

$$L_1 = 2ks(m) = \frac{2Ts(m)}{m}. \tag{4.7}$$

Things become more complicated if the maximum number of concurrently open files b is smaller than the number of chunks k. Assuming k is a power of b, recall that a b-way merge requires $\log_b k$ phases, each reading the entire output of the previous phase. If L_i is the size of data written during the i-th pass, existing literature universally assumes $L_i = L_1$ leading to $L = L_1 \log_b k$; however, this does not take into account removal of duplicate keys during each phase. We next model the exact



(a) IRLbot host graph

(b) WebBase web graph

Figure 4.2: Verification of disk I/O (4.8) in merge sort.

overhead of this process.

Theorem 13. *If the number of chunks k is a power of b, the expected amount of I/O during a b-way multi-phase merge-sort deduplication is:*

$$L = 2k \sum_{i=1}^{\log_b k} \frac{s(b^{i-1}m)}{b^{i-1}}. \tag{4.8}$$

Proof. Focus on the i-th merge phase and notice that each chunk of this level contains all unique items among b^{i-1} original sorted runs, or equivalently among $b^{i-1}m$ keys of the input stream. Due to the IRM assumption, each intermediate file during phase i has the same number of unique keys on average, i.e., $s(b^{i-1}m)$. Since there are k/b^{i-1} chunks in phase i, its produced data size equals:

$$L_i = 2k \frac{s(b^{i-1}m)}{b^{i-1}}. \tag{4.9}$$

Then, summing up across all $\log_b k$ levels, we get (4.8). □

Algorithm 1 Merge-Overhead Computation Algorithm

```
procedure MERGE-OVERHEAD(stream-lengths $M[1, 2, \ldots, k]$ of sorted runs)
    $L \leftarrow 0$
    while $k > 1$ do
        Sort the array $M[1, \ldots, k]$ in ascending order
        if $k > 2b$ then
            $\tau \leftarrow b$
        else
            $\tau \leftarrow k - (b - 1)$
        end if
        $M[1 : \tau] \leftarrow \sum_{i=1}^{\tau} M[i]$                ▷ combine the first $\tau$ entries
        $L \leftarrow L + s\left(\sum_{i=1}^{\tau} M[i]\right)$
        $k \leftarrow k - \tau + 1$
    end while
    return $L$
end procedure
```

On the other hand, if k is not a power of b, Algorithm 1 chooses the optimal merge order of the runs based on their lengths to keep the overhead to minimum and also computes that overhead.

Fig. 4.2 compares (4.8) against actual measurements on the two graphs and confirms the result is quite accurate in both test graphs. We also include the amount of disk I/O separately during the sort and the merge phases. Observe that the disk I/O during the sort phase grows sub-linearly with reducing memory size in the entire range. On the other hand, the merge overhead is exactly same as that of sort up to a threshold (i.e., ratio 128 for part (a) and 64 for part (b)), beyond which there is a linear growth of overhead with shrinking RAM. This happens due to higher number of sorted runs with less volume, leading to multiple passes required for the merge step to complete. Beyond that point, merge overhead dominates the overall cost.

Also, note that the relationship between overhead L (even for single-pass merge) and memory size m is highly non-linear due to the term $s(m)$, which requires summing up $(1 - m/T)^x p_x$ over the entire PMF $\{p_x\}$.

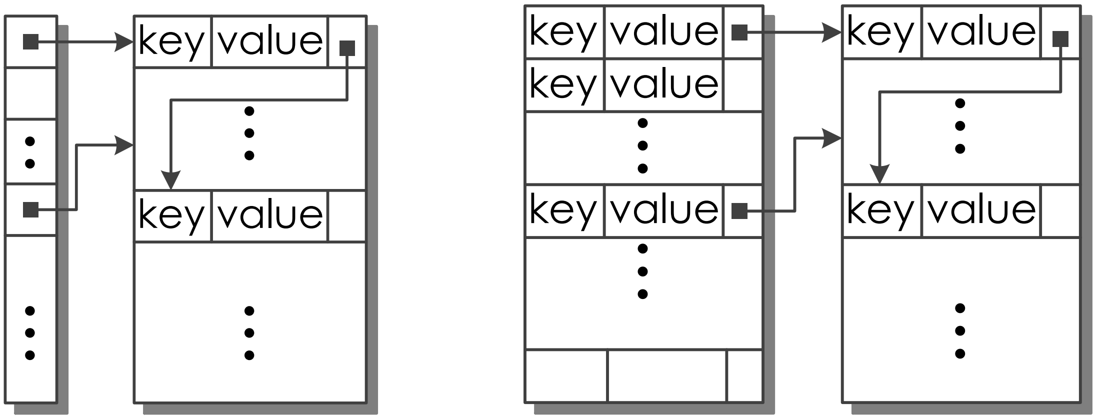
4.4.3 Hash Tables

Since hash tables can be efficient at sorting data, especially for highly redundant input, we next explore their usage in MapReduce. We use terms “key” and “hash” interchangeably, assuming that keys have been already hashed (e.g., as commonly done when storing large graphs). Hash-table (HT) deduplication works as follows. First, keys and their values are read from disk and sequentially inserted into a hash table. If the key already exists, the new value is combined with the existing one. Once the HT is full, all key-value pairs are written to disk in sorted order and the hash table is emptied. The process continues until all input data are processed. The only requirement on the hash table is that it supports sorted-order serialization, which many traditional designs do.

The capacity of a hash table depends on its additional data structures (e.g., overflow chain pointers, empty bins due to over allocation). Assume that the HT adds overhead $\mathcal{O}$ bytes per key. Therefore, RAM of size R can contain at most $h = R/(c + \mathcal{O})$ keys and their values. While we know that each sorted run contains exactly h unique keys, we need to map this value to the number of items m in the stream that became compressed to that size. Our next result shows that all disk chunks fall under (4.3) and (4.5), not just the very first one.

Theorem 14. *Suppose the hash table is emptied at time* t_0*. At time* $t \geq t_0$*, the probability to encounter a unique key is* $p(t - t_0)$ *and the expected seen-set size is* $s(t - t_0)$*.*

Applying Theorem 14, we obtain that $m = s^{-1}(h)$ keys are needed to make the HT full in every sorted run. The inverse exists due to monotonicity of s, but requires numerical methods unless D follows a well-known analytical distribution. Since the merge operation here is identical to that in the previous section, the overhead remains



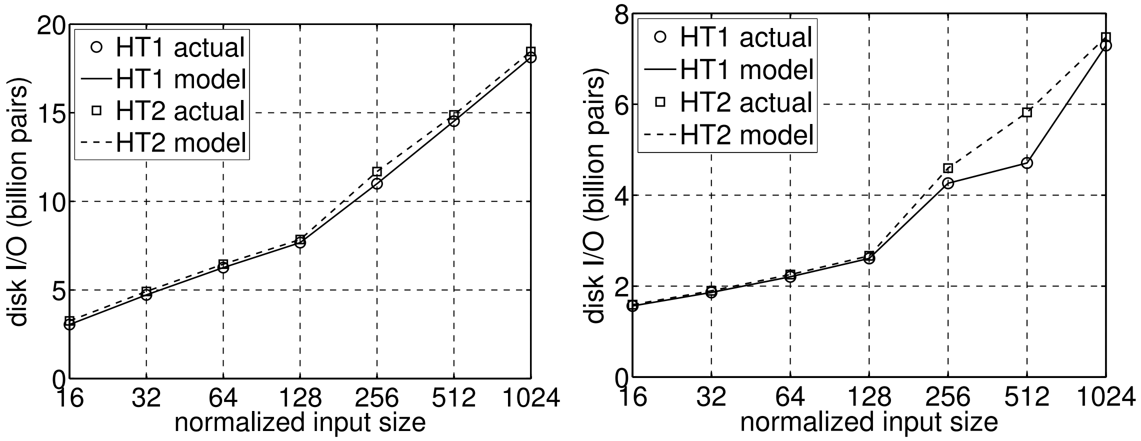
(a) HT1 (b) HT2

Figure 4.3: Two different hash table data structures.

the same as (4.7) and (4.8), with m replaced by its HT equivalent. While L_1 admits a slight simplification to $Th/s^{-1}(h)$, the L metric unfortunately does not.

We test this model in two custom hash-table implementations that both use two arrays. In the first design shown in Fig. 4.3(a), which we call *HT1*, we use B bucket pointers implemented as 4-byte offsets to the second array, where the actual key/value pairs reside, each along with another 4-byte offset to build up the collision chain inside a bucket. In Fig. 4.3(a), the first bucket contains 2 pairs, while the second one contains none. Lookups in HT1 involve a linear search within each chain, which can be made faster by having more bins and thus shorter chains. But this leads to more memory usage and consequently more disk I/O due to shorter runs. For verification purposes, we have determined empirically that a reasonable balance between speed and I/O is achieved when $B = h$. This leads to average chain depth of 1 for HT1, resulting in overhead $\mathcal{O} = 8$.

In the second design shown in Fig. 4.3(b) and termed *HT2* here, the bucket pointers themselves contain space for one key/value pair each. Thus, the right side only contains bucket chains of depth more than 1. This gives better caching perfor-



(a) IRLbot host graph

(b) WebBase web graph

Figure 4.4: Disk I/O for sort and merge phases in HT MapReduce.

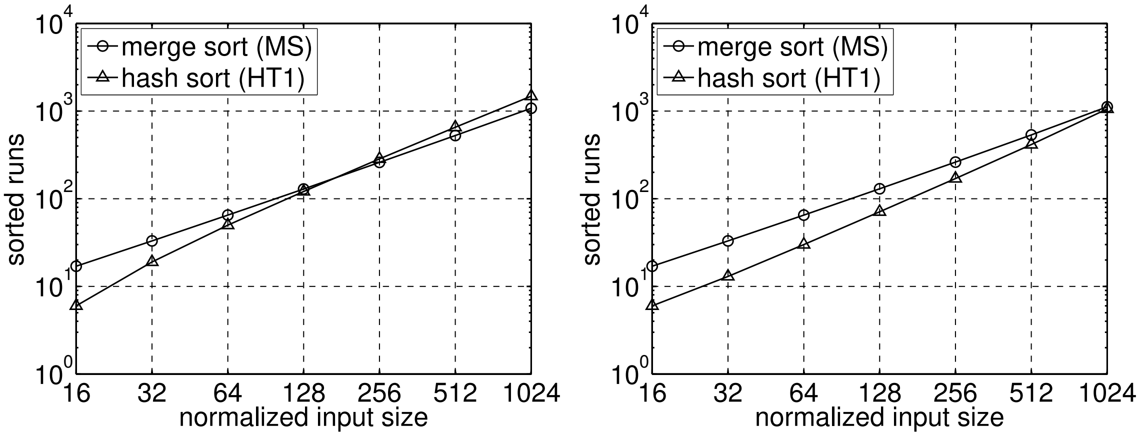
mance and higher lookup speed. In Fig. 4.3(b), the first bucket contains 3 pairs – one in the bucket item, and two more in data items. The second bucket contains only one pair, which stays in the first array. Finally, the last bucket in the figure is empty, which is indicated by a special symbol in the key field.

Since $e^{-h/B}$ fraction of bins are empty, we have

$$\mathcal{O} = (c+4)e^{-h/B} + 4 \tag{4.10}$$

bytes of overhead per key. The average chain depth for HT2 is $1/(1-e^{-1}) \approx 1.58$ leading to $\mathcal{O} \approx 9.92$ bytes/key.

We run the HT MapReduce on our two graphs and show the results in Fig. 4.4, which demonstrates that the derived model is accurate for both HT1 and HT2. Note that the disk I/O of HT2 is slightly higher for smaller RAM sizes compared to HT1. But the advantage of HT2 is its much faster lookup/insertion speed (i.e., 30M pairs per second) compared to 20M/s for HT1, making HT2 a more practical alternative



(a) IRLbot host graph

(b) WebBase web graph

Figure 4.5: Number of sorted runs in merge sort and hash table designs.

for high performance application.

Then, in Fig. 4.5 we compare the number of sorted runs generate by the merge sort and hash sort for the two graphs. Note that HT produces less number of sorted runs $k = T/s^{-1}(h)$ than MS, even though the HT can fit fewer items in RAM. This ability of the hash table to deduplicate more data items tend to diminish as the RAM size shrinks, leading to fewer items in the RAM. As a result, the overhead of the hash table becomes more prominent. This effect is visible for IRLbot host graph, where the number of runs in HT surpass those in MS beyond the point 128. For WebBase web graph, this crossing point seems to be at 1024.

4.5 Replacement Selection

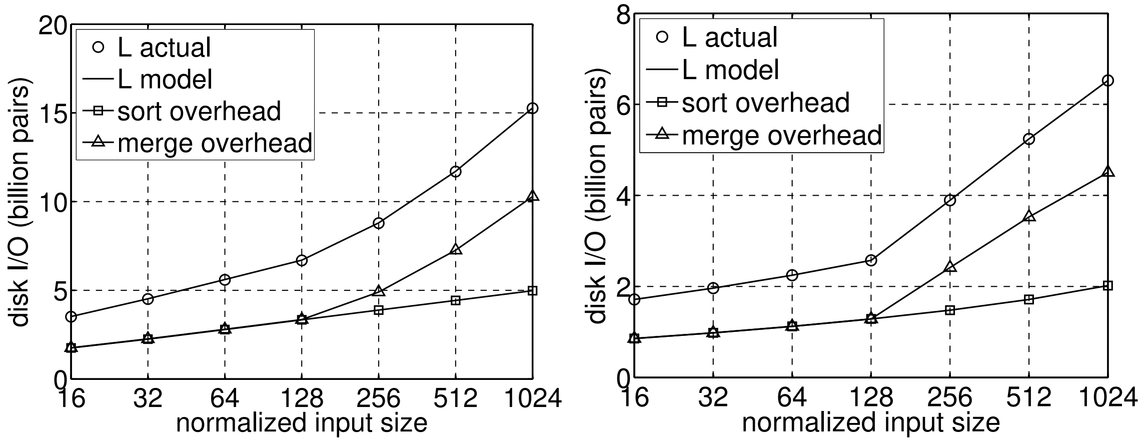
Many external memory program (e.g., databases, MapReduce jobs) can benefit from less number of sorted runs because this leads to less amount of processing and I/O activity for the subsequent merge step. In addition, due to memory limitations on the number of concurrently open files required in a merge, less sorted runs may result in reduction of the number of passes. A classical technique for increasing run length is called replacement selection, which was originally proposed for all-unique keys. The run-length of replacement selection was analyzed by [47], [51], [79], [102] all primarily focusing on all-unique data item (i.e., keys either do not repeat, or repeat with very small probability). The author in [112] first proposed replacement selection for handling duplicate keys, and that in [84] presented a crude but inaccurate model for such method. In this section, we present an accurate model for the intermediate data produced by the classical two-queue implementation of replacement selection method.

4.5.1 Traditional Two Queue Implementation

Replacement selection has been implemented in a number of different ways. The most commonly used method is maintaining two priority queues of data items, where the items in the first queue are part of the "current run" and those in the second are of the "next run". The first one is initially filled (i.e., occupies the RAM completely) with data from the input, and the other one is kept empty. After that, the smallest item from the first queue is popped and output to disk.

Now, an item from the input is read and compared with the last-popped item. If it is larger than the last-popped item, it takes the empty space in the first queue. Otherwise, it goes to the second queue to become part of the next run. In this way, the second queue grows and the first queue shrinks. Once the second queue is full (makes the RAM full), the current run is closed, the next run is opened in the disk, the queues change roles (the second queue become the first, and viceversa), and the process continues until the input is exhausted.

Since a typical MapReduce workload contains keys that are repeated, we change the above design by adding a one-pair buffer between the first queue and the disk. Whenever an item is popped from the queue, rather than writing it directly to disk,



(a) IRLbot host graph

(b) WebBase web graph

Figure 4.6: Verification of Theorem 15.

it is always attempted to combine with the pair in the buffer (the value fields are combined when the keys match). Otherwise, the buffer item is sent out to disk and the popped item takes its place. In this way, all the pairs in the queue with duplicate keys are combined together and the output shrinks. This idea was first proposed in [112].

4.5.2 Disk I/O in Replacement Selection

The following theorem quantifies the number of sorted runs and the amount of disk I/O for the two-queue replacement selection.

Theorem 15. *The expected number of sorted runs in replacement selection is* $k = (T - (e-1)m)/(2m) + 1$*. When* k *is a power of* b*, the amount of disk I/O is:*

$$L = 2k \sum_{i=1}^{\log_b k} \frac{s(b^{i-1}2m)}{b^{i-1}}. \tag{4.11}$$

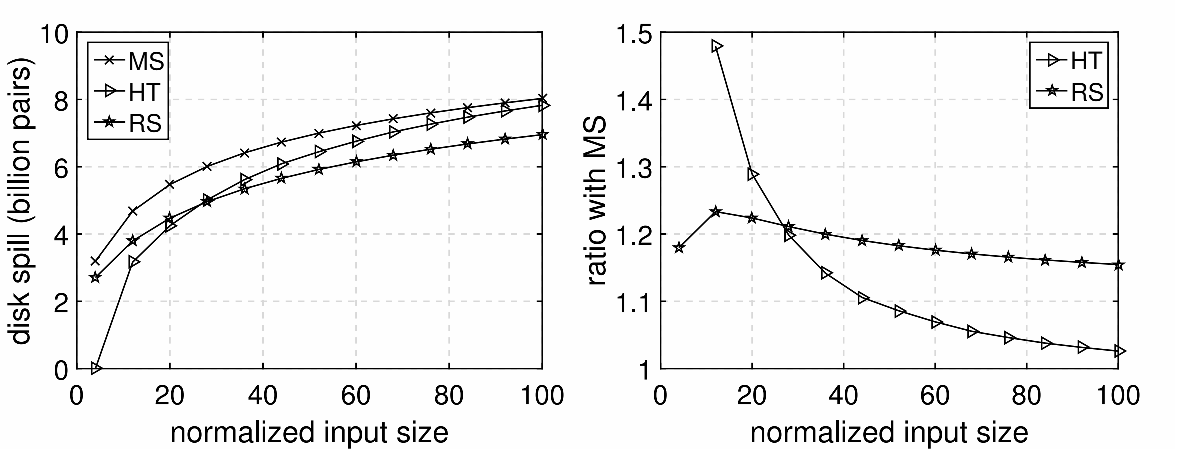
If k is not a power of b, the sort-overhead is:

$$L_s = s((e-1)m) + (k-1)s(2m), \tag{4.12}$$

and merge-overhead L_m follows from Algorithm 1, and the total disk I/O is $L = L_s + L_m$.

Proof. Due to [47], [79], we know that the first sorted run from a replacement selection method contains $(e-1)m$ pairs, while the other runs contain $2m$ items before de-duplication. As a result, there is a total of $k = (T-(e-1)m)/(2m) + 1$ sorted runs in total. Now, when k is a power of b in a b-way merge, we can derive (4.11) in the same way as of (4.8) with the only difference being that each sorted run now span $2m$ instead of m. If, on the other hand, k is not a power of b, we compute sort overhead L_s and L_m separately. Since we de-duplicate and combine the data items before writing to disk, the number of pairs in the first and the other runs become $s((e-1)m)$ and $s(2m)$, respectively. As a result, the total number of pairs written to disk after the sort phase is $L_s = s((e-1)m) + (k-1)s(2m)$ in k sorted runs. Now, the merge overhead L_m of these requires invocation of Algorithm 1, where we feed the k sorted runs and obtain the result. $\square$

We verify the derived disk I/O model in Theorem 15 in Fig. 4.6 for the two graphs, which shows that the L model in Theorem 15 is very accurate in both cases. Similar to merge sort, we have also included the overhead for the sort and the merge phases separately. Note that merge overhead is still the dominant factor for smaller RAM sizes for both graphs. Also observe for WebBase web graph that the more rapid growth of the merge overhead starts at point 256 and onwards, where this starts at point 128 in case of merge sort for the same graph (see Fig. 4.2(b)). This, along



(a) absolute

(b) normalized by MS

Figure 4.7: Comparison of disk I/O various MapReduce designs with respect to merge sort.

with lower values of disk I/O in other cases, indicates the effectiveness of replacement selection in saving disk I/O.

4.6 Discussion

In this section, we present a performance comparison among the above MapReduce designs. Then, we discuss how the presented models can be used in multithreaded and parallel MapReduce systems.

4.6.1 Comparisons

We compare various MapReduce techniques in consideration of their resource disk I/O, which is an important metric in determining their runtime. To enable a fair comparison, we restrict each method with the same amount of memory, and then vary the memory size. We consider that each key is $K = 8$ bytes and the values are each $D = 4$ bytes, and together they take $c = K + D = 12$ bytes. In addition, to obtain high disk I/O performance, we use $w = 4$MB file buffer for each open file. Then, different methods require different number of open files, which we denote by

f. For example, MS and HT both require at least one open file (the input file, and occasionally the output file to write the sorted data). Therefore, we use $f = 1$ for these cases. On the other hand, RS requires two open files – one for the input and the other for the output.

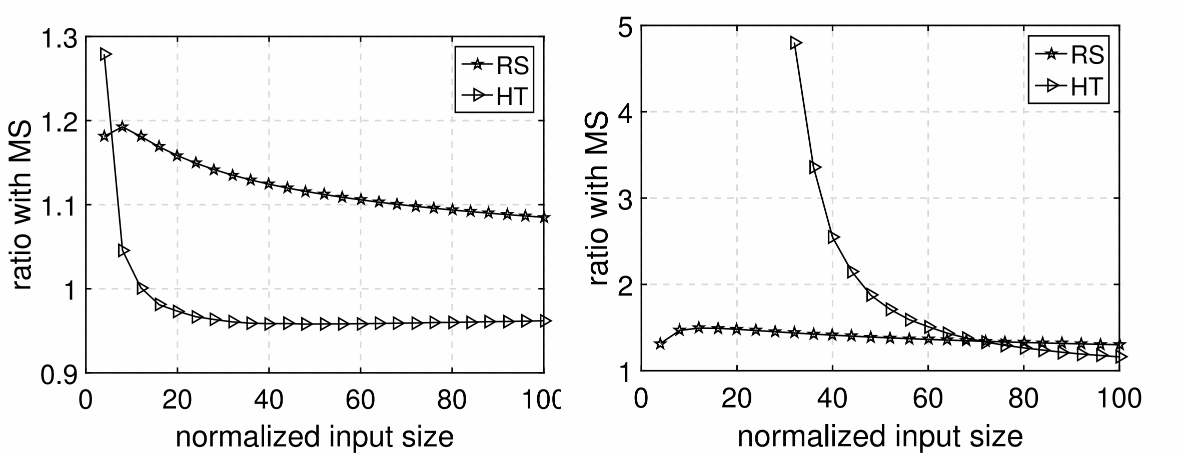
We use the following relation during the sort phase to calculate the number of items that each of the above methods can load in RAM and sort at a time :

$$h = \frac{R - fw}{c + \mathcal{O}}. \tag{4.13}$$

As an example, merge sort method stores pairs in an array and thus does not require any overhead per pair (i.e., $\mathcal{O} = 0$). The same is true for replacement selection based on priority queue, since the tree structure formed in a priority queue is implicit from the indices in the array of data items, and thus does not require extra overhead. On the other hand, HT uses an overhead of $\mathcal{O} = 8$ bytes per pair, since we use HT1 as the underlying implementation for both.

First, we examine the disk I/O of the four MapReduce methods described above together in Fig. 4.7(a) to enable comparison among them. Observe that the RS method generates the least amount of intermediate data, while MS produces the highest in the entire observation range. Things are little complicated for HT and RS, where HT starts out better than RS, but after the crossing point at around point 25, RS starts performing less I/O than HT. In addition, observe that both RS and MS starts out at the same spot, while HT starts at a lower point. The difference is mainly due to no-overhead of RS and MS compared to 8-bytes overhead of HT.

Observe that HT and MS tend to converge towards each other. The reason is as the RAM gets smaller, the de-duplication ability of HT starts diminishing, while because of their no-overhead, RS and MS start catching up. Then, Fig. 4.7(b) shows



(a) $E[D] = 5$ (b) $E[D] = 50$

Figure 4.8: Comparison on synthetic Zipf ($\alpha = 1.5$) graphs with $T = 5$M.

the same results, but this time normalized by the disk I/O of merge sort.

To investigate the behavior of these methods more closely, we examine them on two synthetic Zipf graphs, both with 5M edges, $\alpha = 1.5$, and average degree being 5 and 50 respectively. The results are shown in Fig. 4.8, where we compare the performance of other methods using MS as the baseline. Observer in Fig. 4.8(a) that when the repetition is less (average degree 5), IHT's deduplication ability reduces to the point where its overhead becomes noticeable. As a result it falls below the RS that does not have any overhead. Similarly, HT also looses to MS (ratio being less than 1) because of the same effect. But the results change a little bit when the average degree is 50, which is shown in Fig. 4.8(b). Observe that RS maintains nearly 1.5 times less I/O compared to MS for different RAM sizes. The I/O for HT is also less than that of MS (ratio being more than 1). This confirms that fact that using either HT or RS is more advantageous in cases where the nodes are repeated more.

Furthermore, to examine the practicality of these methods, we run each of these

Table 4.1: Processing speed of different data structures.

method	MS	HT	RS
speed (keys/sec)	9M	20M	3M

methods on the first 1B edges of IRLbot host graph, record the lookups/insertion rates, and collect the results in Table 4.1. Note that while calculating speed, we include the time for interaction with the disk (i.e., read time and write time), and resetting the data structures when a run ends (i.e., time to re-initialize a hash table after the content is written to disk). First, observe that RS is slowest among all, while HT is the fastest one. This somewhat explains why although being I/O efficient, RS has not been used in many systems.

4.6.2 Multi-Core MapReduce

First, consider a single-threaded hash table MapReduce program as described in Sec. 4.3, which processes T pairs and produces k sorted runs $Z_1, Z_2, \ldots, Z_k$. Now, to parallelize this program to multiple threads on a machine with p CPU cores, we can have p hash tables and run them independently from p cores. The input pairs are split into disjoint intervals of the keys such that each core gets roughly the same number of keys and all keys in core i are less than those in core $i+1$ for $i = 1, \ldots, p-1$. The structure is shown in Fig. 4.9, where the keys fall in range $[0, N-1]$. Then, hash table i gets all keys in range $[a_i, b_i)$, where $a_i = (i-1)N/p$ and $b_i = iN/p$ for $i = 1, \ldots, p$. Here, we assume that the keys are uniform in the rage $[0, N)$ which is reasonable since the actual keys are replaced by their hashes.

Now, to maintain the same memory usage as the single threaded version, each hash table is allocated only R/p memory (recall that the main memory is R bytes). As a result, their capacity drop by a factor of p. At the same time, while the single

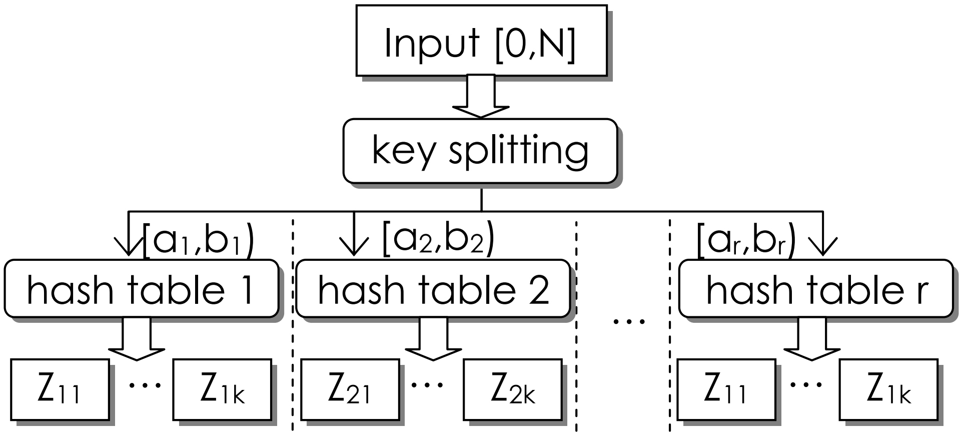


Figure 4.9: Multi-Core hash table MapReduce

threaded hash table had to process T keys, each hash table in the multi-threaded version processes only T/p pairs. Recall that hashing rate is η. Since, p threads are working in parallel, hashing time reduces to $T/(p\eta)$ sec.

At the end, each hash table i produces roughly the same number k of sorted runs $Z_{i1}, \ldots, Z_{ik}$, where the original sorted runs from the single thread $Z_i = \bigcup_{j=1}^{p} Z_{ji}$ for $i = 1, 2, \ldots, k$. Note that there are roughly p times more sorted runs in this case, but each sorted run is smaller by a factor of p. Thus, the number of spilled pairs stays the same as before (i.e., L_s).

Merge sort MapReduce can be adapted to multi-threaded version in a similar fashion. The input data are split into p disjoint intervals in the same way and keys of different intervals can be accumulated in separate arrays. After one/all of them is full (depending on the strategy), all the arrays can be sorted and de-duplicated in parallel. Thus, sorting speed again scales linearly with the number of available cores p.

4.6.3 Cluster MapReduce

In a cluster MapReduce, parallelism is provided by multiple (p in this case) computers connected to some LAN or even over Internet. We assume that the connectivity between the machines are homogeneous with a rate ν bytes/sec for both read and write. In addition, key-splitting happens on a node separate from the compute nodes. Since the keys destined for a particular machine have to travel the network, the distribution process now takes a non-negligible takes an Tc/ν seconds to finish.

Once keys reach their corresponding hosts/machines, they can be hashed locally by a single-threaded hash tables, one for each node. Following the arguments for a multi-core MapReduce, we can infer that the total I/O amount stays the same. But the input data are read from the network and then written to the local disk of each node. If the end result is needed in a single file, merged data from each node are accumulated in a separate node by sending the whole output through the network again, which takes an extra nc/ν seconds.

4.7 Conclusion

In this chapter, we propose analytical models of disk I/O for a number of well-known external-memory MapReduce designs. These models are useful for forecasting resource (disk and main memory) usage without the help of expert knowledge and ad-hoc parameter choices. We also show how the models are easily adapted for more complicated multi-core and multi-node systems. However, note that these models apply to a class of MapReduce problems where the values are scalars before and after the combination operation. Furthermore, either the popularity distribution of the keys has to be known beforehand, or can be computed during the first MapReduce job when multiple of queries/results are required from the same data set. Potential future directions include examination of a number of other MapReduce designs and cluster MapReduce under different network topologies.

5. ADVANCED EXTERNAL MEMORY ALGORITHMS

5.1 Introduction

Caching and sorting are ubiquitous in information processing and data analytics. They are also integral parts of MapReduce and different external-memory algorithms. In the context of the previous chapter, where we presented a number of traditional external sorting techniques (e.g., merge sort, hash sort, replacement selection), in this chapter, we move on to a number of advanced techniques that are faster and produces less disk I/O, both of which are desirable while processing large scale data.

In this chapter, we propose a new MapReduce method based on Incremental Hash Table (IHT). This new method borrows its idea from traditional hash table sorting and works similar to the classical replacement selection method, which has the ability to produce longer sorted runs. We show that this method is much faster than the traditional replacement selection with all its advantages. We also present analytic model for its performance (i.e., its hit rate, length and number of sorted runs) and demonstrate that these models are accurate on both synthetic and real data sets. We also present a much simpler model for the steady-state of the IHT, which makes the model even more practical. In addition, we demonstrate that how the working principal of IHT and Random Replacement (RND) caches are identical, making all these models applicable to this widely-used caching method.

Then, we draw our attention to Bucket Sort, which recursively splits the input into RAM-sized splits based on either radix or division and sort those smaller chunk. Since there a is intra-bucket ordering inherent in the splitting process, no merging is required at the end of sorting. In addition, it is well-known that bucket sort can advantage greatly with use of cache while splitting, which reduces the number of input pairs going to and from the disk. Therefore, to analyze bucket sort, we start by investigating existing caching techniques, their performance and memory overhead. In the process, we propose a novel caching method, we call WATCH, that shows its advantages in both per-key space overhead and its high speed performance. We also propose accurate analytic model for its hit rate. Then, we show how this cache helps a bucket sort routine to achieve very high performance compared to other external sorting methods (e.g., hash sort, merge sort).

In addition, we propose accurate models for the conventional bucket sort (with a fixed split factor) and improve that by introducing an optimized bucket sort method where the split factor can change based on the workload characteristics and available resources. We demonstrate that this new method produces the least amount of disk I/O compared to other MapReduce methods including IHT and the ones presented in the previous chapter.

5.2 Related Work

The related work spans mainly two fields – modern MapReduce and external sorting algorithms that we improve upon, and different caching techniques relevant in our space-efficient caching analysis. Please see the previous chapter (i.e., Chapter 4) for more detailed related work regarding existing MapReduce and other external-memory methods. Since the performance of such methods is controlled mainly by the sorting kernel, we also discuss a number of high-performance external sorting techniques that we see relevant.

5.2.1 Caching

There has been much work [26], [40], [52], [73] on deriving the cache miss rate under various replacement policies (e.g., LRU, LFU, random, FIFO) and specific distributions of key frequency. For example, [73] provides an asymptotic analysis of

LRU for Zipf and Weibull distributions. Another set of results [26] relates the to the miss rate of individual items. In contrast, our LRU model captures the overall miss rate across all available keys and further generalizes [26] using certain intermediate results. Furthermore, our model is not limited to any specific distribution of item frequency.

In [62], the authors present a method of computing output cardinality of a pre-aggregation operator as a function of the buffer size and the clustered-ness of the data under random replacement strategy. In this method, the clusteredness is defined as the average number of changes in keys in between two successive keys of the same value. The authors then propose a series of refinements to their original estimate and then evaluate their final estimate on both synthetic and real data. The experiments suggest that their model is closer on synthetic data, while being less accurate on real data. This is an important method for such data streams, since it incorporates the clusteredness/arrangement order of the data with their frequency distribution. But, their clusteredness measure is not directly related to correlation of the data. In addition, they have to use a replacement strategy for their method to work. Thus, this method is inapplicable to pre-aggregation scenarios where the main memory is large enough to hold all data.

There are a number of problems with the original LRU. First, it only takes into account the recency of a page, not its frequency. As a result, if an important and frequent page is not referenced during certain interval, that page is dropped. Another, problem corresponds to the implementation of LRU, where the movement of a hit page to the front of the recency list requires a global lock in the system. In high-performance applications (e.g., virtual memory manager), this becomes a system bottleneck reducing concurrency in the system. Third, LRU is not scan-resistant, because a simple scan through distinct requested objects will remove all history maintained by a LRU cache and thus pollute it. Over all, LRU is a very static caching scheme with some of its advantages. But it is not adaptive to changes in access patterns. To solve these problems with LRU, there have many proposals to improve its performance.

In [109], the authors describe a new caching algorithm based on LRU, which they call LRU-K, where the history of $K \geq 2$ most recent references to an object is retained whose difference gives inter-arrival time, leading to the estimated probability of hitting that object again. For instance, when $K = 2$ and the cache is full, the item for eviction is the one whose interval between the last two references are the highest. In case, when the last two references are not known, the inter-arrival distance is assumed to be ∞ and thus the probability is 0, leading to eviction of the item. As a result, this scheme is the first proposal to obtain data-dependent hit probability for an item while still maintaining LRU policy. In addition, the authors also discuss a number of issues that need consideration for its correct implementation. For instance, if an item is referenced twice or more in short interval, but not referenced in the future, then unmodified LRU-K will retain it for a long time. To solve this problem, the authors introduce a concept called *Correlated Reference Period* within which the distinct references to the same page are considered as just one. In addition, the authors discuss another problem where a page is referenced once in a while, but evicted every time due to no further information about past references are not available. Thus, the references to such pages are lost always. To solve this problem, the authors introduce another time period called *Retained Information Period* during which, the information about an evicted page is retained in the anticipation that it will accessed again in the future. Thus, the cache still evicts an item, but does not loose valuation history information about it. Using experiments on real data, the authors demonstrated that LRU-2 significantly improves upon LRU, but with higher

K, there is a diminishing return on the value and wasted performance.

Now, the LRU-K solves the problem of frequency, but introduces another problem for its implementation – it needs a priority queue for deciding the next-to-evict item. As a result, the performance of LRU-K can be bottle-necked by the queue-management. To solve this problem, the authors in [78] present a new algorithm where there are two separate queues to hold the request. The first one works in FIFO fashion and is supposed to contain the cold pages until they are evicted. The second one is the repository of the hot pages that are promoted from the first one after being hit again while being there. The items in the first queue get evicted after being not hit before arriving at the tail. But the authors propose to maintain a some information about the pages that are delete so that their history is not lost. Therefore, ultimately there are three queues in this design. The size of all these queues are tunable parameters. The authors show that this new design, despite requiring manual tuning, shows better performance than the conventional LRU and comparable to that of LRU-2.

To solve the performance problem regarding a global lock of LRU and its decedents (e.g., LRU-2, 2Q), the author in [39] presents an improved implementation of LRU, called CLOCK where there is clock-hand like pointer that traverses over the items, each time resetting the reference bit of that item, which is initially set to 1 when it was first accessed. When full and on a miss event, the clock-hand scans through the items in the cache for an item whose reference bit is already 0. In the worst case, the clock hand has to come back the original item where it started its search, which will then be discarded. This new design solves the performance problem of LRU, shows equivalent hit rate as LRU [23], [33], [125], and thus becomes a very good alternative for the cases where LRU's performance was initially unacceptable (e.g., virtual memory manager).

There have been a number proposed improvements over the CLOCK caching scheme – CLOCK-pro [75], GCLOCK [125]. In addition, the proposed improvement over LRU are LIRS [76], FBR [117], LRFU [85], [88] and MQ [146]. The problem with most of the schemes is that they require setting a number of crucial parameters which ultimately control their hit rate. Contrary to these, the two most notable and recent proposals are: ARC [98] based on LRU, and CAR [11] based on CLOCK. The good feature about these two is that they are auto-tuning during runtime and thus no ad-hoc parameter setting is necessary. But there are disadvantages too. Like some of the older methods, they also require retaining some information about the deleted pages. Although in some applications it does not incur much overhead, for many others, the meta data overhead is almost as much as the original key, making the effective cache size much larger (sometime twice). Therefore, none of CAR and ARC are suitable for MapReduce like programs.

5.2.2 High-Performance Sorting

The authors in [31] this paper introduce an algorithm for *parallelizing in-place radix sort.* Their objective is to solve two problems: read-write dependency inherent in the in-place nature and load-balancing required in skewed data. To solve the first problem, the authors proposed a scheme where each processor works a separate part of the input, and each bucket has stripe allocated to each processor. Therefore, the in-place radix sort can work in parallel and independently. But, the problem is is, stripe sizes are not know beforehand and therefore, the above permutation is not accurate. To correct it, there is another phase that pushes the in-correctly assigned items in each bucket towards the end. After that, the above permutation and correct happen in iterations until there is no misclassified items in any bucket.

To solve the load imbalance problem arising in skewed data sets, the authors propose a method of assigning a sub-set of buckets to each processors. They also formulate the load-balancing problem and justify their solution rigorously. In the end, they compare their sorting method gainst a number of other sorting methods (e.g., parallel introsort, parallel hybrid sort which is combination of multi-way merge sort and quick sort, parallel quick sort, GPU radix sort, SIMD parallel merge sort) and existing methods of radix sort (e.g., Buffsort, radix-ax, radix-se, radix-ip). The authors use numeric (8-byte key and 8-byte payload) and string (10-byte key and 90-byte payload) workloads with controlled randomness of data to generate both random and skewed input using **gensort from sortbenchmark.org**. They obtained 160M records/sec using 16 threads on the numeric benchmark and 18 M records/sec on the string benchmark. Overall, they were close to 2.13 GB/sec.

In [118], the authors propose novel methods for efficient parallelization of the prominent radix sort and merge sort, on both SIMD CPUs and GPUs. Their algorithms are 2X faster than the existing results. In the process, the authors identified the bottlenecks (memory, compute, and latency characteristics) of these methods on different platform using their precise analysis of required cycles/elements. They also showed how the designs of these methods differ depending on the architecture. For instance, they proposed a *software managed buffer* sized in a hardware-conscious manner for radix sort with the objective of reducing both capacity and conflict misses in the cache, where the previous methods ignored the effect of conflict misses. This resulted in the fastest known CPU radix sort. On the other hand, they used a *local sort based* scheme for their GPU radix sort.

After that, the authors proposed a novel approach for merge sort that uses *bitonic merge network* for both the sort and merge phases. This merge network is designed to exploit SIMD features efficiently. They also modified the best known method for GPU merge sort to use the bitonic merger network and obtained the fastest GPU merge sort in the process. They also presented accurate counts of different CPU level instructions (read, add, write) for all the above methods and verified that those counts are very accurate from actual trace. Over all, they found that radix sort is memory bound, while the merge sort compute bound.

They evaluated their methods on a workload containing only 4-byte keys (no payload) on two platforms: 3.2GHz core i7 (quad-core) CPU and NVIDIA GTX 280 GPU running at 1.3GHz. Experiments on the CPU methods show that radix sort performance stays pretty stable at 240**M keys/sec** (60M/s per core), while the performance of merge sort drops from the peak 250**M keys/sec** to 150**M keys/sec** with varying input size. The authors attribute the effect on the complexity of these sorting methods (i.e., $O(N)$ and $O(N \log N)$, respectively). Similar trend is true for the GPU methods as well, where the peak for radix and merge sort are 220M/s and 200M/s, respectively. The authors also examined the effect of increasing key size and adding payloads. Due to memory-bound nature of radix sort, it experiences nearly 3 times slower performance for 8-byte keys (20M/s/core). On the other hand, the merge sort experiences less slow down (i.e., 2 times) because of its compute-bound nature.

The authors also tested their methods on some future simulated architectures and showed that the compute-bound merge-sort can take more advantage of the wider SIMD platforms and will scale better than radix sort in the future.

Radix partitioning is a memory intensive operation due to its random nature of memory access pattern. There have been a number of recent techniques to speed up this process. The paper [119] discusses all these techniques by adding them incrementally to the basic partitioning and investigate their relative advantages. The authors mainly consider software-managed buffers, non-temporal streaming stores, pre-fetching, micro row layouts, and huge pages, and investigate their effect on various number of partitions. All experiments are run on a single-threaded implementation on an Intel Xeon 2.2 GHz processor for sorting 100M key-value pairs, where the key and the value are each 4 bytes. Note that depending on the partition numbers (split factor), their original version (without addition of any techniques) was varying between 127M/s for 32 partitions to 27M/s for 16K partitions.

The first technique, software-managed buffer, is targeted to reduce misses in both the data cache and the TLB. In this technique *b* entries per partition are kept in a small cache-resident buffer, where a whole partition is evicted at once when that partition becomes full with *b* entries. As a result, the output memory locations are accessed only at buffer granularity. Larger *b* would lead to more TLB saving, but more data cache misses as well, which leads to a tradeoff. With this software-managed buffer and manually tuning the correct buffer size, the authors observed at most 2.25x speedup.

Non-temporal streaming store is a way of storing data to memory without triggering a caching of that region of memory. This is done because the written/partitioned data will not be used in the near future, thus violating the temporal locality rule that applies in other cases. The following AVX intrinsic instruction can be used for this purpose:

```
_mm256_stream_si256(__m256i*mem,__m256ia)
```

As a result, the processor tries to perform *write combining* which is also useful. The experiments show that this technique achieves at most 1.36x speedup and the advantages are more prominent with larger number of partitions.

Writing buffer contents to memory resembles random accesses that can still cause cache misses. To hide the latency due to these misses, the CPU can be instructed to pre-fetch portion of the memory before they are actually accessed. The following instruction can achieve this:

`_builtin_prefetch(void*mem,intrw,intl)`

Experiments suggest that this method can indeed hide much of the latency that higher buffer sizes show without this and the pre-fetching technique leads to additional saving in sorting time for most partition sizes

Micro row layout is the technique where buffer fill state (counter indicating whether the buffer is full or not) is kept part of the partition buffer. This eliminates the cache line that was otherwise being used for keeping the array of those counters for each buffer. The effect is more prominent for a large partition number. Finally, the benefit of having transparent huge pages (2MB instead of the default 4KB) is prominent for partition number more than 1024.

In [104], the author investigate the duality between bucket sort and hash sort in terms of cache line transfers they have to perform. They presented crude analytical models for the the number of such transfers for both these methods and showed that they are equivalent when some optimizations are applied to both. But, the authors also pointed out how the algorithms perform very differently under different workloads. For instance, hashing is more efficient at early aggregation when there are more repetition in the input. On the other hand, when there is not much repetition/correlation in the input, the extra storage for the hash overhead is wasted, making pure sorting more effective, since sorting is much faster. Taking these facts into consideration, the authors designed an algorithm that would combine both methods in an adaptive manner and enjoy the relative advantage of those. This new algorithm mainly runs on sorting routine, but periodically plugs in hashing routine and gauge the obtained compression. If the compression factor is more than a predefined threshold, it switches to hashing; otherwise sorting routine continues.

The new algorithm is also capable of utilizing multi-core parallelization effectively. To help that, the authors applied a series of optimization techniques so that the computation overhead (they count access latency due cache miss also part of CPU computation) reduces, with the objective of keeping the cost of actual data movement the dominant part of their algorithm. The techniques include *software write combining*, *out-of-order execution by loop-unrolling*, and a *two-level data structure* which they described as the alternative to over-allocated partitions, virtual memory method or an extra counting pass required for radix/bucket partitioning. With these optimizations in place, they achieved 97% of the throughput of that of pure `memcpy`, which indicates their algorithm stays data-movement-bound for the most part.

This algorithm is tested on a 20-core Intel Xeon machine. They demonstrated 16-way scaling with the 20 threads. They also reported achieving 166M keys/sec lookup/insertion speed using Google's `dense_hash_map` with some optimizations. They also compared their algorithm against a number of state-of-the-art methods and demonstrated that their algorithm is the winner in most cases, with sometimes 3X faster performance. They also controlled the skewness of input data using the data generator made available in [36] and showed that their algorithm reacts adaptively across different distributions.

The paper [139] starts out by stressing the fact that with Radix sort where a number of output streams are being written simultaneously, it is impossible to get optimum cache hits unless either the cache is fully associative, or the addresses of each stream differ in certain bit positions, both of which are hard to achieve. This problem is solved by keeping a software managed two-dimensional buffer, where there is a certain portion sized to the cache line for each output stream and they are copied to the output stream when they are full.

The authors also point that although the entire cache lines in the output streams are basically write only (i.e., without any recent reads), they are still read into cache lines before the writes actually happen. This causes extra demand on the memory bandwidth and cache pressure. A new technique facilitated by Intel architectures called "non-temporal streaming store" solves this problem by sending the output directly to the memory without involving the cache. But again this may cause a large volume of bus occupancy which can be solved by another technique called "software write combining" (the default hardware write combining has a limited buffer space) which buffers multiple writes to the same buffer and commits the final result at the end.

Note that the above two problems and their solutions were already known before this paper. Now, the original contribution of this paper is a virtual memory assisted radix sort which eliminated the need for an extra counting-pass during every iteration of the radix sort. This is done by just reserving the output space and committing a page of memory only when an actual access to the reserved space causes a page fault. This is done via signals in POSIX and Vectored Exception Handler in windows.

The main algorithm borrows from the reverse counting [77] where the initial split is done by the most significant digit (a digit is a group of D bits leading to split factor of 2^D), where each split is assigned to a different processor/core. Then, each processor sorts its local data data in $K/D - 1$ LSD (Least Significant Digit) iterations. This leads to more movement of data within the processor, but less movement between the processors. The authors claim that this leads to better memory performance over MSD where the data movement may cross NUMA boundaries. Now, prefix sum is still computed for the output array, but only for the MSD split. For the local splits, no prefix is computed; rather, virtual memory is used to populate the buckets.

The authors used $D = 8$ bits to avoid bit manipulation. Since they sort 32-bit keys, their algorithm needs 3 sets of buckets. As a result, it needs $3 \cdot 2^D \cdot 8 = 6144$ times the input size (8 is the number of cores), limiting the maximum sortable data to only 1.4GB for windows which allows only 8TB of reserve space. For the output splits small pages are used to avoid mapping them to the same TLB entries. On the other hand, input data is managed using large pages to increase its TLB coverage.

The proposed algorithm is tested on dual W5580 CPUs, each 3.2 GHz quad core (as a result total 8 cores), with 48GB of 1066 memory running windows XP 64, where compilation is done with ICC 11.1.082 with a number of other optimizations. They achieve 391 M/s with their VM setup, and then 657M/s with write combining added. This is 64% faster than intel results [118] and 17% faster than another recent work by the same authors of [118] reporting 560 M/s. Recently, [99] reported achieving 1005 M/s in GTX 480 GPU. However, after counting the communication overhead with the GPU over PCIe bus, that method becomes only 501 M/s, loosing to this method.

In [133], the authors propose a new LSD radix sort, where the counting pass is eliminated by keeping an overflow buffer. First, a buffer array is allocated that is slightly larger than the input array, and it is logically divided into buckets. Now the input items are dealt into these buckets while computing histogram for the next LSD digit. The dealing may lead some buckets to overflow, when these items are dealt back to the beginning of the input array where there is adequate space. After this pass finishes, the overflow items remained are deal to the overflow area while building histogram for the next LSD digit as well. Now, in the second pass, the buffer items are dealt from each bucket and also from the overflow buckets. Since the exact counts are now available from the two histograms combined, this pass produces the output that is exactly same as that produced after two passes of a counting-based radix sort.

The authors run their experiments Intel i7 3960 3.6 GHz processor over-clocked to 4.8 GHz with 32 GB DDR3 (1700) RAM to sort with digit size 8 bits. They achieved 118M/s for 32 bit integers, but omitted the exact numbers for 64 bit numbers.

However, they mentioned that their algorithm performs 8% faster than the nearest contender LSD radix sort.

5.3 Preliminaries

5.3.1 Finite Stream

Consider an input stream that consists of key-value pairs, with keys belonging to a finite set V of size n. Define $d(v)$ to be the number of times v appears in the stream, also called frequency of v and let $T = \sum_{v \in V} d(v)$ be the stream length. Now, appearance of the keys can be viewed as a stochastic process $\{X_t\}_{t=1}^{T}$, where X_t is the random key, possibly accompanied by some value, in position $t \in \{1, 2, \ldots, T\}$. This happens for many real-world data (e.g., user clicks, DNS queries, web page requests to a web cache, search in large graphs, queries against a database index, nested MapReduce jobs), where we have only access to a particular window of observation $[1, T]$ giving us only one sample path. Therefore, the frequency counts $d(v)$ should be viewed as for only one instance of the underlying process.

Now, without the knowledge of the underlying distribution, we construct a distribution off the observed frequencies of the nodes and denote that by D. Then, its PMF (probability mass function) is given by:

$$p_x := P(D = x) = \frac{1}{n} \sum_{v \in V} \mathbf{1}_{d(v)=x}, \tag{5.1}$$

where $\mathbf{1}_A$ is an indicator of event A and $x = 1, 2, \ldots, T$.

We assume that the distribution of D is known from a previous experiment with the same data. This knowledge may be available when the stream is queried multiple times for different purposes, or its properties can be predicted from other data sets. Additionally, D may be given by a well-known theoretical distribution in order to understand how the parameters of input affect the resulting behavior of the system. For example, if the frequency becomes more heavy-tailed (i.e., power-law α gets smaller), how does it affect caching behavior on this stream?

As input pairs are being processed one at a time, let $S_t = \bigcup_{j=1}^{t} X_j$ be the set of keys seen by time t. Similarly, suppose set $U_t = V \setminus S_t$ contains the remaining (i.e., unseen) keys. Unless the workload is explicitly known to exhibit correlated occurrence of keys, it is reasonable to assume that the $d(v)$ copies of v are spread uniformly across the length of the stream. This is known as the *Independent Reference Model* (IRM) [40], [97].

In addition, recall the definitions of $p(t)$ from (4.3) for the probability of seeing a unique key and $s(t)$ from (4.5) for the expected seen set size both time at t.

5.4 Incremental Hash Table

Although priority queues support $O(\log n)$ lookup and insertion operations, in practice, they are slow with regard to high performance requirements of large-scale applications. For this reason, we implement replacement selection using hash table, which provides faster performance. In this design, we just keep an eviction pointer that points to the next bucket to evict. Since we hash based on the leading bits of the key, the buckets are already sorted in their key order (all items in the i-th bucket is smaller than those in the $i + 1$). When the hash table is full, the items in the bucket pointed to by the eviction pointer are evicted, and the eviction pointer then advances to the next bucket. Once the eviction pointer reaches the last bucket, it moves back to the first bucket, the current run closes, and the next run starts.

In this section, we develop accurate models for the sorted runs (their size and number) for this Incremental Hash Table (IHT).

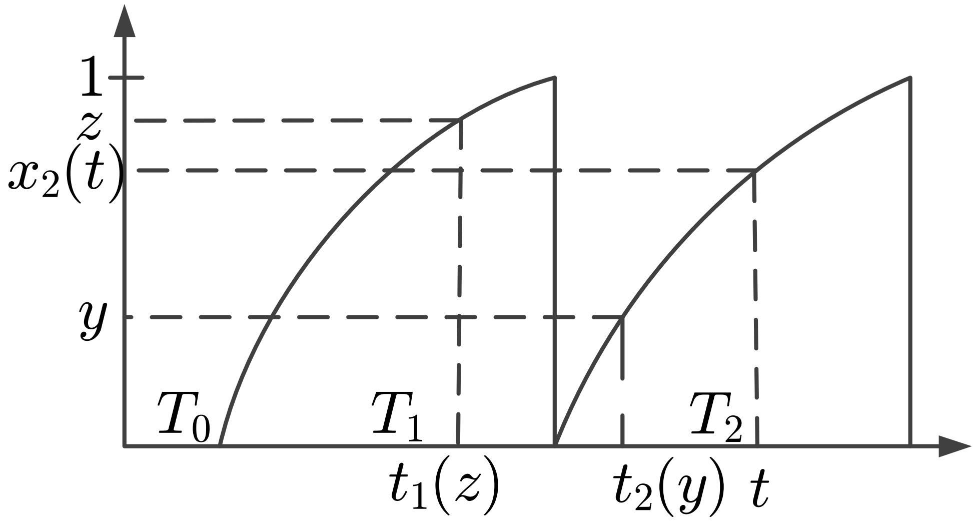


Figure 5.1: System model in IHT.

5.4.1 Individual Run Analysis

We start our analysis following Moore's snow plow model, first proposed in [102] for unique/random data, where a snow plow is moving across a circular track, snow is falling at a constant rate, and the speed of the plow is such that the track always contains a constant amount of snow. Similar to this model, we have a hash table along with a eviction pointer that moves around the buckets while evicting items in those buckets and always maintaining $\sim h$ tuples in the hash table.

Let $x_i(t)$ be the position of the eviction pointer at time t during the i-th sorted run, $x_i(0) = 0$, and $x_i(T_i) = 1$. The inverse function $t_i(x)$ also exists such that $t_i(0) = 0$ and $t_i(1) = T_i$. Fig. 5.1 shows the system model, where the eviction pointer starts at the origin point 0, travels to 1, and then starts over at 0 again. The point x is reached at time $t_{i-1}(x)$ during the $(i-1)$-th run and at time $t_i(x)$ during the i-th.

The rate of movement of the eviction pointer is an important quantity of interest, because it determines the time length for generating each sorted run, eventually leading to their length and number. Before that, we need a number of new functions.

5.4.2 Partial Key Space Functions

Define $s_{ab}(t)$ to be the expected size of the seen set for the keys in range $[a, b]$, where $0 \leq a, b \leq 1$, and we assume the full range $[0, 1]$ to cover all the buckets. Therefore, a and b are two points in Y-axis of Fig. 5.1. Define $p_{ab}(t)$ to be the probability of the t-th key in range $[a, b]$ being previously unseen. Due to uniform key distribution, these functions are controlled only by the width of the range $(b-a)$, rather than the position (i.e., a or b). Therefore, we use notations $p_{ab}(t)$ and $p_{(b-a)}(t)$ interchangeably. The same applies for notations $s_{ab}(t)$ and $s_{(b-a)}(t)$. The following lemma expresses $p_{ab}(t)$ and $s_{ab}(t)$ as functions of original $p(t)$ and $s(t)$ respectively.

Lemma 3. *The quantities $p_{ab}(t)$ and $s_{ab}(t)$ are given by:*

$$p_{ab}(t) = p(t/(b-a)), \quad s_{ab}(t) = (b-a)s(t/(b-a)). \tag{5.2}$$

Proof. From (4.3) and assuming that the number of unique keys in the entire stream belong in range $[a, b]$ to be T_{ab}, we have :

$$\begin{aligned} p_{ab}(t) &= \frac{1}{E[D]} E[(1 - \frac{t}{T_{ab}})^{D-1}] \\ &= \frac{1}{E[D]} E[(1 - \frac{t}{T(b-a)})^{D-1}] = p(t/(b-a)), \end{aligned}$$

where we used the fact that $T_{ab} = T(b-a)$.

Furthermore, from (4.5) and assuming n_{ab} to be the number of unique keys from the input in range (a, b), we have:

$$\begin{aligned} s_{ab}(t) &= n_{ab}E[1-(1-\frac{t}{T_{ab}})^D] \\ &= n(b-a)E[(1-\frac{t}{T(b-a)})^D] = (b-a)s(t/(b-a)). \end{aligned}$$

□

5.4.3 Key Density Functions

We define key density $\Omega(t)$ as the number of keys in an infinitesimally small key-range after t time-units (i.e., after seeing t pairs from the input) since the last time this bucket was evicted. Therefore,

$$\Omega(t) = \lim_{a \to b} \frac{s_{ab}(t)}{b-a}. \tag{5.3}$$

Note that $\Omega(t)$ is not tied to a any location a or b, because the location is implicit in the time duration t. When we look at a particular snapshot of the hash table, each bucket has a different density $\Omega(t)$, because each of them are exposed to incoming keys for a different duration t. The following theorem quantifies density $\Omega(t)$ in terms of $s(t)$.

Theorem 16. *The density function $\Omega(t)$ is equivalent to the expected size of the seen set at t, i.e.,*

$$\Omega(t) = s(t). \tag{5.4}$$

Proof. The density of a key region $[a, b]$ changes whenever a previously unseen key enters into buckets belonging to this region. Therefore, the amount of change is:

$$\Omega(t+1) - \Omega(t) = p_{ab}(s_{ab}^{-1}(\Omega(t)(b-a))). \tag{5.5}$$

After using Lemma 3 on the right side of (5.5), it becomes:

$$\frac{d\Omega(t)}{dt} = p_{ab}((b-a)s^{-1}(\Omega(t))) = p(s^{-1}(\Omega(t))), \tag{5.6}$$

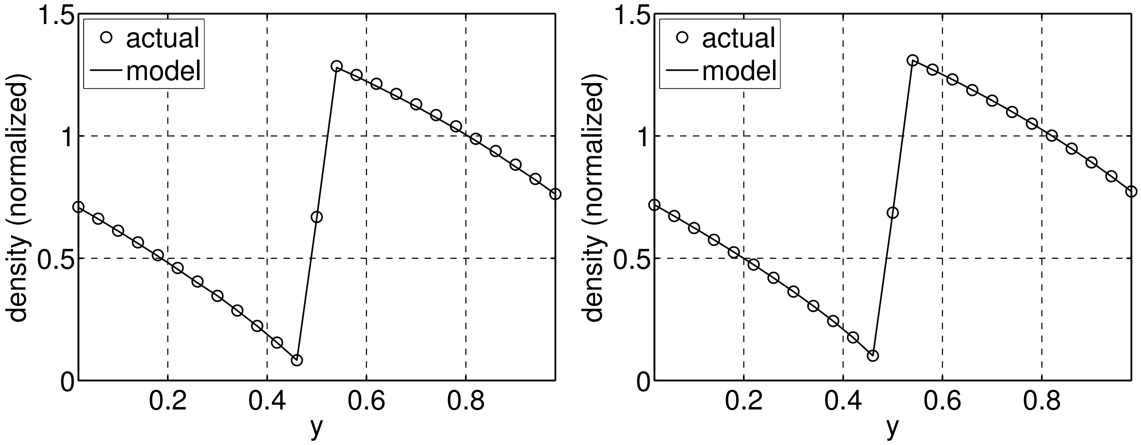
which solves with $\Omega(t) = s(t)$. □

Now, given that the eviction pointer is located at x, we define $\rho_i(x, y)$ to be the spatial density of keys in the vicinity of y during sorted run i. Consider two scenarios from Fig. 5.1 during sorted run i. First, when $y < x$ (i.e., y is to the left of the eviction pointer at x), the vicinity of y is exposed to incoming keys since the time the eviction pointer was x, which amounts to $t_i(x) - t_i(y)$. Second, when $y >= x$ (i.e., y is to the right of x), the vicinity of y was evicted during the $(i-1)$-th run, but not yet during the i-th run. Therefore, its exposure time to incoming keys is $T_{i-1} - t_{i-1}(y) + t_i(x)$. Combining these two, we get:

$$\rho_i(x, y) = \begin{cases} s(t_i(x) - t_i(y)) & \text{if } y < x, \\ s(T_{i-1} - t_{i-1}(y) + t_i(x)) & \text{if } y \geq x. \end{cases} \tag{5.7}$$

We measure density $\rho_i(x, y)$ by fixing $x = 0.5$ and varying y in the entire range $[0, 1]$, and then show the results in Fig. 5.2 for IRLbot host graph and WebBase web graph, which shows that the model (5.7) is accurate for $x = 0.5$. We have also verified for other x values and they are accurate as well.

A special case of the above is the density in the vicinity of x when the eviction



(a) IRLbot host graph

(b) WebBase web graph

Figure 5.2: Verification of (5.7) with $x = 0.5$.

pointer is also at x, i.e., it is just about to clear the bucket at x. In that case, from (5.7), we have:

$$\rho_i(x, x) = s(T_{i-1} - t_{i-1}(x) + t_i(x)). \tag{5.8}$$

Observe from (5.8) that the duration $T_{i-1} - t_{i-1}(x)$ measures the time since the last time the bucket at x was cleared. If we assume that it takes m keys from input to fill the hash table during the first run, we obtain that $T_0 - t_0(x) = m$, because each bucket started empty at the same time. Therefore, only for the first run, we have:

$$\rho_1(x, x) = s(m + t(x)). \tag{5.9}$$

Finally, the eviction pointer starts from $x = 0$, evicts the bucket at x, and then advances to the next bucket. Thus, every time buckets with density $\rho_i(x, x)$ are sent to the disk. This enables us to compute the length of the i-th sorted run, which we denote by l_i, can compute as:

$$l_i = \int_0^1 \rho_i(x, x)dx. \tag{5.10}$$

5.4.4 Miss Rate

Now, given the eviction pointer position at x during sorted run i, define $q_i(x)$ to be the probability of a key being missed in hash table. The following theorem quantifies $q_i(x)$.

Theorem 17. *The probability of a miss in the IHT with the eviction pointer at x is:*

$$\begin{aligned} q_i(x) = &\int_0^{x(t_i)} p(t_i(x) - t_i(y))dy \\ &+ \int_{x(t_i)}^1 p(T_{i-1} - t_{i-1}(y) + t_i(x))dy. \end{aligned} \tag{5.11}$$

Proof. Defining the set of keys in the hash table during the i-th run given a pointer location x by $H_i(x)$ and denoting the t-th key by X_t, we have:

$$\begin{aligned} q_i(x) &= P(X_t \notin H_i(x)) \\ &= \sum_{i=1}^{1/dy} P(X_t \notin H_i(x) | X_t \in [y, y + dy])P(X_t \in [y, y + dy]) \\ &= \sum_{i=1}^{1/dy} p_{dy}(s_{dy}^{-1}(\rho_i(x, y)dy)) \approx \int_0^1 p(s^{-1}(\rho_i(x, y)))dy. \end{aligned} \tag{5.12}$$

Then, we obtain (5.11) by replacing $\rho_i(x, y)$ using (5.7) in (5.12). $\square$

5.4.5 Main Results and Convergence

Now, the rate of movement of the eviction pointer and its inverse function are given by:

$$\frac{dx}{dt} = \frac{q_i(x)}{\rho_i(x, x)}, \quad \frac{dt_i}{dx} = \frac{\rho_i(t)}{q_i(t)}. \tag{5.13}$$

After the system goes through a number of sorted runs, it reaches a convergence condition, where the eviction pointer speed becomes constant and it becomes inversely proportional to cycle length T_∞, i.e.,

$$\frac{dx}{dt} = \frac{1}{T_\infty}, \tag{5.14}$$

In addition, sorted run i and $i-1$ take the same amount of time to cross the point x (i.e., $t_i(x) \approx t_{i-1}(x)$) after convergence. Therefore, from (5.8), the density functions change to $\rho_\infty(x, x) = s(T_\infty)$. Using this in (5.13) and defining the steady-state miss rate as q_∞, we get:

$$q_\infty = \frac{s(T_\infty)}{T_\infty}, \tag{5.15}$$

which implies that q_∞ becomes constant and independent of x after convergence.

Now, what remains to determine is the value of T_∞, which we derive next. Consider (5.7) and a scenario where have fixed $x = 0$. In this case, all $y \in [0, 1]$ points are to the right of x. Since $t_i(0) = 0$, this leads to:

$$\rho_i(0, y) = s(T_{i-1} + t_i(0) - t_{i-1}(y)) = s(T_{i-1} - t_{i-1}(y)). \tag{5.16}$$

Now, after the system has converged, we can rewrite the above by replacing i

with ∞ as follows:

$$\rho_\infty(0, y) = s(T_\infty - t_\infty(y)). \tag{5.17}$$

At this point, similar anytime else, the hash table contains h items. By definition of density, we have:

$$h = \int_0^1 \rho_\infty(0, y)dy = \int_0^1 s(T_\infty - t_\infty(y))dy. \tag{5.18}$$

Since we assume that $dx/dt = 1/T_\infty$ after converge, we obtain $t_\infty(x) = xT_\infty$. Putting this in (5.18), we get:

$$h = \int_0^1 s(T_\infty - yT_\infty)dy. \tag{5.19}$$

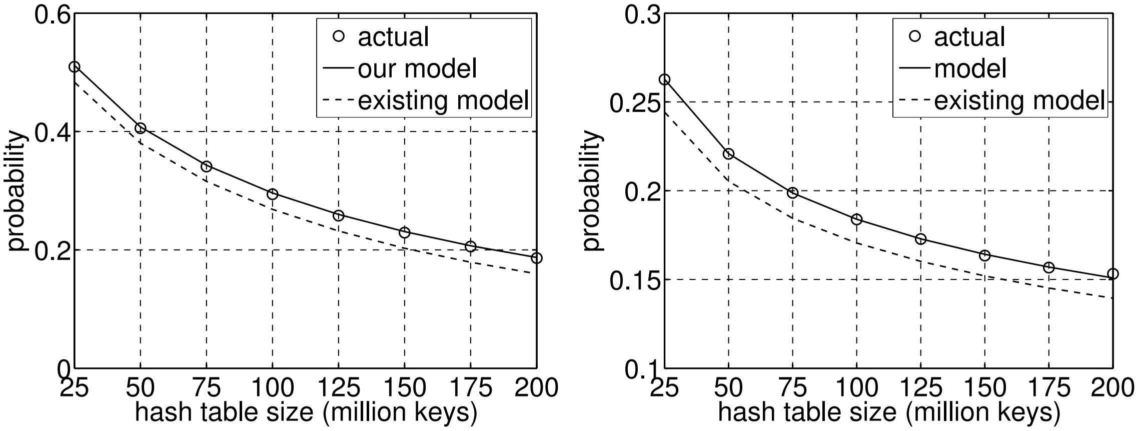
Then, after using $t = T_\infty - yT_\infty$ and changing the limits accordingly, we obtain:

$$h = \frac{1}{T_\infty} \int_0^{T_\infty} s(t)dt, \tag{5.20}$$

which can inverted numerically to obtain T_∞ for a given h.

We verify q_∞ on IRLbot host graph-WebBase web graph in Fig. 5.3, which shows that our model is accurate. We also include the miss rate model from [84] which is the closest existing work regarding this problem, which, although being close, is not accurate.

At this point, we apply the above results to analyze individual runs. First, we investigate the first run. The speed of the eviction pointer can be approximated from



(a) IRLbot host graph (b) WebBase web graph

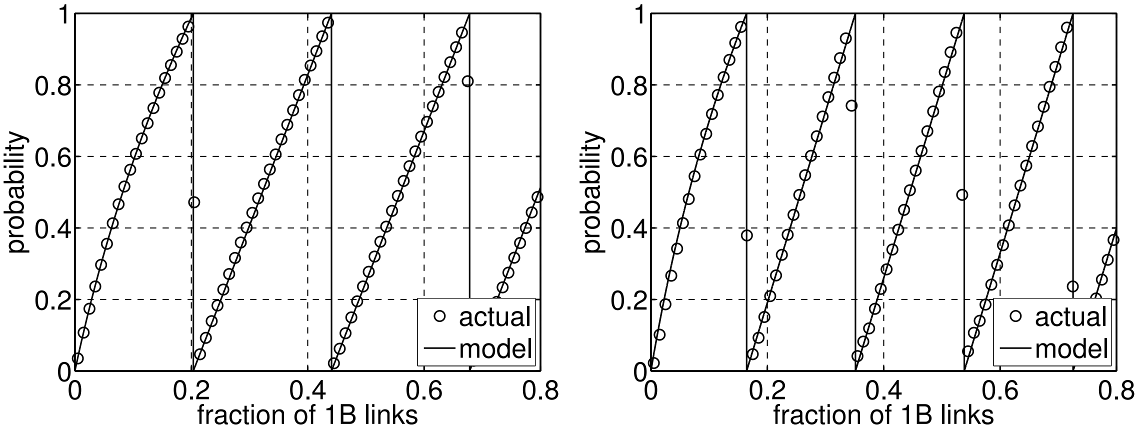
Figure 5.3: Verification of q_∞ in (5.15).

(5.13) by replacing $q_i(x)$ with q_∞ and using (5.9) in (5.13) as follows:

$$\frac{dx}{dt} \approx \frac{q_\infty}{s(m+t(x))} = \frac{s(T_\infty)}{s(m+t(x))T_\infty}. \tag{5.21}$$

Empirically, we observe that IHT reaches convergence state after the first run. Therefore, for all subsequent runs, we simply use (5.14). We verify (5.21) and (5.14) while generating the first few runs from the IRLbot host graph and WebBase web graph graphs, and show the results in Fig. 5.4(a) and (b), respectively. Observe that the models are accurate in both cases.

Now, we can count the number of sorted runs by a variable that counts how many times the value of x hits 1. Although being a simple metric to compute, the number of sorted runs is an important measure for determining the merge overhead. Also note that there is no existing work that models the sorted runs generated by such IHT-based replacement selection. We verify this model in Fig. 5.5 on IRLbot host graph and WebBase web graph, which demonstrates the model is exact.



(a) IRLbot host graph

(b) WebBase web graph

Figure 5.4: Verification of (5.13), (5.14), and (5.21).

We derive the length of the first run as:

$$l_1 = \int_0^1 \rho_1(x, x) dx, \tag{5.22}$$

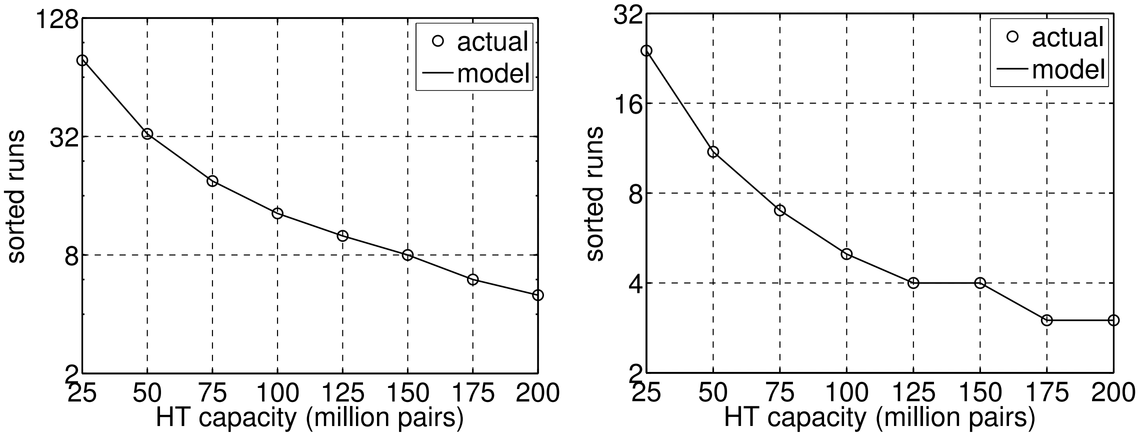
and for other runs (i.e., $i > 1$), length is given by simply:

$$l_i = l = s(T_\infty). \tag{5.23}$$

Now, using (5.22), (5.23), and the iterative model for k, we can compute the total disk I/O for IHT by first computing the sort overhead as:

$$L_s = l_1 + (k-1)l, \tag{5.24}$$

and then merge overhead L_m follows from Algorithm 1. Finally, we verify this model on IRLbot host graph and WebBase web graph in Fig. 5.6. Observe that the model is accurate in both cases.



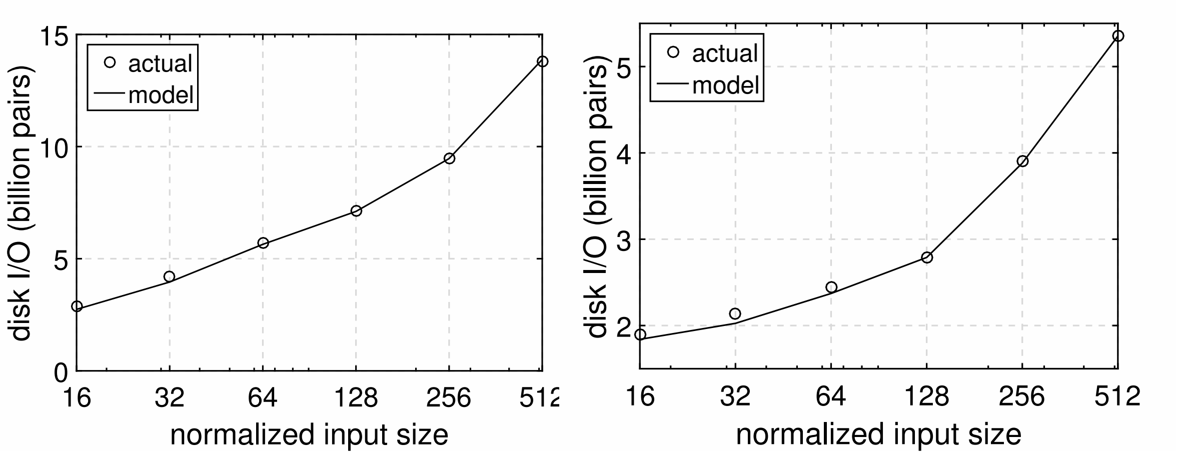
(a) IRLbot host graph (b) WebBase web graph

Figure 5.5: Verification of sorted runs generated by IHT MapReduce.

Table 5.1: Run lengths (normalized by h) of IHT on the IRLbot graph.

h	Run	Length	Our model		Existing model	
			value	error	value	error
25M	1	1.5852	1.5876	0.15%	1.72	8.50%
	2	1.7756	1.8025	1.50%	1.95	9.82%
	3	1.7997	1.8025	0.15%	1.99	10.57%
200M	1	1.4384	1.4312	0.49%	1.72	19.58%
	2	1.5758	1.5883	0.79%	1.95	23.74%
	3	1.5846	1.5883	0.23%	1.99	25.58%

Note that although some parts of derivation of the above model target a specific implementation (i.e., IHT) of replacement selection, it is not limited to this particular design as it can easily modified to other type of implementations. In addition, we argue that this model is universal, since IHT is one of the few ways, if not the only way, to implement the principal of replacement selection with the added advantage of online aggregation (looking up and combining new items with existing ones).



(a) IRLbot host graph (b) WebBase web graph

Figure 5.6: Verification of disk I/O by IHT MapReduce.

5.4.6 All-Unique Keys

Here we investigate a special case where all keys are unique, i.e., there is no/negligible repetition of them. Note that, existing literature focuses mostly on this scenario. There are a few changes that result from such work load. First of all, miss rate is 100% (i.e., $q_i(x) = q_\infty = 1$). Then, the size of the seen set after processing t keys is exactly t (i.e., $s(t) = t$). We apply these to (5.20) and get:

$$h = \frac{1}{T_\infty} \int_0^{T_\infty} t dt. \tag{5.25}$$

After rearranging, the above becomes:

$$T_\infty = 2h, \tag{5.26}$$

which states that in the steady state, the replacement selection hash table's sorted runs are twice as large as the hash table capacity h. Now, we draw our attention to the first sorted run length, which we assume to be k keys. For this k time period, the spatial density at x, given the eviction pointer at x from (5.9) is:

$$\rho_1(x,x) = s(h + t(x)) = h + t(x). \tag{5.27}$$

Applying this to (5.21), we get: $\dot{x} = 1/(h+t)$, which after integrating from time 0 to k, we get:

$$\int_0^1 dx = \int_0^k \frac{dt}{t+h}, \tag{5.28}$$

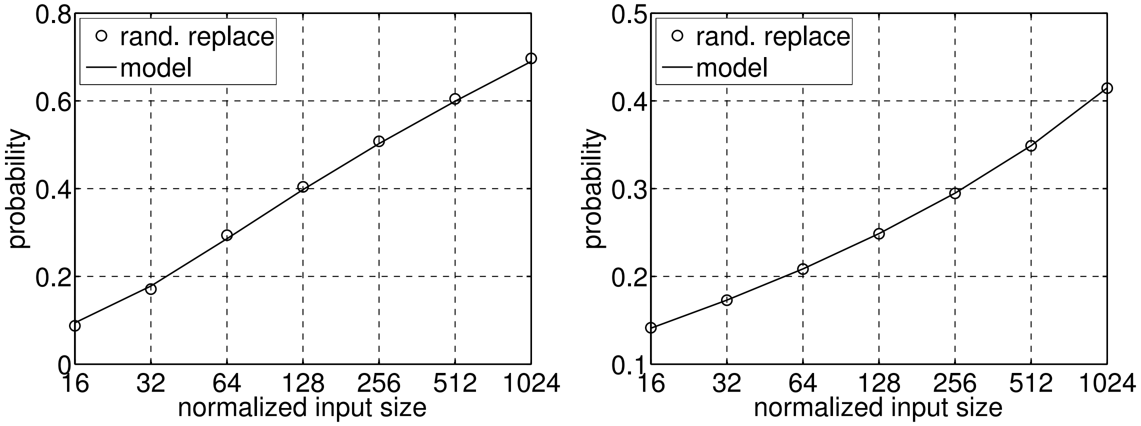
which after solving the integral becomes:

$$k = (e-1)h.$$

Note that both (5.26) and (5.28) are well-known results due to [47], [79], [102], where the authors obtained the same results using separate techniques. Thus, we establish that we solve a more general replacement selection problem where the keys can be both repeated and all-unique/randomly generated, while existing literature was limited to only the all-unique case.

In addition, to demonstrate the usefulness of our model, we present the relative length of the sorted runs produced by replacement selection hash table in Table. 5.1 under different hash table capacities. The first thing to notice is that the size of the sorted runs is a function of the hash table size. For h = 25M, the first sorted run is 1.58h and the other are 1.8h, while these numbers change to 1.43h and 1.58h respectively for a large hash table of size h = 200M. Therefore, we cannot universally assume one factor for each hash table size as the existing literature assumed for replacement selection (i.e., 1.72h for the first and 2h for the others).

Also note that the error of our model capturing the second run is the highest for



(a) IRLbot host graph (b) WebBase web graph

Figure 5.7: Verification of q_∞ on rand. replacement.

both hash table sizes (i.e., 1.5% and 0.79% respectively). This happens due to our simplifying assumption that the system converges after the first run, which might not be always true and can be easily removed from our model.

5.4.7 Extension to Caching Techniques

Interestingly, the models derived here can be easily extended to random replacement and First-In-First-Out (FIFO) caches. In a random replacement cache, it is the best if the lookup is an $O(1)$ operation and for that, hash table is a typically used data structure. But the evictions happen at uniformly and randomly chosen data items. We implement a random replacement cache like this, pass IRLbot host graph and WebBase web graph through that, record the miss rates under varying cache sizes, and plot the results in Fig. 5.7(a) and (b). The figures show that the q_∞ model derived in (5.15) is accurate in all the data points. The author in [54] shows that the performance of random replacement and FIFO caches are same under IRM traffic which we assume as well. Therefore, we can argue that the q_∞ model is accurate for FIFO caches too.

5.5 Space Efficient Caching

For an external memory program like merge sort, hash sort or bucket sort, the main memory is extremely scarce resource. Therefore, it is very important that it is used economically. Since a number of sorting methods including bucket sort takes advantage of using a cache between its levels, it is obvious that the used cache takes minimum amount of space and still shows good (i.e., comparable to LRU) hit rate. But we know that LRU cache has to move an item to the front of the eviction list, where items are evicted from the back. This requires a global lock in when multiple threads are using a the same cache. Even in the absence of multiple threads, the required data movement still causes a considerable performance penalty. In addition, LRU requires a huge space overhead (i.e., a hash table for fast look up, and the linked list to maintain eviction order). As a result, other simpler caching methods (e.g., random replacement or FIFO) are more attractive that have less overhead and higher processing speed. But, on the downside, their hit rates are less than LRU.

To solve this problems, we propose a new caching method, we call WATCH, that has a high throughput and also has minimal space overhead (only 1 byte/key compared 40 bytes/key in LRU). We discuss the design and performance model of this cache in the following.

5.5.1 WATCH

In this cache, there is an array of B tiny CLOCK caches, each with a fixed but configurable size r, where r is typically between 2 to 8 elements. Similar to CLOCK, each of these tiny caches maintains a hand to find out next-to-evict item. Each slot in tiny caches is described by one of the 3 possible statuses: *empty, not hit,* and *hit*. Note that the measurement of popularity is just a binary decision: popular (indicated by hit) or not (indicated by not hit status).

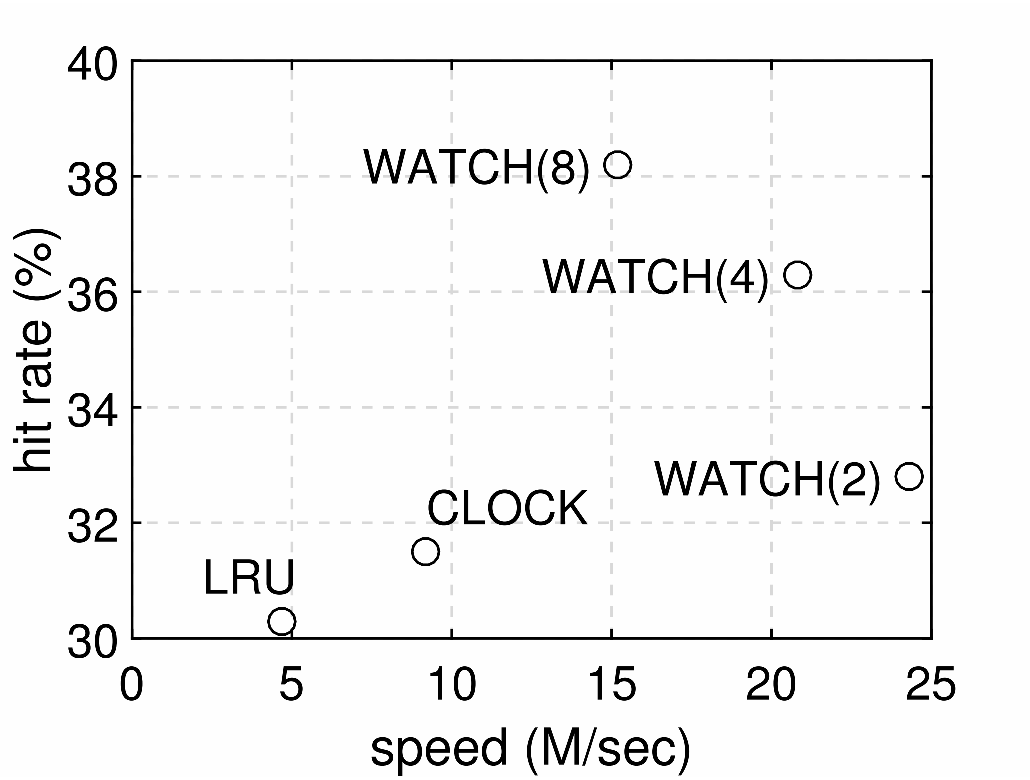


Figure 5.8: Tradeoff between speed and r in terms of hit rate (128M lookups, 64K cache size).

Each slot starts as an empty one, until a item is first inserted in that slot. A item is first hashed to choose a cache line (i.e., one of $[1, B]$). Then, within that cache, the item is attempted for a lookup. If the item does not exist in any of the slots, then it is inserted into an empty slot, and the status of the slot is changed to "not hit". Subsequently, if the same item is looked up again, that indicates that this item is popular and therefore, marked "hit".

If a key does not exist in the target cache, and there is no empty slots there, an existing item has to be evicted to make room for the new item. This is done with the help of a hand, one for each cache, which moves around turning each "hit" slot into a "not hit" one and stoping at an item that is "not hit", which is then evicted and the new item is placed there. In the worst case, the hand moves around circularly over all r items when all items in that cache were initially marked “hit” and then stops at the item where it started its search, which is then evicted.

Observe in Fig. 5.8 that there is a trade-off between r and the observe processing speed of the cache. If we increase r to 8, the hit rate increases, but the speed decreases, because of the long movement cycle of the hand. In addition, note that the hit rate is higher compared to both LRU and CLOCK.

5.5.2 Analysis of WATCH

Suppose WATCH is a low-overhead, compact version of CLOCK. When the number of unique keys $n \geq 2h$, the expected number of unique keys evicted by WATCH is pretty close to $n - h$; however, when $h \approx n$, this no longer holds. Indeed, evictions can occur even when $n < h$.

Given a WATCH(r) cache with $B = h/r$ bins, suppose the number of keys in chain i is $X_i \sim$ Binomial(n, $1/B$). For sufficiently large n, this is Poisson-distributed with rate $\lambda = n/B$. Then, after passing n unique keys through the cache, the expected number of them evicted is:

$$\begin{aligned}\mathcal{E}_r &= \sum_{i=1}^{B} E[X_i - r | X_i > r] P(X_i > r) \\ &= \frac{h}{r} e^{-\lambda} \sum_{j=r+1}^{n} (j - r) \frac{\lambda^j}{j!}. \end{aligned} \tag{5.29}$$

For $n = h$, the fraction of evicted keys is 9.9% for $r = 16$, 14.0% for $r = 8$, and 19.5% for $r = 4$. This shows that retaining n keys in RAM requires a cache of size n and a secondary hash table with between $0.1n$ and $0.2n$ keys. Of course, this approach consumes slightly more CPU cycles because of two table lookups for $10 - 20\%$ of the keys, but it also significantly reduces disk I/O in certain cases (i.e.,

Table 5.2: Verification of (5.32) on the IRLbot graph.

Size	B($r = 4$)	n	Evictions		Error
			actual	model	
2^{20}	2^{18}	2^{18}	1,139	1,144	0.44%
		2^{19}	19,776	19,904	0.65%
		2^{20}	211,425	208,628	1.32%
		2^{21}	1,129,257	1,105,160	2.13%
		2^{22}	3,464,575	3,447,571	0.49%
2^{21}	2^{19}	2^{19}	2,309	2,299	0.43%
		2^{20}	40,505	39,920	1.44%
		2^{21}	436,850	419,672	3.93%
		2^{22}	2,307,684	2,262,064	1.98%
		2^{23}	7,191,291	7,224,491	0.46%
2^{22}	2^{20}	2^{20}	4,728	4,609	2.52%
		2^{21}	84,477	80,150	5.12%
		2^{22}	884,714	851,707	3.73%
		2^{23}	4,705,652	4,655,209	1.07%
		2^{24}	15,320,619	14,937,303	2.50%
2^{23}	2^{21}	2^{21}	9,806	9,247	5.70%
		2^{22}	170,724	161,965	5.13%
		2^{23}	1,779,735	1,731,839	2.69%
		2^{24}	10,102,415	9,484,318	6.12%
		2^{25}	32,021,347	31,267,935	2.35%

by eliminating the last level of bucket split).

If $n = h$ and $r = 8$, we need $13 + 0.14 \times 21.22 = 15.97$ bytes per key (the overhead is then $\mathcal{O} = 3.97$). Now, given the size of input n, what is the optimal value of h to use that would consume the least RAM? For $r = 8$, binary search over $\mathcal{E}_r$ produces $h_{opt} \approx 0.85n$ and thus $15.60n$ bytes/key. This is slightly better than the naive approach above with its 15.97 bytes/key. For $r = 4$, we get $h_{opt} \approx 0.76n$ and 16.47 bytes/key. Naturally, as r decreases, the overhead increases.

Expanding on these ideas further, when RAM allows h larger than n, it is possible to switch to a smaller r to make the cache faster. Therefore, we want to use the smallest r allowed by RAM, coupled with the corresponding $h_{opt}(r)$.

A more careful modeling of WATCH involves considering each bin of it as an LRU cache of capacity r items. Then, total eviction out of WATCH is just the summation of expected eviction from each of the constituent LRU caches. Suppose X_i keys in bin i correspond to Y_i keys in the original stream, where ideally we have $Y_i = X_i E[D]$. In addition, assume that $m = s_i^{-1}(r)$ is the number keys in the stream required to fill up a cache of size r. Thus, we obtain:

$$s_i(t) = X_i \left(1 - E[(1 - \frac{t}{Y_i})^D]\right) = \left(\frac{X_i}{n}\right) s \left(\frac{tn}{X_i}\right), \tag{5.30}$$

which leads to:

$$m = s_i^{-1}(r) = \left(\frac{X_i}{n}\right) s^{-1} \left(\frac{rn}{X_i}\right). \tag{5.31}$$

Now, using (5.33), the total eviction out of the B bins is:

$$\mathcal{E}_r = \sum_{i=1}^{B} E[(Y_i - m)p(m)|X_i > r]P(X_i > r)$$

$$= B \sum_{j=r+1}^{\infty} (Y_i - m)p(m)P(X_i = j)$$

$$= B \sum_{j=r+1}^{\infty} (jE[D] - m)p(m)P(X_i = j) \tag{5.32}$$

Table 5.2 compares model (5.32) against actual results obtained by passing a varying number (stream containing different number of unique items n) of IRLbot edges through WATCH cache of varying sizes and a fixed $r = 4$. We can observe that the model is pretty accurate with the maximum error of 6.12% in case of cache size of 2^{24} items, and a stream containing 2^{23} unique items. Note that the length of the stream to process a fixed number n of unique items is easily obtained from $s^{-1}(n)$.

5.6 Bucket Sort

This section describes a very popular external memory method called bucket sort. As opposed to load-sort-store methods, there is no final merging step necessary.

5.6.1 Bucket Sort with Cache

In a traditional bucket sort, the input is b-way divided into file buckets ensuring that all hashes in bucket i are smaller than those in $i+1$. This process is recursively applied to each bucket until they all fit in main memory. At the last level of partitioning, buckets $1, 2, \ldots$ are loaded in RAM in sequential order and sorted in place. The result is then de-duplicated and appended to the output file. It is also common to deploy a cache of size h to accumulate the most frequent keys before the data is written into the buckets. This causes additional RAM usage, but saves in disk I/O. Division into buckets forms a tree-like processing structure shown in Fig. 5.9.

For the experiments here, we also consider another cache design known as CLOCK, which emulates LRU by keeping a circular buffer and a *hand* that moves around to indicate the next-to-evict item. Each element has a *recently accessed* flag indicating

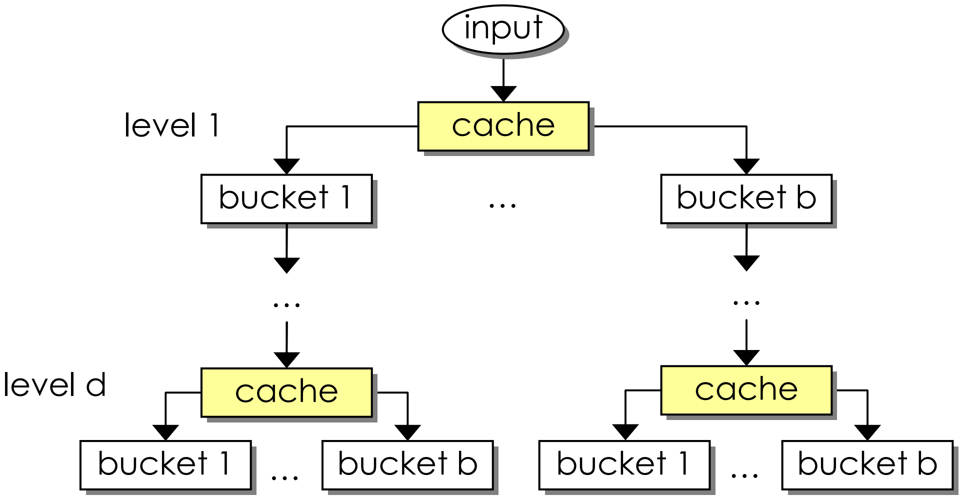

input
level 1
cache
bucket 1
…
bucket b
…
…
level d
cache
cache
bucket 1
…
bucket b
bucket 1
…
bucket b

Figure 5.9: Partitioning tree in bucket sort for homogeneous split.

whether the item was accessed recently, which is set to 1 on hit and reset to 0 by hand while moving over each item in search of an already 0.

5.6.2 Data Compression under Caching

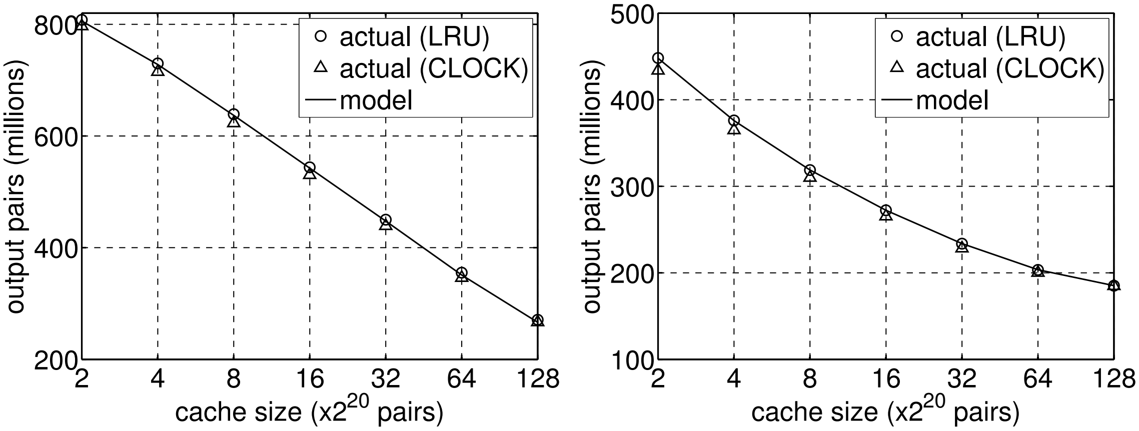
The following theorem quantifies the amount of data that comes off an LRU cache when the input stream passes through it.

Theorem 18. *A stream of size T that goes through an LRU cache of size h compacts the input to:*

$$L_1 = h + (T - m)p(m) \tag{5.33}$$

records on average, where $m = s^{-1}(h)$.

Fig. 5.10(a) and (b) compare L_1 from (5.33) and against actual disk I/O while running the the first 1B edges of the IRLbot and WBWG graphs through LRU and CLOCK caches of varying sizes. All of them demonstrate that the model is exact



(a) IRLbot

(b) WebBase

Figure 5.10: Verification of L_1 (5.33) on real graphs under LRU and CLOCK.

for LRU and very close for CLOCK. Therefore, the compaction model in (5.33) can be used an approximate model for CLOCK, which is usually preferred for its better processing performance.

5.6.3 Homogeneous Split

Here, we develop the disk I/O model for bucket sort that uses the same b in each level. In this chapter, we call this homogeneous split. The other alternative is heterogeneous splits, where each level splits the respective input using a different split factor. We start with the homogeneous case.

Theorem 19. *For homogeneous b, the average amount of I/O in multi-phase LRU bucket-sort deduplication is:*

$$L = \sum_{i=1}^{d} \Big[b^{i-1}h + (T - b^{i-1}m_i)p(b^{i-1}m_i) \Big], \tag{5.34}$$

where $m_i = s^{-1}(b^{i-1}h)/b^{i-1}$ *and* d *is the smallest integer* i *such that:*

$$\frac{b^{i-1}h + (T - b^{i-1}m_i)p(b^{i-1}m_i)}{b^i} \leq \frac{R}{c}. \tag{5.35}$$

Proof. We assume a centralized cache model, which in this problem has the same expected performance as b caches located in front of each bucket. Notice that the cache at level i receives a substream of T/b^{i-1} items whose frequency distribution follows that of D, which follows from uniform sampling of the original stream. The corresponding uniqueness probability and seen-set size for each substream are:

$$p_i(t) = \frac{E[D(1 - b^{i-1}t/T)^{D-1}]}{E[D]} = p(b^{i-1}t) \tag{5.36}$$

$$s_i(t) = \frac{n}{b^{i-1}}E[1 - (1 - b^{i-1}t/T)^{\mathcal{I}}] = \frac{s(b^{i-1}t)}{b^{i-1}}. \tag{5.37}$$

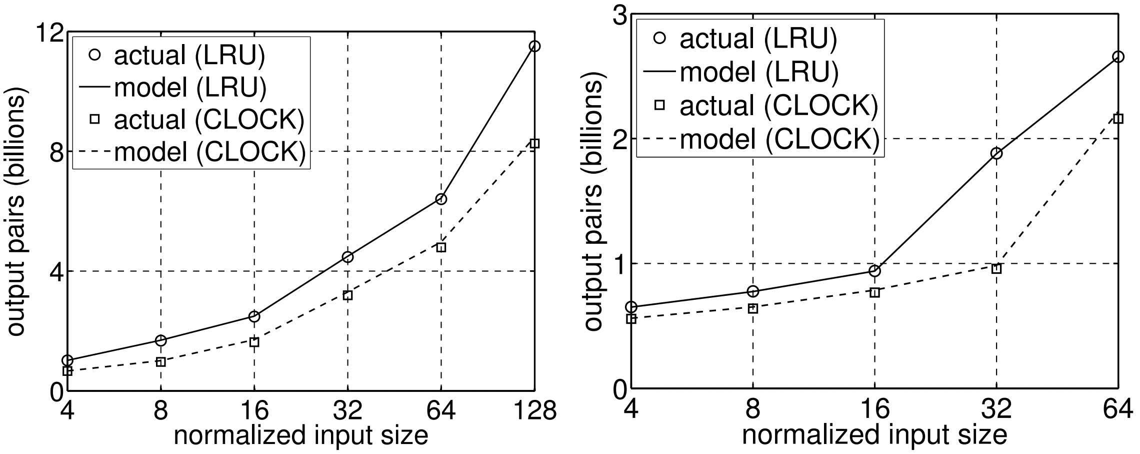
As the cache has a fixed size h, the number of substream keys at level i needed to fill it up is $m_i = s_i^{-1}(h) = s^{-1}(b^{i-1}h)/b^{i-1}$. Considering all b^{i-1} substreams fed through the cache at level i, their combined write I/O from (5.33) is:

$$L_i = b^{i-1}\Big[h + \Big(\frac{T}{b^{i-1}} - m_i\Big)p(b^{i-1}m_i)\Big]. \tag{5.38}$$

Simplifying and summing up for all levels yields (5.34). □

Interestingly, the cache can be unplugged from the model by setting $h = 0$, which results in $m_i = 0$ and $p(b^{i-1}m_i) = 1$. Then, (5.34) produces the non-caching result $L = dT$. Also, if $d = 1$, we obtain $L = L_1$ as expected.

We compare (5.34) against bucket sort using split factor b on the IRLbot and WebBase graphs in Fig. 5.11 (a) and (b), which show that the model is accurate in both cases. Note that as the normalized input size (RAM size to input size ra-



(a) IRLbot

(b) WebBase

Figure 5.11: Verification of L (5.34) for homogeneous b on real graphs under LRU and CLOCK.

tio) grows, the LRU based method requires much more I/O compared to CLOCK, although as we saw from Fig. 5.10 that these methods have very close cache compaction. This indicates that cache overhead $\mathcal{O}$ causes LRU to fit less items in the RAM compared to CLOCK.

Next, we move to the case where the bucket sort program can choose to use different split factor b in each level.

5.6.4 Heterogeneous Split

When the split factor b varies across levels, (5.34) does not apply anymore. Rather, we propose Algorithm 2 to compute the I/O overhead L. The basic difference between this algorithm and Theorem 19 is that, the algorithm uses a vector $\mathbf{b}$, whose i-th member b_i equals the split factor in level i. In addition, we denote by $\mathbf{0}_n$ and $\mathbf{1}_n$ two vectors each containing n items, where each item equals 0 and 1, respectively.

The algorithm first computes the cache capacity h that is available in RAM (line

4) and then obtains the expected bucket size $B[i]$ for that level (lines 5-7) using (5.33). In addition, we also use (5.36) and (5.37), since a sub-stream of average length $T/nBuckets$ maps to each bucket in level i (i.e., $prod = b_1 \times b_2 \dots b_{i-1}$). The output of the cache is then split into b_i equal-sized buckets. If such a bucket fits in the RAM (checked in line 11), the function returns the computed cost L, which leads to termination before expanding the tree to level n.

Another way the bucket sort program can terminate early is that all keys of a bucket would fit in a single cache in the next level, which leads to no data spilled to disk from the cache for the subsequent level. This condition is checked in line 13. Once the loop ends, the sequence b is truncated to the number of levels that are used by the algorithm to finish (line 20). Finally, if none of the early terminations happened, the buckets stayed bigger than the RAM at the end, leading to no feasible solution. This condition is reflected by setting the cost L to ∞ in line 22. This setting of cost to ∞ will be used for guiding optimization algorithms that we explore in the next subsection. In the end, the procedure returns both the cost and the required $\mathbf{b}$ sequence.

To verify correctness of Algorithm 2, we run bucket sort with a number of predetermined $\mathbf{b}$ combinations, compare the actual I/O with L computed by Algorithm 2, and present the results in Fig. 5.11(a) and (b) for LRU and CLOCK caching, respectively. We use a fixed size RAM of 818 MB, file buffer size $w = 64$ MB, overhead $\mathcal{O} = 71$ bytes/key for LRU and 28 bytes/key for CLOCK. The figure shows that the algorithm is exact for LRU. In case of CLOCK as shown in part (b), the model follows the same dynamics as the actual values. Note that we also experimented with a number of other combinations, among which some (e.g., $(4,4)$, $(4,2)$, $(8,2)$) gave infeasible solutions, while some others (e.g., $(2,2,2)$, $(2,2,4)$, $(9,5)$) gave much higher costs. Furthermore, the minimum cost solution for LRU (i.e., combination

Algorithm 2 Bucket Sort Cost L for heterogeneous b

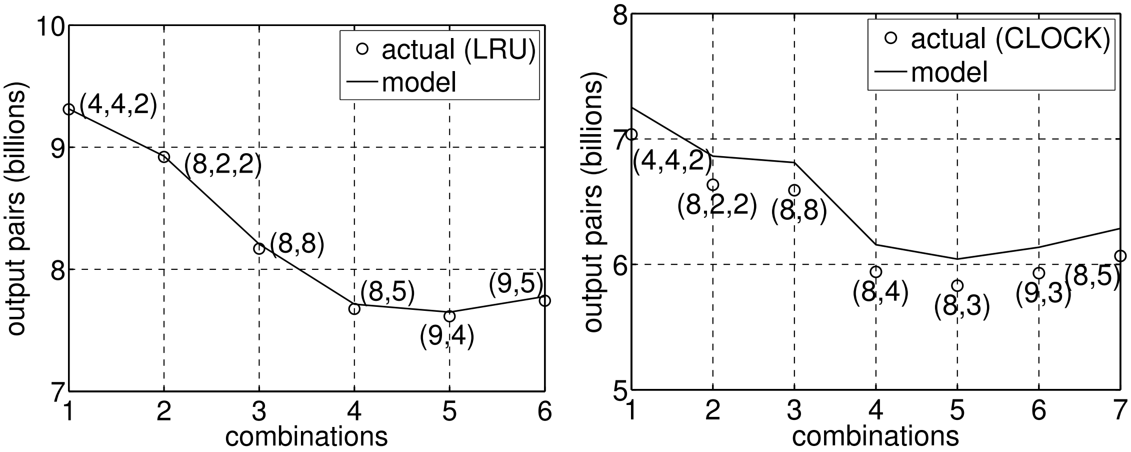
```
procedure BucketSortCost($\mathbf{b}, R, c, w, \mathcal{O}$)
    $L \leftarrow 0, nBuckets \leftarrow 1, i \leftarrow 1, l \leftarrow |\mathbf{b}|, B \leftarrow \mathbf{0}_l$
                                        ▷ $B$ is an array of average bucket sizes in each level
    while $i \leq l$ do
        $h \leftarrow (R - b_i \times w)/(c + \mathcal{O})$
        $m \leftarrow s^{-1}(h \times nBuckets)/nBuckets$
        $B[i] \leftarrow (h + (T/nBuckets - m) \times p(m \times nBuckets))/b_i$
        $B[i] \leftarrow \min(B[i], B[i-1])$
        $nBuckets \leftarrow nBuckets \times b_i$
        $L \leftarrow L + nBuckets \times B[i]$
        if $B[i] \leq R/c$ then                       ▷ Terminate early when the..
            break                                    ▷ bucket size fits in memory
        end if
        if $n/nBuckets \leq (R - w)/(c + \mathcal{O})$ then
                                        ▷ Terminate early when the next iteration's ..
            break                       ▷ cache output fits in RAM without splitting
        end if
        $i \leftarrow i + 1$
    end while
    $\mathbf{b} \leftarrow \{b_1, b_2, \ldots, b_i\}$                  ▷ Truncate $\mathbf{b}$ to length $i$
    if $i > l$ then                               ▷ If the buckets do not fit in RAM,
        $L \leftarrow \infty$                              ▷ then $\mathbf{b}$ is not a feasible solution.
    end if
    return $(L, \mathbf{b})$
end procedure
```

$(9, 4)$) and CLOCK (i.e., combination $(8, 3)$) are different.

Observing these results, it is clear that for a particular configuration (RAM size and caching policy), the disk I/O L varies for each b sequence. Therefore, is there a optimal sequence of split factors that would lead to the least amount of L. This leads us to the next subsection that describes how to choose that sequence.

5.6.5 Optimal Split

We propose Algorithm 3 that chooses the optimal b sequence for a particular configuration (i.e., $R, c, w, \mathcal{O}$). This algorithm consists of 2 procedures. The main procedure is OptimizedSplitFactor(.) in line 9 that is invoked with a configura-



(a) LRU (b) CLOCK

Figure 5.12: Verification of Algorithm 2 for heterogeneous b on IRLbot under LRU and CLOCK.

tion (i.e., R, c, O, w). Then, the maximum required depth of the bucket sort tree is determined using SelectDepth(.) in line 10, which eventually calls the third procedure BucketSortCost(.) from Algorithm 2. After the maximum depth and maximum split factor b_{max} are computed, this algorithm then uses a simplex optimization method (available in MATLAB) with an initial $\mathbf{b}$ vector where each item is set to $b_{max}/2$ and gets back the optimal $\mathbf{b}_{opt}$ vector. Finally, the cost function BucketSortCost(.) is invoked again with $\mathbf{b}_{opt}$ to obtain the cost.

To verify Algorithm 3, we run it under the configuration $R = 818$ MB, $c = 12$ bytes, $\mathcal{O} = 28$ bytes/key, and $w = 64$ MB as before and obtain the optimal sequence to be $(9, 4)$ and $(8, 3)$ for LRU and CLOCK, respectively. Note that these match our findings in Fig. 5.12 (a) and (b). We also experiment with a number of other combinations and find the above points representing the respective global minimum among all possible combinations.

Algorithm 3 Computing Optimal Bucket Sort Cost

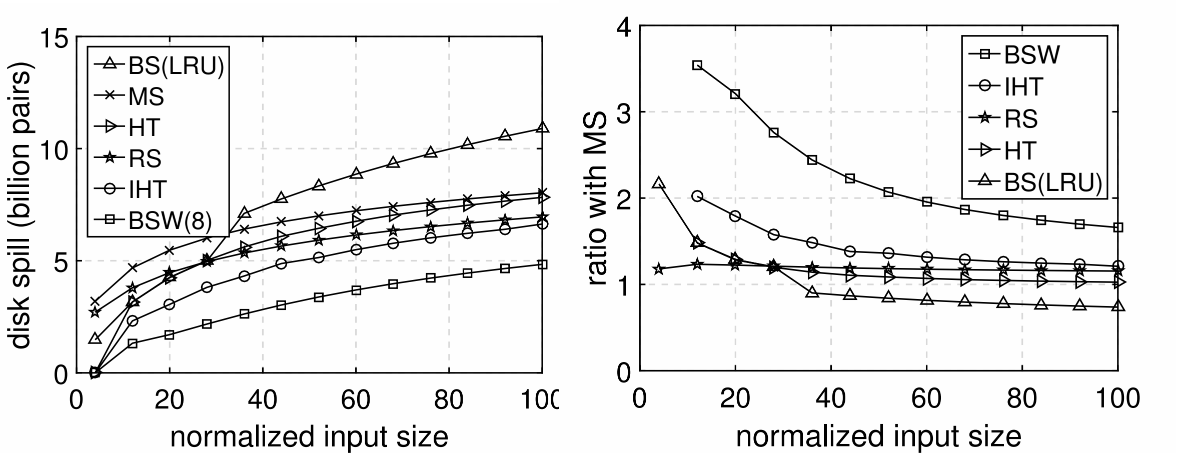
```
procedure SelectDepth($R, c, w, \mathcal{O}$)
    $\mathbf{b} \leftarrow \emptyset, L \leftarrow \infty$          ▷ Initialize b as an empty multi-set
    while $L = \infty$ do
        $\mathbf{b} \leftarrow \mathbf{b} \cup \{2\}$          ▷ Concatenate 2 to the multi-set
        $L \leftarrow$ BucketSortCost($\mathbf{b}, R, c, w, \mathcal{O}$)
    end while
    return $|\mathbf{b}|$
end procedure
procedure OptimalBucketSortCost($R, c, w, \mathcal{O}$)
    $l \leftarrow$ SelectDepth($R, c, w, \mathcal{O}$)
    $b_{max} \leftarrow (R - c - \mathcal{O})/w$          ▷ $b_{max}$ is the maximum possible split
                                                          ▷ factor allowed by the configuration
    $\mathbf{b} \leftarrow b_{max} \times \mathbf{1}_l/2$          ▷ b is a vector of $l$ items each equal $b_{max}/2$
    $\mathbf{b}_{opt} \leftarrow$ Optimize(BucketSortCost, $\mathbf{b}, R, c, w, \mathcal{O}$)
    $(L, \mathbf{b}_{opt}) \leftarrow$ BucketSortCost($\mathbf{b}_{opt}, R, c, w, \mathcal{O}$)
    return $L$
end procedure
```

5.7 Comparisons

In this section, we present a performance comparison among the classic and the newly proposed MapReduce designs. We compare these techniques in consideration of their resource disk I/O, which is an important metric in determining their runtime. To enable a fair comparison, similar to the previous chapter, we allow the same amount of RAM for each method and convert that number of items that fit in it using the following:

$$h = \frac{R - fw}{c + \mathcal{O}}, \tag{5.39}$$

where c bytes are taken for each key-value pair, f is the number of required file handles, w file buffer size, and $\mathcal{O}$ is the data structure overhead. We run the experiments on different workloads, RAM sizes, and different parameter choices (e.g., $r = 2, 4, 8$



(a) absolute (b) normalized by MS

Figure 5.13: Comparison of disk I/O various MapReduce designs with respect to merge sort.

for WATCH) and verify that our model is accurate in all cases. Therefore, in this section, we just use the model values rather than actual experiment values.

We compute the disk I/O model for each of the classic and advanced methods on the IRLbot graph and show the results in Fig. 5.13(a) and (b), where the first part shows actual (model) values and the second shows it in comparison with MS (ratio of L between MS and the method under consideration). Therefore, in part (b), the higher a curve, the better is the method compared to MS.

Observe that the BS(LRU) produces the most data, while BS with WATCH produces the least. This shows how reducing the data structure overhead can dramatically affect I/O performance of these methods. After BS(LRU), the next highest is MS, and then, HT, RS, and IHT produce progressively less I/O. Note that IHT, our newly proposed method, would produce the least amount of I/O among all load-sort-merge based methods also the fastest among them (see Table 5.3. In addition, HT and MS tend to converge towards each other, while the same is true for the pair

IHT and RS. The reason is as the RAM gets smaller, the de-duplication ability of HT based methods (HT and IHT) start diminishing, while because of their overhead RS and MS start catching up. Ultimately, the MS-HT and RS-IHT both pairs converge, but potentially at different points, where the difference is mainly due to RS-IHT's ability to double the effective RAM size. This is also evident from the Figure since the disk spill of IHT at NIS=80 is same as that of HT at NIS=40, while the same is true for the MS-RS pair.

Then, Fig. 4.7(b) shows the same results, but this time normalized by the disk I/O of merge sort. This result shows also confirms that IHT is the best load-sort-merge method in terms of disk I/O and it is at some point uses 2 times less I/O than MS. On the other hand, the use of low-overhead WATCH makes BS the best among all methods (i.e., as high as 3.5 times better than MS), while the basic BS with the traditional LRU was not such a good choice.

To investigate the behavior of these methods more closely, we examine them on two synthetic Zipf graphs, both with 5M edges, $\alpha = 1.5$, and average degree being 5 and 50 respectively. The results are shown in Fig. 4.8, where we compare the performance of other methods using MS as the baseline. Observer in Fig. 4.8(a) that when the repetition is less (average degree 5), IHT's deduplication ability reduces to the point where its overhead becomes noticeable. As a result it falls below the RS that does not have any overhead. Similarly, HT also looses to MS (ratio being less than 1) because of the same effect.

In Fig. 4.8(b), observe that IHT is much better than MS compared to Fig. 4.8(a) with a maximum of 8× and ending at 2× better than MS. This indicates how IHT can perform significantly better than other methods depending on the repetition factor (or average frequency) of the data items. Furthermore, RS is slightly better than, but still very close to MS in this case.

Table 5.3: Sorting and aggregation speed of different methods.

method	MS	HT	RS	IHT
speed (keys/sec)	9M	20M	3M	12M

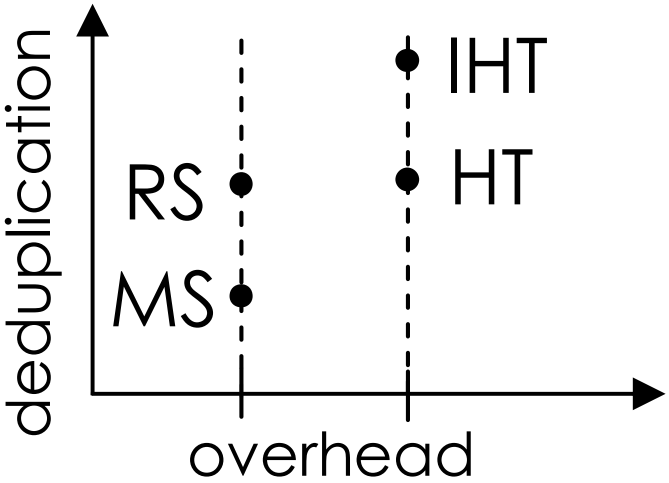


Figure 5.14: Duality between MS-HT and RS-IHT pairs.

Furthermore, to examine the practicality of these methods, we run each of these methods on the first 1B edges of IRLbot, record the lookups/insertion rates, and collect the results in Table 5.3. Note that while calculating speed, we include the time for interaction with the disk (i.e., read time and write time), and resetting the data structures when a run ends (i.e., time to re-initialize a hash table after the content is written to disk). First, observe that RS is slowest among all, while HT is the fastest one. This somewhat explains why although being I/O efficient, RS has not been used in many systems. Second, from this speed and other characteristics of these methods, we observe some duality between the pairs MS-HT and RS-IHT, which is elaborated in Fig. 5.14.

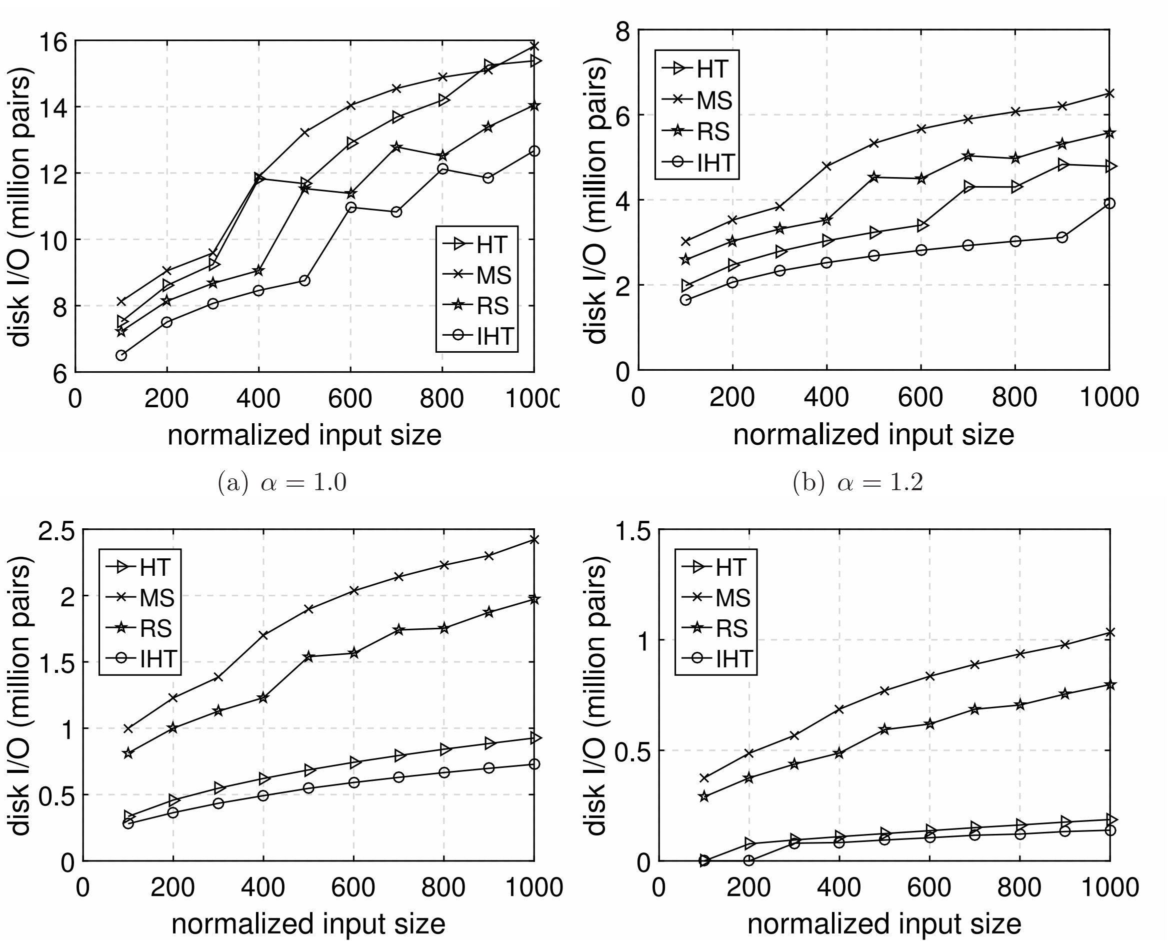
Note that the MS is located at the left-bottom corner of the figure where the disk

I/O is the highest, because the deduplication rate is the lowest. The amount of disk I/O reduces as we either move up (RS), or to the right (HT), or both (IHT). The methods on the right (i.e., HT and IHT) support $O(1)$ aggregation at the cost of added overhead to maintain the HT. But experiments in this paper suggest that the advantage of fast lookup surpasses the disadvantage of having 8 byets of overhead per key that these methods require. Furthermore, for MS and HT, each run is produced by evicting all items to the disk at a single shot. Contrary to that, RS and IHT both have an eviction mechanism that removes one item at a time to make room for the next incoming item.

To investigate frequency-skew's effect on the outcome of these algorithms, we run them all on synthetic Zipf distributed streams and show the results in Fig. 5.15. For all these experiments, we have used a stream length $T = 10M$ and average frequency $E[D] = 5$, but varied the α parameter from 1.0 to 1.6, which correspond to increasing degree of skew in the frequency distribution. It is clear that all these methods are more efficient with more skew – $\alpha = 1.6$ results in the least amount of intermediate data L, while $\alpha = 1.0$ causes the highest. In addition, the advantage of having either extra overhead for HT and IHT pays off much more in case of higher skew (Fig. 5.15(d)) compared to the case of low skew Fig. 5.15(a)).

5.8 Conclusion

In this chapter, we proposed a number of advanced MapReduce techniques. First, we propose Incremental Hash Table and present detailed performance analysis for it. Then, we propose a new caching method that takes only 1 byte/key, works much faster than existing caching methods (e.g., LRU, CLOCK), and shows hit rate comparable to LRU and CLOCK. We also show that when this caching method is incorporated with bucket sort, it produces the least amount of intermediate data



(a) $\alpha = 1.0$

(b) $\alpha = 1.2$

(c) $\alpha = 1.4$

(d) $\alpha = 1.6$

Figure 5.15: Comparison on synthetic Zipf graphs with $T = 10\text{M}$, $E[D] = 5$.

compared to other alternatives (e.g., hash sort, merge sort).

6. SUMMARY AND FUTURE WORK

6.1 Summary

The true motivation of this dissertation is efficient processing of Internet-scale information in a scalable manner. We approach this goal in steps which leads to the division of our work in the following major areas – a general stream modeling framework, modeling of external-memory algorithm performance with their different components (e.g., LRU, random replacement cache) and crawl analysis of random graphs.

In the first part, we develop a modeling framework for shuffled stream of data. We define a metric called *uniqueness probability* as the probability to find a new key while processing a stream of incoming data where the keys from a finite set are repeated, and then quantify this metric in terms of the frequency distribution of the keys and the processing length (i.e., how many keys have been processed so far). This is the central metric of interest which leads us to the size of the seen set (i.e., the set containing all the nodes seen so far) after a certain processing length. We verify that the models are accurate on both synthetic and real stream of data.

An interesting application of stream modeling is the experience modeling of a crawling process on random graphs. When a crawler is designed, there are some important components of the crawler that the designers must plan for. For instance, the growth rate of the seen set is required to determine how fast the other components of the crawler have to work. To prioritize the frontier set, it is important to know how big its size will be at a particular stage of of the crawling process. Additionally, given an incomplete sample of the graph, it may be necessary to forecast various metrics of the underlying graph. We present models for all these features in our crawl modeling section when the graph is random and does not have any clustering.

Another application of the stream modeling is to determine the amount of intermediate data produced by an external-memory merge-sort program given a particular RAM capacity. This leads to our second topic, where we analyze a number of well-known external sorting and aggregation methods (e.g., hash sort, merge sort, replacement selection, bucket sort) with consideration of the respective data-structure overheads for their implementations, which has not been done in existing literature. In this process, we develop accurate models for replacement selection, where using a hash table with eviction mechanism, it is possible to attain nearly twice the run-length that would be possible compared to hash- and merge-sort methods. This is a novel method which is much faster than existing two-queue based method for replacement selection. In addition, this is the first in the literature that performs data aggregation in case of duplicate keys. Furthermore, we devise accurate models for the run lengths, number of runs, and miss rate of this new data structure, which is novel, since all existing models of replacement selection are only applicable to random/all-unique keys. Furthermore, we compare fairly among all these sorting methods under fixed RAM sizes and show how each of their performance. We also show cases where one methods performs better than the others based on the particular hardware characteristics (e.g., CPU or disk speed).

A key takeaway from the above is that space overhead of each data structure deeply impact the eventual amount of intermediate data produced by each method. This understanding motivated us to investigate the very attractive bucket sort program in tandem with the underlying caching mechanism. It is well-known that conventional LRU is inefficient in both speed and per-key overhead. The traditional solution to that is CLOCK which still leads to more than 8 bytes per key overhead. To solve this problem, we devise a new caching mechanism called WATCH, where the overhead is only 1 byte for each key, while the hit rate is very similar and sometimes exceeding that of LRU. We use this new cache with bucket sort and show that this new sorting method is the best under a wide hardware resource spectrum. We show these result in the final part of this dissertation.

6.2 Future Work

The proposed work has a number of promising future directions. First, it would be interesting to investigate the relationship between uniqueness probability and the clustering coefficient of a graph. Since the difference between random graphs and real graphs are mainly the clustering along with a number of other features (e.g., small world effect, scale-free effect), this investigation has a great deal of potential to eventually establish similar measures for real graphs as well.

It is possible to adapt our graph models to probabilistic graphs to formulates problems as information/rumor spreading. This would enable a new paradigm to solve such problems. In our crawling analysis, we have so far only discussed BFS and frontier-randomization crawling. But, the formulation can be generalized to other more realistic graph crawling methods (e.g., frontier ranking in some importance metric). As a result, those more realistic crawls can also be analyzed.

In this dissertation, we have considered the external-memory algorithms only in terms of their intermediate data volume. The last part of the work presents optimized bucket sort where the optimization criteria is mainly the intermediate data. But if we consider runtime, then there are a number of other features that we have to consider. For instance, in case of hash sort, when the number of buckets is small the overhead is less, leading to less intermediate data because the RAM has more space to put the actual data. On the other hand, this is not optimal in terms of speed, because when hashing speed is higher when the hash table is sparse (i.e., there are more buckets than actual number data). Similarly, for bucket sort, the more the split factor beyond a threshold, the less is the sorting speed. Therefore, in a fully optimized system would try to choose the right number of buckets or the correct split factor to strike a balance between the obtained speed and produced intermediate data. Future work of this version will potentially consider such complicated optimization problem and determine the best software combination under a given setting (i.e., hardware properties and data distribution). Furthermore, all these external memory models are yet to be tested in cluster setups.

REFERENCES

[1] “CLuE Cluster,” 2008. [Online]. Available: http://www.nsf.gov/cise/clue/index.jsp.

[2] S. Abiteboul, M. Preda, and G. Cobena, “Adaptive On-Line Page Importance Computation,” in *Proc. WWW*, May 2003, pp. 280–290.

[3] S. T. Ahmed and D. Loguinov, “Modeling Randomized Data Streams in Caching, Data Processing, and Crawling Applications,” in *Proc. IEEE INFOCOM*, Apr. 2015, pp. 1625–1633.

[4] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, “Hadoop GIS: A High Performance Spatial Data Warehousing System over Mapreduce,” *VLDB Endow.*, vol. 6, no. 11, pp. 1009–1020, Aug. 2013.

[5] M. H. Alam, J. Ha, and S. Lee, “Novel Approaches to Crawling Important Pages Early,” *Springer Knowledge and Information Systems*, Sep. 2012.

[6] J. Alpert and N. Hajaj, “We Knew the Web Was Big...” Jul. 2008. [Online]. Available: http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html.

[7] A. Aly, A. Sallam, B. Gnanasekaran, L. Nguyen-Dinh, W. Aref, M. Ouzzani, and A. Ghafoor, “M3: Stream Processing on Main-Memory MapReduce,” in *Proc. IEEE ICDE*, Apr. 2012, pp. 1253–1256.

[8] S. Babu, “Towards Automatic Optimization of MapReduce Programs,” in *Proc. ACM SoCC*, Jun. 2010, pp. 137–142.

[9] R. Baeza-Yates and C. Castillo, “Crawling the Infinite Web: Five Levels are Enough,” in *Proc. WAW*, 2004, pp. 156–167.

[10] R. Baeza-Yates, C. Castillo, M. Marin, and A. Rodriguez, “Crawling a Country: Better Strategies than Breadth-First for Web Page Ordering,” in *Proc. WWW*, May 2005.

[11] S. Bansal and D. S. Modha, “CAR: Clock with Adaptive Replacement,” in *Proc. USENIX FAST*, Mar. 2004, pp. 187–200.

[12] L. Becchetti, C. Castillo, D. Donato, A. Fazzone, and I. Rome, “A Comparison of Sampling Techniques for Web Graph Characterization,” in *Proc. LinkKDD*, Aug. 2006.

[13] J. Berlinska and M. Drozdowski, “Scheduling Divisible MapReduce Computations,” *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 450–459, Mar. 2011.

[14] K. Bharat, B.-W. Chang, M. Henzinger, and M. Ruhl, “Who Links to Whom: Mining Linkage between Web Sites,” in *Proc. IEEE ICDM*, Nov. 2001.

[15] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, “A Comparison of Join Algorithms for Log Processing in MaPreduce,” in *Proc. ACM SIGMOD*, Jun. 2010, pp. 975–986.

[16] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, “Structural Properties of the African Web,” in *Proc. WWW*, 2002.

[17] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, “UbiCrawler: A Scalable Fully Distributed Web Crawler,” *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, Jul. 2004.

[18] S. Brin and L. Page, “The Anatomy of a Large-Scale Hypertextual Web Search Engine,” in *Proc. WWW*, Apr. 1998, pp. 107–117.

[19] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, “Graph Structure in the Web,” *Computer Networks*, vol. 33, pp. 309–320, Jun. 2000.

[20] A. Z. Broder, M. Najork, and J. L. Wiener, “Efficient URL Caching for World Wide Web Crawling,” in *Proc. WWW*, May 2003, pp. 679–689.

[21] M. Burner, “Crawling Towards Eternity: Building an Archive of the World Wide Web,” *Web Techniques Magazine*, vol. 2, no. 5, May 1997.

[22] J. Callan, M. Hoy, C. Yoo, and L. Zhao, “The ClueWeb09 Dataset,” Nov. 2009. [Online]. Available: http://boston.lti.cs.cmu.edu/classes/11-742/S10-TREC/TREC-Nov19-09.pdf.

[23] R. W. Carr and J. L. Hennessy, “WSCLOCK&Mdash; A Simple and Effective Algorithm for Virtual Memory Management,” in *Proc. ACM SOSP*, Dec. 1981, pp. 87–95.

[24] C. Castillo, M. Marin, A. Rodriguez, and R. Baeza-Yates, “Scheduling Algorithms for Web Crawling,” in *Proc. Latin American Web Conference*, Oct. 2004.

[25] A. Chandramouli, S. Gauch, and J. Eno, “A Popularity-Based URL Ordering Algorithm for Crawlers,” in *Proc. IEEE HSI*, May. 2010, pp. 259–263.

[26] H. Che, Y. Tung, and Z. Wang, “Hierarchical Web Caching Systems: Modeling, Design and Experimental Results,” *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 7, pp. 1305–1314, Sep. 2006.

[27] J. Cho and H. Garcia-Molina, “Parallel Crawlers,” in *Proc. WWW*, May 2002, pp. 124–135.

[28] J. Cho, H. Garcia-Molina, T. Haveliwala, W. Lam, A. Paepcke, and S. R. G. Wesley, “Stanford WebBase Components and Applications,” *ACM Trans. Internet Technology*, vol. 6, no. 2, pp. 153–186, May 2006.

[29] J. Cho, H. Garcia-Molina, and L. Page, “Efficient Crawling through URL Ordering,” in *Proc. WWW*, Apr. 1998, pp. 161–172.

[30] J. Cho and U. Schonfeld, “RankMass Crawler: A Crawler with High PageRank Coverage Guarantee,” in *Proc. VLDB*, Sep. 2007.

[31] M. Cho, D. Brand, R. Bordawekar, U. Finkler, V. Kulandaisamy, and R. Puri, “PARADIS: An Efficient Parallel Algorithm for In-place Radix Sort,” *VLDB Endow.*, vol. 8, no. 12, pp. 1518–1529, Aug. 2015.

[32] H. Choi, J. Son, H. Yang, H. Ryu, B. Lim, S. Kim, and Y. D. Chung, “Tajo: A Distributed Data Warehouse System on Large Clusters,” in *Proc. IEEE ICDE*, Apr. 2013, pp. 1320–1323.

[33] H.-T. Chou and D. J. DeWitt, “An Evaluation of Buffer Management Strategies for Relational Database Systems,” in *Proc. VLDB*, Aug. 1985, pp. 127–141.

[34] C. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, “Map-Reduce for Machine Learning on Multicore,” *Advances In Neural Information Processing Systems*, vol. 19, pp. 281–288, Dec. 2006.

[35] A. Chudnovsky, “Majestic12 C# HTML Parser,” Aug. 2007. [Online]. Available: http://www.majestic12.co.uk/projects/html_parser.php.

[36] J. Cieslewicz and K. A. Ross, “Adaptive Aggregation on Chip Multiprocessors,” in *Proc. VLDB*, Sep. 2007, pp. 339–350.

[37] ClueWeb09 Dataset. [Online]. Available: http://www.lemurproject.org/clueweb09/.

[38] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online," in *Proc. USENIX NSDI*, Apr. 2010, pp. 21–21.

[39] F. J. Corbato, "A Paging Experiment with the Multics System," MIT MAC Report, Tech. Rep. MAC-M-384, May 1968.

[40] A. Dan and D. Towsley, "An Approximate Analysis of The LRU And FIFO Buffer Replacement Schemes," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 18, no. 1, pp. 143–152, May 1990.

[41] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. USENIX OSDI*, Dec. 2004, pp. 137–150.

[42] D. Eichmann, "The RBSE Spider – Balancing Effective Search Against Web Load," in *Proc. WWW*, May 1994.

[43] N. Eiron, K. S. McCurley, and J. A. Tomlin, "Ranking the Web Frontier," in *Proc. WWW*, May 2004, pp. 309–318.

[44] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A Runtime for Iterative MapReduce," in *Proc. ACM HPDC*, Jun. 2010, pp. 810–818.

[45] El-Kabong Project. [Online]. Available: http://sourceforge.net/projects/ekhtml/.

[46] I. Elghandour and A. Aboulnaga, "ReStore: Reusing Results of MapReduce Jobs," *VLDB Endow.*, vol. 5, no. 6, pp. 586–597, Feb. 2012.

[47] T. Espelid, "On Replacement Selection and Dinsmore's Improvement," *BIT Numerical Mathematics*, vol. 16, no. 2, pp. 133–142, Jun. 1976.

[48] Fast, Robust Html Parser. [Online]. Available: http://htmlparser.sourceforge.net/.

[49] A. Fedoniouk, "Fast and Compact HTML/XML Scanner/Tokenizer," Oct. 2007. [Online]. Available: http://www.codeproject.com/KB/recipes/HTML_XML_Scanner.aspx.

[50] P. Ferrera, I. de Prado, E. Palacios, J. Fernandez-Marquez, and G. Di Marzo Serugendo, "Tuple MapReduce: Beyond Classic MapReduce," in *Proc. IEEE ICDM*, Dec. 2012, pp. 260–269.

[51] E. H. Friend, "Sorting on Electronic Computer Systems," *J. ACM*, vol. 3, no. 3, pp. 134–168, Jul. 1956.

[52] M. Gallo, B. Kauffmann, L. Muscariello, A. Simonian, and C. Tanguy, "Performance Evaluation of The Random Replacement Policy for Networks of Caches," in *Proc. ACM SIGMETRICS*, Jun. 2012, pp. 395–396.

[53] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a High-Level Dataflow System on Top of Map-Reduce: The Pig Experience," *VLDB Endow.*, vol. 2, no. 2, pp. 1414–1425, Aug. 2009.

[54] E. Gelenbe, "A Unified Approach to the Evaluation of a Class of Replacement Algorithms," *IEEE Trans. Computers*, vol. C-22, no. 6, pp. 611–618, Jun. 1973.

[55] D. Gibson, R. Kumar, K. S. Mccurley, and A. Tomkins, "Dense Subgraph Extraction," in *Mining Graph Data*, D. J. Cook and L. B. Holder, Eds. Wiley, 2007, pp. 411–441.

[56] GNU wget. [Online]. Available: http://www.gnu.org/software/wget/.

[57] M. A. Golshani, V. Derhami, and A. ZarehBidoki, "A Novel Crawling Algorithm for Web Pages," in *Proc. AIRS*, 2011, pp. 263–272.

[58] D. Gruhl, L. Chavet, D. Gibson, J. Meyer, P. Pattanayak, A. Tomkins, and J. Zien, "How to Build a WebFountain: An Architecture for Very Large-Scale Text Analytics," *IBM Systems Journal*, vol. 43, no. 1, pp. 64–77, 2004.

[59] D. Gruhl, D. N. Meredith, and J. H. Pieper, "The Web Beyond Popularity," in *Proc. WWW*, May 2006, pp. 183–192.

[60] S. Hammoud, M. Li, Y. Liu, N. Alham, and Z. Liu, "MRSim: A Discrete Event Based MapReduce Simulator," in *Proc. FSKD*, vol. 6, Aug. 2010, pp. 2993–2997.

[61] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce Framework on Graphics Processors," in *Proc. PACT*, Oct. 2008, pp. 260–269.

[62] S. Helmer, T. Neumann, and G. Moerkotte, "Estimating the Output Cardinality of Partial Preaggregation with a Measure of Clusteredness," in *Proc. VLDB*, vol. 29, Sep. 2003, pp. 656–667.

[63] H. Herodotou, "Hadoop Performance Models," Duke University, Tech. Rep. CS-2011-05, Jun. 2011.

[64] A. Heydon and M. Najork, "Mercator: A Scalable, Extensible Web Crawler," *World Wide Web*, vol. 2, no. 4, pp. 219–229, Dec. 1999.

[65] Y. Hirate, S. Kato, and H. Yamana, "Web Structure in 2005," in *Proc. WAW*, Nov. 2006, pp. 36–46.

[66] M.-J. Hsieh, C.-R. Chang, L.-Y. Ho, J.-J. Wu, and P. Liu, "SQLMR: A Scalable Database Management System for Cloud Computing," in *Proc. IEEE ICPP*, Sep. 2011, pp. 315–324.

[67] L. Huang, J. J. H. Zhu, and X. Li, "Histrace: Building a Search Engine of Historical Events," in *Proc. WWW Poster Session*, Apr. 2008, pp. 1155–1156.

[68] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis," in *Proc. IEEE ICDEW*, Mar. 2010, pp. 41–51.

[69] S. Ibrahim, H. Jin, L. Lu, L. Qi, S. Wu, and X. Shi, "Evaluating MapReduce on Virtual Machines: The Hadoop Case," in *Proc. CloudCom*, Dec. 2009, pp. 519–528.

[70] Internet Archive. [Online]. Available: http://archive.org/.

[71] IRLbot Project at Texas A&M. [Online]. Available: http://irl.cs.tamu.edu/crawler/.

[72] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," in *Proc. ACM SIGOPS/EuroSys*, Mar. 2007, pp. 59–72.

[73] P. R. Jelenkovic, "Asymptotic Approximation of the Move-to-Front Search Cost Distribution and Least-Recently Used Caching Fault Probabilities," *The Annals of Applied Probability*, vol. 9, no. 2, pp. 430–464, 1999.

[74] Q. Jiang and Y. Zhang, "SiteRank-Based Crawling Ordering Strategy for Search Engines," in *Proc. CIT*, Oct. 2007, pp. 259–263.

[75] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement," in *Proc. USENIX ATEC*, Apr. 2005, pp. 323–336.

[76] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance," in *Proc. ACM SIGMETRICS*, Jun. 2002, pp. 31–42.

[77] D. Jiménez-González, J. J. Navarro, and J.-L. Larrba-Pey, “Fast Parallel In-Memory 64-bit Sorting,” in *Proc. ACM ICS*, Jun. 2001, pp. 114–122.

[78] T. Johnson and D. Shasha, “2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm,” in *Proc. VLDB*, Sep. 1994, pp. 439–450.

[79] D. E. Knuth, “Length of Strings for a Merge Sort,” *Commun. ACM*, vol. 6, no. 11, pp. 685–688, Nov. 1963.

[80] E. Krevat, T. Shiran, E. Anderson, J. Tucek, J. J. Wylie, and G. R. Ganger, “Applying Performance Models to Understand Data-Intensive Computing Efficiency,” CMU, Tech. Rep. CMU-PDL-10-108, May 2010.

[81] M. Kurant, A. Markopoulou, and P. Thiran, “Towards Unbiased BFS Sampling,” *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1799–1809, Oct. 2011.

[82] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, “SkewTune: Mitigating Skew in Mapreduce Applications,” in *Proc. ACM SIGMOD*, May 2012, pp. 25–36.

[83] P. Lama and X. Zhou, “AROMA: Automated Resource Allocation and Configuration of Mapreduce Environment in the Cloud,” in *Proc. ACM ICAC*, Sep. 2012, pp. 63–72.

[84] P.-A. Larson, “Grouping and Duplicate Elimination: Benefits of Early Aggregation,” Microsoft Research, Tech. Rep. MSR-TR-97-36, Dec. 1997.

[85] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, “On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies,” in *Proc. ACM SIGMETRICS*, May 1999, pp. 134–143.

[86] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, "IRLbot: Scaling to 6 Billion Pages and Beyond," in *Proc. WWW*, Apr. 2008, pp. 427–436.

[87] H.-T. Lee, D. Leonard, X. Wang, and D. Loguinov, "IRLbot: Scaling to 6 Billion Pages and Beyond," *ACM Trans. Web*, vol. 3, no. 3, pp. 1–34, Jun. 2009.

[88] J. Lee and G. de Veciana, "Resource and Topology Discovery for IP Multicast Using a Fan-Out Decrement Mechanism," in *Proc. IEEE INFOCOM*, pp. 1627–1635.

[89] S. H. Lee, P.-J. Kim, and H. Jeong, "Statistical Properties of Sampled Networks," *Physical Rev. E*, vol. 73, no. 1, p. 016102, Jan. 2006.

[90] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A Platform for Scalable One-Pass Analytics Using Mapreduce," in *Proc. ACM SIGMOD*, Jun. 2011, pp. 985–996.

[91] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang, "MOON: MapReduce On Opportunistic eNvironments," in *Proc. ACM HPDC*, Jun. 2010, pp. 95–106.

[92] B. T. Loo, S. Krishnamurthy, and O. Cooper, "Distributed Web Crawling over DHTs," EECS Dept., University of California, Berkeley, Tech. Rep. UCB/CSD-04-1305, 2004.

[93] J. loup Gailly and M. Adler, "Zlib Compression Library," 2005. [Online]. Available: http://www.zlib.net/.

[94] L. Lovász, "Random Walks on Graphs: A Survey," in *Combinatorics, Paul Erdös is Eighty*, D. Miklós et al., Ed. János Bolyai Mathematical Society, Budapest, 1996, vol. 2, pp. 353–398.

[95] P. Lu, Y. C. Lee, C. Wang, B. B. Zhou, J. Chen, and A. Zomaya, "Workload Characteristic Oriented Scheduler for MapReduce," in *Proc. IEEE ICPADS*, Dec. 2012, pp. 156–163.

[96] O. A. McBryan, "GENVL and WWWW: Tools for Taming the Web," in *Proc. WWW*, May 1994.

[97] J. McCabe, "On Serial Files with Relocatable Records," *Operations Research*, vol. 13, pp. 609–618, Jul. 1965.

[98] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proc. USENIX FAST*, Mar. 2003, pp. 115–130.

[99] D. G. Merrill and A. S. Grimshaw, "Revisiting Sorting for GPGPU Stream Architectures," in *Proc. ACM PACT*, Sep. 2010, pp. 545–546.

[100] M. Molloy and B. Reed, "A Critical Point for Random Graphs with a Given Degree Sequence," *Random Structures and Algorithms*, vol. 6, no. 2/3, pp. 161–180, Mar./May 1995.

[101] K. Moody and M. Palomino, "SharpSpider: Spidering the Web through Web Services," in *Proc. IEEE LA-WEB*, Nov. 2003, pp. 219–221.

[102] E. F. Moore, Patent US 2 983 904 columns 3-4, 1961.

[103] Mozilla, "Public Suffix List," May 2013. [Online]. Available: http://publicsuffix.org/.

[104] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber, "Cache-Efficient Aggregation: Hashing Is Sorting," in *Proc. ACM SIGMOD*, May 2015, pp. 1123–1136.

[105] M. Najork and A. Heydon, “High-Performance Web Crawling,” Compaq SRC, Tech. Rep. 173, Sep. 2001. [Online]. Available: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-173.pdf.

[106] M. Najork and J. L. Wiener, “Breadth-First Search Crawling Yields High-Quality Pages,” in *Proc. WWW*, May 2001, pp. 114–118.

[107] Nutch. [Online]. Available: http://nutch.apache.org/.

[108] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, “MRShare: Sharing Across Multiple Queries in MapReduce,” *VLDB Endow.*, vol. 3, no. 1-2, pp. 494–505, Sep. 2010.

[109] E. J. O’Neil, P. E. O’Neil, and G. Weikum, “The LRU-K Page Replacement Algorithm for Database Disk Buffering,” in *Proc. ACM SIGMOD*, May 1993, pp. 297–306.

[110] J. Padmanabhan, “Introduction to Grid Computing via Map-Reduce & Hadoop,” Dec. 2010. [Online]. Available: http://internationalnetworking.iu.edu/sites/internationalnetworking.iu.edu/files/indo-us-preso.pdf.

[111] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, “A Comparison of Approaches to Large-scale Data Analysis,” in *Proc. ACM SIGMOD*, Jun. 2009, pp. 165–178.

[112] S. Pearson, “Replacement Selection with Duplicate Key Handling,” Patent US 20 030 061 191 A1, Mar., 2003.

[113] D. Perino and M. Varvello, “A Reality Check for Content Centric Networking,” in *Proc. ACM SIGCOMM Workshop on ICN*, Aug. 2011, pp. 44–49.

[114] B. Pinkerton, “Finding What People Want: Experiences with the Web Crawler,” in *Proc. WWW*, Oct. 1994.

[115] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *Proc. IEEE HPCA*, Feb. 2007, pp. 13–24.

[116] M. Richardson, A. Prakash, and E. Brill, "Beyond PageRank: Machine Learning for Static Ranking," in *Proc. WWW*, May 2006, pp. 707–715.

[117] J. T. Robinson and M. V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," in *Proc. ACM SIGMETRICS*, May 1990, pp. 134–142.

[118] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort," in *Proc. ACM SIGMOD*, Jun. 2010, pp. 351–362.

[119] F. M. Schuhknecht, P. Khanchandani, and J. Dittrich, "On the Surprising Difficulty of Simple Things: The Case of Radix Partitioning," *VLDB Endow.*, vol. 8, no. 9, pp. 934–937, May 2015.

[120] P. Sethi and P. Kumar, "Leveraging Hadoop Framework to Develop Duplication Detector and Analysis Using Mapreduce, Hive and Pig," in *Proc. IC3*, Aug. 2014, pp. 454–460.

[121] B. Sharma, R. Prabhakar, S. Lim, M. Kandemir, and C. Das, "MROrchestrator: A Fine-Grained Resource Orchestration Framework for MapReduce Clusters," in *Proc. IEEE CLOUD*, Jun. 2012, pp. 1–8.

[122] V. Shkapenyuk and T. Suel, "Design and Implementation of a High-Performance Distributed Web Crawler," in *Proc. IEEE ICDE*, Mar. 2002, pp. 357–368.

[123] A. Signhal, “Breakfast with Google’s Search Team,” Aug. 2012. [Online]. Available: https://www.youtube.com/watch?v=8a2VmxqFg8A.

[124] A. Singh, M. Srivatsa, L. Liu, and T. Miller, “Apoidea: A Decentralized Peer-to-Peer Architecture for Crawling the World Wide Web,” in *Proc. SIGIR Workshop on Distributed Information Retrieval*, Aug. 2003, pp. 126–142.

[125] A. J. Smith, “Sequentiality and Prefetching in Database Systems,” *ACM Trans. Database Syst.*, vol. 3, no. 3, pp. 223–247, Sep. 1978.

[126] C. Sparkman, H.-T. Lee, and D. Loguinov, “Agnostic Topology-Based Spam Avoidance in Large-Scale Web Crawls,” in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 811–819.

[127] R. J. Stewart, P. W. Trinder, and H.-W. Loidl, “Comparing High Level Mapreduce Query Languages,” in *Advanced Parallel Processing Technologies*, 2011, pp. 58–72.

[128] T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasundaram, “ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval,” in *Proc. WebDB*, Jun. 2003, pp. 67–72.

[129] J. Talbot, R. M. Yoo, and C. Kozyrakis, “Phoenix++: Modular MapReduce for Shared-memory Systems,” in *Proc. International Workshop on MapReduce and Its Applications*, Jun. 2011, pp. 9–16.

[130] The Stanford WebBase Project. [Online]. Available: http://dbpubs.stanford.edu:8091/~testbed/doc2/WebBase/.

[131] The Stanford WebBase Project, “WebBase Archive,” Sep. 2008. [Online]. Available: http://dbpubs.stanford.edu:8091/~testbed/doc2/WebBase/.

[132] M. Thelwall and D. Wilkinson, “Graph Structure in Three National Academic Webs: Power Laws with Anomalies,” *J. Am. Soc. Inf. Sci. Technol.*, vol. 54, no. 8, pp. 706–712, Jun. 2003.

[133] S. Thiel, L. Thiel, and G. Butler, “Relaxing the Counting Requirement for Least Significant Digit Radix Sorts,” Concordia University, Tech. Rep., 2015. [Online]. Available: http://users.encs.concordia.ca/~sthiel/DS/SEA2015_FastRadix.pdf.

[134] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, “Hive - A Petabyte Scale Data Warehouse Using Hadoop,” in *Proc. IEEE ICDE*, Mar. 2010, pp. 996–1005.

[135] F. Tian and K. Chen, “Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds,” in *Proc. IEEE CLOUD*, Jul. 2011, pp. 155–162.

[136] D. Tiwari and D. Solihin, “Modeling and Analyzing Key Performance Factors of Shared Memory MapReduce,” in *Proc. IEEE IPDPS*, May 2012, pp. 1306–1317.

[137] A. Verma, L. Cherkasova, and R. H. Campbell, “ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments,” in *Proc. ACM ICAC*, Jun. 2011, pp. 235–244.

[138] X. Wang, Z. Yao, and D. Loguinov, “Residual-Based Estimation of Peer and Link Lifetimes in P2P Networks,” *IEEE/ACM Trans. Networking*, vol. 17, no. 3, pp. 726–739, Jun. 2009.

[139] J. Wassenberg and P. Sanders, “Engineering a Multi-Core Radix Sort,” *Euro Parallel Processing*, vol. 6853, pp. 160–169, 2011.

[140] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi, “Query Optimization for Massively Parallel Data Processing,” in *Proc. ACM SoCC*, Oct. 2011, pp. 12:1–12:13.

[141] Yahoo Hadoop. [Online]. Available: http://developer.yahoo.com/hadoop/.

[142] H. Yan, J. Wang, X. Li, and L. Guo, “Architectural Design and Evaluation of an Efficient Web-Crawling System,” *J. Systems and Software*, vol. 60, no. 3, pp. 185–193, Feb. 2002.

[143] L. Yang and M. Michailidis, “Sample Based Estimation of Network Traffic Flow Characteristics,” in *Proc. IEEE INFOCOM*, May 2007, pp. 1775–1783.

[144] X. Yang and J. Sun, “An Analytical Performance Model of MapReduce,” in *Proc. IEEE CCIS*, Sep. 2011, pp. 306–310.

[145] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, “Improving MapReduce Performance in Heterogeneous Environments,” in *Proc. USENIX OSDI*, Jun. 2008, pp. 29–42.

[146] Y. Zhou, J. Philbin, and K. Li, “The Multi-Queue Replacement Algorithm for Second Level Buffer Caches,” in *Proc. USENIX Annual Technical Conference*, Jun. 2001, pp. 91–104.

[147] J. J. H. Zhu, T. Meng, Z. Xie, G. Li, and X. Li, “A Teapot Graph and Its Hierarchical Structure of the Chinese Web,” in *Proc. WWW Poster Session*, Apr. 2008, pp. 1133–1134.