Online Prediction Algorithms for Databases and Operating Systems

P. Krishnan

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-24
August 1995

Online Prediction Algorithms for Databases and Operating Systems[1]

P. Krishnan[2]
Department of Computer Science
Brown University
Providence, R.I. 02912–1910

Abstract

In making online decisions, computer systems are inherently trying to predict future events. Typical decision problems in computer systems translate to three prediction scenarios: predicting *what* event is going to happen in the future, *when* a specific event will take place, or *how much* of something is going to happen. In this thesis, we develop practical algorithms for specific instances of these three prediction scenarios, and prove the goodness of our algorithms via analytical and experimental methods.

We study each of the three prediction scenarios via motivating systems problems. The problem of *prefetching* requires a prediction of which page is going to be next requested by a user. The problem of *disk spindown in mobile machines*, modeled by the *rent-to-buy* framework, requires an estimate of when the next disk access is going to happen. Query optimizers choose a database access strategy by predicting or estimating *selectivity*, i.e., by estimating the size of a query result.

We analyze the novel idea of using data compression techniques for prediction, and investigate the similarities and differences between the different prediction scenarios. We show the theoretical optimality and practical merit of prefetchers based on data compressors, study predictive and adaptive disk spindown strategies, and develop methods for estimating alphanumeric selectivity in relational databases. In each of our prediction algorithms, we cope with the stringent limits on the storage and time available to do prediction that are imposed by system environments. We conclude that good and practical predictors can be obtained from data compressors.

[1] A similar version of this report was submitted in fulfillment of the dissertation requirement for P. Krishnan's Ph.D.

[2]Support was provided in part by an IBM fellowship, by NSF research grant CCR–9007851, by an NSF Presidential Young Investigator Award CCR–9047466 with matching funds from IBM, by the Office of Naval Research and the Advanced Research Projects Agency under contracts N00014–91–J–4052, ARPA order 8225, and contract N00014–83–K–0146, ARPA order 6320, amendment 1, by Army Research Office grants DAAH04–93–G–0076 and DAAL03–91–G–0035, and by the Air Force Office of Scientific Research grants F49620–94–1–0217 and F49620–92–J–051.

Online Prediction Algorithms for Databases and Operating Systems

by
P. Krishnan
B. Tech., Indian Institute of Technology, Delhi, May 1989
Sc. M., Brown University, May 1991

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

May 1995

Copyright © 1995
by
P. Krishnan

Vita

I was born on 22 February, 1968 in Nasik Road, Maharashtra, India. I completed my high school education in 1985 from D.T.E.A. Senior Secondary School in Mandir Marg, New Delhi, India. I then studied Computer Science and Engineering at the Indian Institute of Technology, Delhi, India, where I was awarded the B. Tech degree in 1989. Subsequently, I joined the Ph.D. program in Computer Science at Brown University, where I obtained an Sc. M. degree in 1991. From January 1993 onwards, I have been a visiting research scholar at Duke University in Durham, NC.

I am called "`Krishnan`" or "`P.K.`"

Acknowledgements

I have been fortunate to have Jeff Vitter as my guru through the Ph.D. phase of my life. He introduced me to research, and has always provided excellent technical advice. He has taught me the necessity of clarity in thought and writing, and the importance of precision while conveying ideas. Jeff is demanding, particular about details, and pushes one to achieve one's best and then some. The decision to move with him to Duke University in 1993 to continue this learning process and work on my thesis from there was therefore natural. I would like to thank him for some of the personal advice he has given me over the years; it has proved to be invaluable. Looking back over these past years, I feel a certain sense of pride in that my more recent drafts have come back from Jeff with less amounts of red ink on them. And I knew I had spent sufficient time in graduate school when I was able to decipher Jeff's handwriting without any problem. Meeting his family—his wife Sharon, and their children Jillian, Scott, and Audrey—has always been a lot of fun, and the dinners at their house have been excellent.

I would like to thank Paris Kanellakis and Stan Zdonik, my thesis committee members, for all their comments and help, including the ones related to possible titles for the thesis.

My two summer internships were valuable from both a technical and personal point of view. I had a fun and productive summer at Matsushita Information Technology Laboratories in 1994, and value my interactions with Fred Douglis, Brian Marsh, Rafael Alonso, Ramon Cáceras, Dan Lopresti, and Hank Korth. I am especially glad for the continuing collaborations I have had with Fred. My internship at AT&T Bell Laboratories introduced me to the new area of network management, and I would like to thank Binay Sugla for it. I enjoyed the many discussions with Binay, Venky Krishnaswamy, Tom London, Sandy Thuel and John Zhao.

Jeff's moving from Brown to take the position of Chair of the Department of Computer Science at Duke University in 1993, and my accepting his invitation to move to Duke and work on my thesis from there has given me the opportunity to interact with people at two excellent institutions of higher learning in the United States. I would like to thank the Computer Science Departments and the Graduate Schools at both institutions for allowing me the use of their facilities with ease, and their prompt advice on the many administrative issues; in particular, I would like to thank Mary Andrade, Katrina Avery, Dorinda Moulton, and Dawn Nicholaus at Brown, and Cathie Caimano, Burdette Connell, Anna Drozdowski, and Tina Gaither at Duke. John Eng-Wong and Marilyn at the Foreign Student Office at Brown have helped me much with my visa-related questions. I would also like to thank Andy van Dam and the graphics group at Brown, and Dr. Henry Fuchs and the graphics group at the University of North Carolina at Chapel Hill for making it possible for me to give my thesis proposal presentation remotely over the televideo network.

My many friends and collaborators, both graduate students and faculty, have helped

make work interesting and life fun: Swarup Acharya, Ajit Agrawal, Ken Curewitz, Gerd Hillebrand, Paul Howard, Dave Langworthy, Alberto Marquez, Mark Palmer, Andrea Pietracaprina, Sridhar Ramaswamy, Ramamurthy Ravi, Peter Revesz, Sairam Subramanian, Roberto Tamassia, and Matthias Wloka at Brown; Pankaj Agarwal, Thomas Alexander, Lars Arge, Subro Bhattacharya, Eddie Grove, Dzung Hoang, Ming Kao, Phil Long, T. M. Murali, and Darren Vengroff at Duke. Thanks to Murali (T_{MAX}) for many valuable comments on an earlier draft of the thesis, and to Matthias for all his help in many matters stemming from my not being physically present in Providence for much of the last two years. Thanks to all my friends with whom I have spoken in a language not involving bits and bytes for helping me keep life in perspective.

I would like to express my deep regards and thanks to my parents; their advice and encouragement through the years has been invaluable, and it continually amazes me to rediscover some of their practical wisdom.

Credits

Portions of the work described in this thesis were done in collaboration with other researchers. In particular, the material in Chapter 3 is joint work with my thesis advisor, Jeff Vitter, and a version of the chapter appears as [ViK]. The material in Chapter 4 is joint work with Jeff Vitter, and a version of the chapter appears as [KrV]. The material in Chapter 5 is joint work with Ken Curewitz and Jeff Vitter, and a version of the chapter appears as [CKV]. The material in Chapter 7 is joint work with Fred Douglis and Brian Marsh, and was performed at Matsushita Information Technology Laboratories (MITL); a version of the chapter appears as [DKM]. The material in Chapters 8 and 9 is joint work with Phil Long and Jeff Vitter, and a version of these chapters appears as [KLV]. The material in Chapter 11 is joint work with Jeff Vitter. Extensions to the work presented in Chapter 11 are ongoing with Bala Iyer at IBM, Santa Teresa. I am grateful to my co-authors for permission to use parts of our joint work in the present thesis.

I would like to thank Mark Palmer and Digital Equipment Corporation for providing their database access traces (used in the experiments reported in Chapter 5). I would like to thank John Wilkes and the Hewlett-Packard Company for making their file system traces (used in the experiments reported in Chapters 7 and 9) available to us. Thanks also to Bill Sproule at MITL who collected the Powerbook traces (used in the experiments reported in Chapter 7), and to MITL for allowing me to use the results of some of the research (reported in Chapter 7) performed when I was there. Thanks to Bala Iyer at IBM, Santa Teresa for introducing us to the problem of predicting alphanumeric selectivity, providing the *dbgen* software (used in the experiments reported in Chapter 11), and for valuable practical advice and many interesting discussions related to the problem of predicting alphanumeric selectivity in relational databases.

The research presented in this thesis was partly supported by an IBM fellowship, by NSF research grant CCR–9007851, by an NSF Presidential Young Investigator Award CCR–9047466 with matching funds from IBM, by the Office of Naval Research and the Advanced Research Projects Agency under contracts N00014–91–J–4052, ARPA order 8225, and contract N00014–83–K–0146, ARPA order 6320, amendment 1, by Army Research Office grants DAAH04–93–G–0076 and DAAL03–91–G–0035, and by the Air Force Office of Scientific Research grants F49620–94–1–0217 and F49620–92–J–051. I gratefully acknowledge the support.

UNIX is a registered trademark of UNIX Systems Laboratories, Inc. DEC Object/DB is a registered trademark of Digital Equipment Corporation. Powerbook is a registered trademark of the Apple Computer Corporation. Kittyhawk and HP-UX are registered trademarks of Hewlett-Packard Company. Word is a registered trademark of Microsoft Corporation. Eudora is a registered trademark of Qualcomm. Go•Drive is a registered trademark of Quantum.

Contents

1 Introduction **1**
1.1 Prefetching . . . 1
1.2 Disk Spindown via Rent-to-Buy . . . 2
1.3 Selectivity . . . 3
1.4 Our Thesis . . . 3

2 Prefetching: Predicting the *What* **5**
2.1 Background and Related Work . . . 6
2.2 Our Approach . . . 6

3 Optimal Prefetching via Data Compression **8**
3.1 Page Request Models and Main Results . . . 9
3.2 A Prefetching Algorithm Based on Ziv-Lempel . . . 10
3.3 Analysis of our Prefetching Algorithm . . . 12
3.3.1 Bounds on Compression . . . 12
3.3.2 Bounds on Fault Rate . . . 17
3.4 The mth order Markov Source Model . . . 22
3.5 Discussion . . . 22

4 Optimal Prefetching in the Worst Case **24**
4.1 Analysis Model and Main Results . . . 25
4.2 The Prefetching Algorithm P_1 . . . 26
4.3 One-State Case: Optimality of P_1 vs. $\mathcal{F}(1)$ . . . 28
4.3.1 The Approximately Balanced Sequence is Sufficiently Worst-Case . . . 29
4.3.2 $Fault_{P_1}(\widetilde{\sigma}_1^n)$ is Close to $Fault_{\mathcal{F}(1)}(\sigma_1^n)$ . . . 32
4.4 Generalizing P_1 to Get P . . . 34
4.4.1 Proof of Theorem 4.2 . . . 36
4.5 Constant-Time Prediction . . . 37
4.6 Optimality of P_1' for $\alpha \geq 2,\ k = 1$ . . . 38
4.7 Discussion . . . 40

5 Practical Prefetching via Data Compression **41**
5.1 System Environment . . . 42
5.2 Algorithms for Prefetching . . . 43
5.2.1 Algorithm LZ . . . 43
5.2.2 Algorithm PPM . . . 44
5.2.3 Algorithm FOM . . . 44

5.2.4 Cache Replacement Issues . . . 45
5.3 Restricted Memory Environment . . . 45
5.3.1 Paging the Data Structure . . . 46
5.3.2 Sequence of Fast Page Requests . . . 46
5.4 Simulation Environment . . . 47
5.4.1 Simplifying Assumptions . . . 47
5.4.2 Simulation Method . . . 48
5.5 Experimental Results . . . 49
5.5.1 Description of the Traces . . . 49
5.5.2 Prefetch Results for Uniform Prefetching . . . 49
5.5.3 Prefetch Results with Fast Page Requests . . . 52
5.5.4 Analyzing the Results . . . 52
5.6 Discussion . . . 54

6 Disk Spindown and Rent-to-Buy: Predicting the *When* 57
6.1 The Rent-to-Buy Framework . . . 58
6.2 Background and Related Work . . . 58
6.3 Our Approach . . . 59

7 Fixed Threshold and Predictive Disk Spindown Policies 60
7.1 Disk Energy States . . . 61
7.2 Policies . . . 61
7.2.1 Offline Policies . . . 61
7.2.2 Threshold-based Policies . . . 63
7.2.3 Predictive Policies . . . 63
7.2.4 Taxonomy . . . 63
7.3 Methodology . . . 64
7.3.1 Traces . . . 64
7.3.2 Simulator . . . 65
7.4 Results . . . 68
7.4.1 Offline Algorithms . . . 69
7.4.2 Threshold-Demand Algorithms . . . 69
7.4.3 The Impact of Disk Characteristics . . . 69
7.5 Predictive Algorithms . . . 70
7.5.1 Experiences with Using PREDICTIVE_DEMAND . . . 72
7.6 Discussion . . . 72

8 Optimal Rent-to-Buy in Probabilistic Environments 74
8.1 Online Rent-to-Buy . . . 74
8.2 Definitions and Main Analytical Results . . . 76
8.3 The Main Idea: Algorithm A_ϵ . . . 78
8.3.1 First Stage . . . 78
8.3.2 Second stage . . . 79
8.4 Goodness of Algorithm A_ϵ . . . 80
8.4.1 Guarantees about the First Stage . . . 80
8.4.2 Convergence of Algorithm A_ϵ . . . 81
8.4.3 Computation Time of Algorithm A_ϵ . . . 84

8.5 Getting Algorithms L and L_s from Algorithm A_ϵ . . . 85
8.5.1 Algorithm L . . . 85
8.5.2 Algorithm L_s . . . 86
8.6 Discussion . . . 87

9 Adaptive Disk Spindown via Rent-to-Buy 88
9.1 Adaptive Disk Spindown . . . 89
9.2 Methodology . . . 89
9.3 Results . . . 90
9.3.1 Effective Cost vs. a . . . 90
9.3.2 Excess Energy vs. a . . . 93
9.3.3 Operations Delayed vs. a . . . 93
9.3.4 Adaptability and Rent-to-Buy . . . 93
9.3.5 Other Observations . . . 93
9.4 Discussion . . . 97

10 Alphanumeric Selectivity: Predicting the *How Much* 98
10.1 Background and Related Work . . . 99
10.2 Our Approach . . . 100

11 Alphanumeric Selectivity in the Presence of Wildcards 101
11.1 The Suffix Tree . . . 102
11.2 The Preprocessing Phase . . . 104
11.2.1 Stage 1: Building the Suffix Tree T . . . 104
11.2.2 Stage 2: Building the Catalog C . . . 106
11.3 The Online Phase . . . 109
11.3.1 Independence-based Strategies, I_* . . . 110
11.3.2 Child Estimation-based Strategies CE_* . . . 112
11.3.3 Depth Estimation-based Strategies, DE_* . . . 114
11.4 Methodology for the Experiments . . . 115
11.5 Results for the Preprocessing Phase . . . 116
11.5.1 Size of the Tree in Stage 1 . . . 117
11.5.2 Statistics Dealing With Catalog Building . . . 117
11.5.3 Catalog Size . . . 118
11.5.4 Size of the Catalog versus Prune Count . . . 118
11.6 Results for the Online Phase . . . 119
11.6.1 Positive Single Patterns . . . 120
11.6.2 Positive Double Patterns . . . 122
11.6.3 Negative Patterns . . . 123
11.6.4 Comparing the Strategies Globally . . . 124
11.7 A "Strawman" Random Sampling Scheme . . . 125
11.8 Extensions . . . 126
11.8.1 Non-Unit Patterns . . . 126
11.8.2 Optimization With Extra Knowledge . . . 126
11.9 Discussion . . . 127

12 Conclusions 128

List of Tables

5.1 Trace characteristics and fault rate for LRU and OPT 50
5.2 Uniform prefetching fault rates 50
5.3 Uniform prefetching memory use 52
5.4 Data structure page I/Os 52
5.5 Effect of cache size 54

7.1 Disk characteristics 62
7.2 Trace characteristics 65
7.3 Bucket definition 70
7.4 A sample transition table 71

11.1 Depth vs. count distribution 115
11.2 Grades for the different algorithms 125
11.3 Grades for the "strawman" random sampling scheme 125

List of Figures

3.1 The parse tree constructed by encoder $\mathcal{E}$ 11

4.1 Snapshot of the data structure for Algorithm P 35

5.1 The client-server model 43
5.2 Snapshot of the data structure for PPM of order 2 45
5.3 Updating the LZ data structure 47
5.4 Uniform prefetching for the trace CAD1 51
5.5 Uniform prefetching for the trace OO7_T1 51
5.6 Fast page request prefetching for the trace OO7_T1 53

7.1 Disk energy states 61
7.2 Simulation results using the Powerbook trace 66
7.3 Simulation results using the HP-UX trace 67

8.1 Snapshot of the data structure used by algorithm A_ϵ 84

9.1 Effective cost vs. a for the Kittyhawk disk 91
9.2 Effective cost vs. a for the Go•Drive disk 92
9.3 Excess energy vs. a for the Kittyhawk disk 94
9.4 Operations delayed vs. a 95
9.5 Excess energy vs. operations delayed 96
9.6 Reads delayed vs. a 97

11.1 Suffix tree and its creation 102
11.2 An example suffix tree 105
11.3 Deriving the catalog from the table 107
11.4 An example catalog 111
11.5 Statistics from the preprocessing phase 117
11.6 Prune count vs. number of nodes in the catalog 119
11.7 Independence-based strategies for positive single patterns 120
11.8 Child Estimation-based strategies for positive single patterns 121
11.9 Depth Estimation-based strategies for positive single patterns 121
11.10 Best performing strategies for positive single patterns 122
11.11 Best performing strategies for positive double patterns 122
11.12 Best performing strategies for negative patterns 123

Chapter 1

Introduction

We should all be concerned about the future because
we will have to spend the rest of our lives there.

—Charles F. Kettering (1949)

Computer systems often deal with streams of events and make decisions with each event that affect system performance. These decisions are made *online*: an online algorithm only uses past knowledge to make its decisions. To make optimal decisions, a system often needs to estimate or predict the future. Typical scenarios involving such decisions require a prediction of *what* is going to happen, or *when* a specific event will take place, or *how much* of something is going to happen.

In this thesis, we investigate important decision problems in typical systems scenarios and develop techniques for predicting the future that aid in optimal decision making. We use three motivating systems problems that exemplify the three prediction scenarios mentioned. In each scenario, predictions are used to make decisions that improve the response-time performance of the system. Apart from being accurate, online predictors for computer systems (e.g., databases and operating systems) must be efficient in the time and space they use to make their predictions.

1.1 Prefetching

The question of predicting what is going to happen in the future is best illustrated by the problem of *prefetching*. Most computer memories are organized as a hierarchy. A typical two-level memory consists of a relatively small but fast *cache* (such as internal memory) and a relatively large but slow memory (such as disk storage). Two-level memories can also model on-chip versus off-chip memory in VLSI systems, or a client-server model of computation. The pages requested by an application must be in cache before computation can proceed. In the event that a requested page is not in cache, a *page fault* occurs and the application has to wait while the page is fetched from slow memory to cache. This method of fetching pages into cache only when a fault occurs is called *demand fetching*. The problem of *cache replacement* or *caching* is to decide which pages to remove from cache to accommodate the incoming pages.

In many systems and database applications, users spend a significant amount of time processing a page, and the computer and the I/O system are typically idle during that

period. If the computer can predict which page the user will request next, it can fetch that page into cache (if it is not already in cache) *before* the user asks for it. When the user requests the page, it is available in cache, and the user perceives a faster response time. This method of getting pages into cache in the background before they are requested is called *prefetching*. The primary action in prefetching is predicting which page will be next requested by the user.

1.2 Disk Spindown via Rent-to-Buy

The problem of predicting when an event is going to happen in the future is well illustrated by the problem of *rent-to-buy*. The *single rent-to-buy decision* problem is described as follows: we need a resource for an unknown amount of time, and we have the option to rent it for \$1 per unit time, or to buy it once and for all for \$$c$. For how long do we rent the resource before buying it? Clearly, we should buy the resource if we expect to use it for c more time units. With full knowledge of the future, we will buy the resource immediately if the resource will be used for more than c time units, otherwise, we will rent the resource. Deciding when to buy is closely related to estimating for how long a resource will be used. Many interesting systems problems can be modeled well by a *sequence* of single rent-to-buy problems. To solve the tth single rent-to-buy problem (or the tth *round*), the online algorithm can use its experience from the previous $t-1$ rounds.

A real-life systems scenario that motivates us to study the rent-to-buy framework is the *disk spindown problem*. Energy conservation is an important issue in mobile computing. Portable computers run on battery power and can function for only a few hours before draining their batteries. Current techniques for conserving power are based on shutting down components of the system after reasonably long periods of inactivity. Recent studies show that the disk sub-system on notebook computers is a major consumer of power [DKM, LKH, MDK]. Most disks used for portable computers (e.g., the small, light-weight Kitty-hawk from Hewlett Packard [Pac]) have multiple energy states. Conceptually, the disk can be thought of as having two states: the *spinning* state in which the disk can access data but consumes a lot of power and a *spundown* state in which the disk consumes effectively no power but cannot access data. Spinning down a disk and spinning it up consumes a fixed amount of energy and time (and also produces wear and tear on the disk). During periods of inactivity, the disk can be spundown to conserve energy at the expense of increased latency for the next request. The *disk spindown problem* is to decide when to spindown the disk so as to conserve energy, while keeping latency acceptable.

The disk spindown problem can be modeled as a rent-to-buy problem as follows. A round is the time between any two requests for data on the disk. For each round, we need to solve the disk spindown problem. Keeping the disk spinning is viewed as renting, since energy is continuously expended to keep the disk spinning. Spinning down the disk is viewed as a buy, since the energy to spindown the disk and spin it back up upon the next request is independent of the remaining amount of time until the next disk access. The cost of the increased latency in serving the next disk access is also an important consideration in deciding when to spindown, and can be merged into the buy cost c. If we can predict in each round when the next access at disk is going to happen, we can make an optimal spindown decision.

1.3 Selectivity

The question of predicting how much is going to happen in the future is well illustrated by the problem of *predicting selectivity in relational databases*. Query optimization is an important part of database management systems [Dat, Ull]. The size of databases is growing rapidly and decisions support is an important component of databases. A typical query to a database is to return all records of the database satisfying predicate P. To answer such queries, the database performs query optimization to develop a search strategy. It is critical to be able to predict *selectivity*, i.e., the fraction of rows in the database that satisfy the predicate. The prediction is used by the query optimizer, for example, to determine whether to use a file scan to access all rows of the table, evaluate the predicate against each row, and return qualifying rows to the user, or to use indices (if present) to access only those rows that qualify. Index access costs and file scan costs are quite different. Query optimizers have cost models that estimate the access cost as a function of the predicted number of qualifying rows, and thus determine the cheaper alternative. Finding a good search strategy depends on the query optimizer's ability to predict or estimate the percentage of rows that satisfy the predicate.

A particularly difficult prediction scenario involves estimating non-numeric selectivity in relational databases in the presence of wildcards [Iye]. The problem here is to preprocess the relational database (during off-hours, e.g., during the weekend) and come up with a compact structure that will allow online estimation of non-numeric queries involving wildcards. For example, given a relational database specifying the inventory of a department store, how many rows of the database have the color column entry matching the pattern "$*$`green`$*$", where "$*$" represents a wildcard, and matches any string of 0 or more characters. The main difficulty in this scenario is that the space and time available to the estimator are often particularly stringent.

1.4 Our Thesis

We have observed three important problems that are typical in systems applications. If we have good predictors for these types of problems, we can significantly improve the response-time performance of systems. But there is more to making the predictions than just outputting what is going to happen in the future. While it is non-trivial to develop good predictors, it is imperative that such predictors be efficient in the time required to make the predictions and the space used in making the predictions, since the use of these predictors is made online in systems with strict limits on available resources.

Our thesis is that space- and time-efficient predictors can be developed for such typical prediction problems via techniques based on data compressors. Although we use the specific problems described above to develop our predictors, the prediction techniques we use are not limited to these specific problems, but are applicable across the gamut of computer systems. Our work on these problems as described in the following chapters suggests a very intriguing fact: Although the three scenarios seem different, there is an intuitive similarity between good prediction techniques for these scenarios. We have proposed and investigated the novel approach of using data compression techniques for certain types of prediction (especially the "what" type), and studied the generalizability of this approach towards developing efficient prediction algorithms for the other prediction scenarios. We show how to incorporate the restrictions imposed by the system environment to develop practical predictors.

In Chapters 2–5, we study the problem of prefetching in detail, and describe our novel approach of using data compression techniques for prefetching. We study the disk spindown problem and the rent-to-buy framework in Chapters 6–9. In Chapters 10 and 11, we study the problem of predicting selectivity of specific types of queries towards query optimization. We conclude in Chapter 12 with a synopsis of the similarities and differences between the different prediction scenarios, and the important intuition gained from our study.

Chapter 2

Prefetching: Predicting the *What*

Who's on first, What's on second, ...
—Bud Abbott and Lou Costello (1945)

Most computer memories are hierarchical. A typical two-level memory consists of a relatively small but fast *cache* (such as internal memory) and a relatively large but slow memory (such as disk storage). A two-level memory also models the *client-server* paradigm of computing in which the client is the database user (or application) and the server manages the database; we can think of the client workstation as a cache, and the server as slow memory. The pages requested by an application must be in cache before computation can proceed. In the event that a requested page is not in cache, a *page fault* occurs and the application has to wait while the page is fetched from slow memory to cache. The method of fetching pages into cache only when a page fault occurs is called *demand fetching*. The problem of *cache replacement* or *caching* is to decide which pages to remove from cache to accommodate the incoming pages.

In many systems and database applications, the user spends a significant amount of time processing a page, and the computer and the I/O system are typically idle during that period. The computer can use this time to predict which page the user will request next, and fetch that page into cache (if it is not already in cache) in the background. When the user requests the page, it is available in cache, and the user perceives a faster response time. *Prefetching* is a method of improving response time performance by anticipating future user requests, and getting the necessary data into cache in the background before an explicit request is made by the application. The primary action in prefetching is predicting which page will be next requested by the user. In this chapter, and Chapters 3–5, we denote the cache size by k, and the database size by α, where $\alpha \gg k$.

The UNIX system uses a simple scheme for prefetching that is optimized for sequential file access; it anticipates and prefetches page $i+1$ when a request is made for page i. Current database systems perform prefetching using such sequential prefetching techniques derived from older virtual memory prefetchers. The I/O bottleneck is seriously impeding performance in large-scale databases, and the demand for improving response time performance is growing [Bra]. The older virtual memory-based prefetchers are inadequate for newer object-oriented and hypertext applications, and this has stimulated renewed interest in developing improved algorithms for prefetching [ChB, Lai, MLG, PaZb, RoL]. Our work [CKV, KrV, ViK] described in Chapters 3–5 develops provably optimal and practical prefetchers by understanding and exploiting the intimate relationship between data compressors and predictors.

2.1 Background and Related Work

At a high level, recent work in prefetching can be divided into three categories: transparent informed prefetching, compiler-based techniques for prefetching, and automatic prediction for prefetching.

In Transparent Informed Prefetching (or TIP) [PGS], the application gives explicit hints for prefetching; i.e., application levels of the system *disclose* future access patterns rather than advise lower-level policies. This information is used for converting the high throughput of new technologies such as disk-arrays and log-structured file systems into low latency for applications. The idea of giving hints has been proposed in many contexts; Trivedi [Tri] suggested using programmer or compiler generated hints for prefetching. Database researchers have studied prefetching based on application level knowledge [Stoa].

Compiler-based techniques for prefetching lie at a lower level of abstraction. Under this paradigm, the compiler reorders instructions in application code and introduces explicit prefetching instructions to reduce the effect of cache misses [MLG]. These schemes have been applied to scientific programs, by having the compiler perform locality analysis to selectively prefetch only those references that are likely to cause cache misses. Hardware schemes of non-blocking and prefetching caches that let processing continue when a cache miss occurs, blocking only when the missed data is actually needed have also been proposed [ChB]. In [RoL], a combined hardware and software approach has been studied where an optimizing compiler and speculative loads are used to issue read requests in anticipation of a demand request.

Interesting approaches have been proposed for prefetching based on predicting future page requests by monitoring past user behavior [Lai, PaZb, Sal]. Palmer and Zdonik [PaZb] use a pattern matching approach to prediction. Based on previous user sessions, they build a model in a training phase; this model is frozen, and in the prediction phase the model is used for prefetching. Salem computes various first-order statistics for prediction [Sal], and Laird uses a growing Markov predictor [Lai]. Song et al. study prefetching based on page fault history [SoC]. Alexander et al. consider hardware prefetching schemes between processor and main memory by extracting tables of fixed length contexts from past user requests [AlK]. Our approach to prefetching as described in Section 2.2 is also based on monitoring past user behavior; we develop a systematic approach to prefetching based on data compression principles.

Prefetching has also been studied in other systems environments. Prefetching in a parallel environment is studied in [KoE]. More recently, the possibility of predictive prefetching in mobile computing has been studied by Kuenning et al. [KPR]. Cao et al. study integrated prefetching and caching strategies [CFK] in file systems assuming full knowledge of future requests; the concern in this case is to decide when to prefetch so as to minimize the total time consumed to satisfy a page request sequence.

2.2 Our Approach

In Chapter 3, we explore the idea of using data compression techniques for prefetching. The intuition is that data compressors typically operate by postulating (either implicitly or explicitly) a dynamic probability distribution on the data to be compressed. Data expected with high probability are encoded with few bits, and unexpected data with many bits. Thus, if a data compressor successfully compresses the data, then its probability distribution on the

data must be realistic and can be used for effective prediction. We formalize this intuition and develop prefetchers that are provably optimal by modeling the user generating the page requests by powerful models such as general Markov sources and mth order Markov sources.

In Chapter 4, we extend our theoretical understanding of the problem by developing an algorithm that is provably optimal for *worst-case* user page requests, when compared against the best finite state prefetcher. This algorithm is randomized and different from the algorithm presented in Chapter 3; however, the underlying structure of the predictor is closely related to the prefetcher from Chapter 3. This strengthens our intuition about the relationship between data compression and prefetching.

For the theoretical study of prefetching discussed in Chapters 3 and 4, we use the model of *pure prefetching*. In pure prefetching, which is valid in many hypertext and iterative database systems, there is sufficient time between user requests to prefetch as many pages as desired limited only by the cache size k. Pure prefetching is an important theoretical and practical model that helps in understanding the benefits of fetching pages in the background, by isolating the prediction component of prefetching from cache replacement. We shall see in subsequent chapters that pure prefetchers can be converted into efficient and practical non-pure prefetchers by melding them with good cache replacement strategies.

In Chapter 5, we investigate the practical issues of doing prefetching via our techniques based on data compressors, and develop strategies to cope with non-pure prefetching, and strict memory and time limitations on prefetching. We investigate prefetchers derived from popular data compressors and show their practical merit via simulations on traces obtained from CAD applications and the OO1 and OO7 benchmarks. We observe significant improvements in the page fault rate with respect to the least recently used cache replacement heuristic.

Our model of prefetching presented in Chapters 3–5 is motivated by hypertext and iterative database applications, and concentrates on prefetching for the next page request. This is different from the compiler-based methods described in Section 2.1 that are oriented towards prefetching much in advance of an anticipated page request. Techniques derived from compiler-based prefetching could potentially be used with our online strategies to perform online prefetching many time steps in advance of an expected page request.

Chapter 3

Optimal Prefetching via Data Compression

As described in Chapter 2, the intuition for using data compressors for prefetching is simple: in order to compress data effectively, we have to be able to predict future data well, and thus good data compressors should be able to predict well for purposes of prefetching. In this chapter, we present our method of using data compression techniques for prefetching, and prove the optimality of our techniques analytically.

The first difficulty in an analytical study of prefetching is developing a technique for proving the goodness of prefetchers. An algorithm is *online* if it must make its decisions based only on past history. An *offline* algorithm can use knowledge of the future. Any implementable algorithm for cache replacement or prefetching must clearly be online. The notion of *competitiveness* introduced by Sleator and Tarjan [SlT] determines the goodness of an online algorithm by comparing its performance to that of offline algorithms. An online cache replacement or prefetching algorithm is *c-competitive* if there exists a constant b such that for any sequence of page requests σ, the number of page faults $NumFaults_X(\sigma)$ that the online algorithm X incurs is at most b more than c times the number of page faults $NumFaults_{opt}(\sigma)$ incurred by an optimal offline algorithm; i.e.,

$$NumFaults_X(\sigma) \leq c \cdot NumFaults_{opt}(\sigma) + b. \tag{3.1}$$

(In general, the *cost* of the algorithm replaces *NumFaults* in (3.1).). Competitive algorithms for cache replacement are well examined in the literature [BIR, FKL, IKP, KPR, McS, SlT].

It is unreasonable to expect algorithms to be competitive in this sense for prefetching. An optimal offline algorithm for prefetching would never fault, if it can prefetch at least one page every time. In order to be competitive, an online algorithm would have to be an almost perfect predictor for any sequence, which seems intuitively impossible.

We resolve this matter by considering powerful probabilistic models. In our main model, the sequence of page requests is assumed to be generated by a labeled deterministic finite state automaton with probabilities on the transitions (a *probabilistic FSA* or *Markov source*). In the second model, we consider the special case of *mth order Markov sources*, in which the states correspond to the contexts of the previous m page requests. We evaluate our prefetching algorithm relative to the best online algorithm that has complete knowledge of the structure and transition probabilities of the Markov source.

Prefetching is a learning problem that involves predicting the page requests of the user. Work in computational learning theory [BEHa, BEHb, BoP] has shown that prediction is

synonymous with generalization and data compression. If a data compressor expects a certain character to be next with a very high probability, it will assign that character a relatively small code. In the end, if the net code length is small, then the predictions of the data compressor must have been good.

In this chapter, we adapt an optimal data compression method to get an optimal pure prefetching algorithm for each of our models. In pure prefetching, as described in Section 2.2, there is sufficient time to prefetch as many pages as wanted, limited only by the cache size k. In general, prefetching requests would be interrupted by the user's actual read requests; we consider non-pure prefetching in Chapter 5.

Our models and results are summarized in Section 3.1. In Section 3.2, for our main Markov source model we apply a character-by-character version of the Ziv-Lempel data compression algorithm [ZiLb] upon which the UNIX *compress* program is based. In Section 3.3, we compare our online algorithm to the best algorithm that has full knowledge of the Markov source. We show that the page fault rate of our algorithm converges for almost all page request sequences to this best algorithm's page fault rate. The trick is to show that good compression results in good prefetching. In the process of showing the fault rate convergence, we extend results in data compression related to the optimality of the Ziv-Lempel data compressor [ZiLb]. In Section 3.4 we show faster convergence to optimality for mth order Markov sources. Other related issues are discussed in Section 3.5.

3.1 Page Request Models and Main Results

In keeping with the analogy between prefetching and text compression, we use the terms "page" and "character" interchangeably. We denote the cache size by k and the total number of different pages (or alphabet size) by α. The logarithm to the base two is denoted by "lg," the natural logarithm by "ln," and the empty string by λ.

Definition 3.1 [Gal] We define a *probabilistic finite state automaton* (probabilistic FSA) as a quintuple (S, A, g, p, z_0), where S is a finite set of states with $|S| = s$, A is a finite alphabet of size α, g is a deterministic "next state" function that maps $S \times A$ into S, p is a "probability assignment function" that maps $S \times A$ into $[0, 1]$ with the restriction that $\sum_{i \in A} p(z, i) = 1$ for all $z \in S$, and $z_0 \in S$ is the start state. A probabilistic FSA when used to generate strings is called a *Markov source*. A Markov source M is *ergodic* if it is irreducible and aperiodic, meaning that each state can reach every other state, and the gcd of the possible recurrence times for each state is 1.

Our main model assumes the source to be a Markov source. For such a source M we introduce the notion of *minimum expected fault rate* F_M, a concept intuitively similar to *entropy* in data compression. More specifically, F_M is the best possible expected fault rate achievable by any online prefetching algorithm, even one with full knowledge of the "next state" function and the transition probabilities of the Markov source M. With a slight abuse of notation, we denote by M the best possible prefetching algorithm (which has full knowledge of the Markov source M). When the source is in state z, the optimal prefetcher M puts into cache the k pages having the k maximum probabilities, which minimizes the probability of page fault during the next page request. This minimum fault probability, weighted by the probability of being in state z, summed over all states z, gives us F_M. This is formalized later in Definition 3.4.

We adapt a character-by-character version of the Ziv-Lempel [ZiLb] data compressor to get our optimal prefetcher $\mathcal{P}$. Theorems 3.1 and 3.2 below are our main results.

Theorem 3.1 *Let M be a Markov source. The expected page fault rate achieved by $\mathcal{P}$ approaches F_M, as the page request sequence length $n \to \infty$. If M is ergodic, we get not only convergence of the mean but also the much stronger convergence almost everywhere: for a finite size dictionary data structure, $\mathcal{P}$'s page fault rate approaches F_M arbitrarily closely for almost all page request sequences σ of length n, as $n \to \infty$.*

We can also show that $\mathcal{P}$ is optimal in the limit, even if we compare $\mathcal{P}$ to the best probabilistic FSA prefetcher tuned individually for each page request sequence σ of length n.

Theorem 3.2 *Let M be an ergodic Markov source. For any page request sequence σ of length n, let F_σ be the best fault rate achievable by a probabilistic FSA with at most s states applied to σ. For a finite size dictionary data structure, $\mathcal{P}$'s page fault rate approaches F_σ arbitrarily closely for almost all page request sequences σ, as $n \to \infty$.*

The convergences in Theorems 1 and 2 also hold when we let the number of states s and the alphabet size α get arbitrarily large, as long as n, s, and α tend to ∞ in that relative order.

An interesting case is when the Markov source is stationary, and the start state is chosen randomly according to steady state probabilities of the states. In this case, since the start state is random, it is unclear how a "best algorithm" M (with full knowledge of the source M) would operate! However, since our prefetcher $\mathcal{P}$ is optimal when M knows the start state (which makes M "stronger"), $\mathcal{P}$ is still optimal even when the start state is random.

Corollary 3.1 *Theorems 3.1 and 3.2 hold even when the Markov source M is stationary.*

Our bound on convergence rate is slow, but the Ziv-Lempel encoder works well in practice for data compression. To get a faster provable convergence of fault rate, we consider a second model, in which the source is assumed to be mth order Markov. In such sources, the probability of the next character is dependent only on the past m characters. We can therefore build a finite state machine for the source, the states labeled by m-contexts and the transitions denoting the unique change from one m-context to the next. The state structure of the source is hence known. In this situation, we develop a simple algorithm $\mathcal{M}$ that collects statistics on the transitions that converge exponentially fast to the actual transition probabilities.

Theorem 3.3 *If the source is an mth order Markov source, for any page request sequence, the page fault rate of $\mathcal{M}$ converges exponentially fast to the fault rate of the source.*

The difficulty of the main model (Theorems 3.1 and 3.2) as opposed to the mth order model (Theorem 3.3) is that the state structure of the source is unknown in the main model and the problem of prefetching is thus significantly harder than simply estimating transition probabilities.

3.2 A Prefetching Algorithm Based on Ziv-Lempel

In this section we develop our prefetching algorithm $\mathcal{P}$ based on a character-based version $\mathcal{E}$ of the Ziv-Lempel algorithm for data compression. The original Ziv-Lempel algorithm [ZiLb]

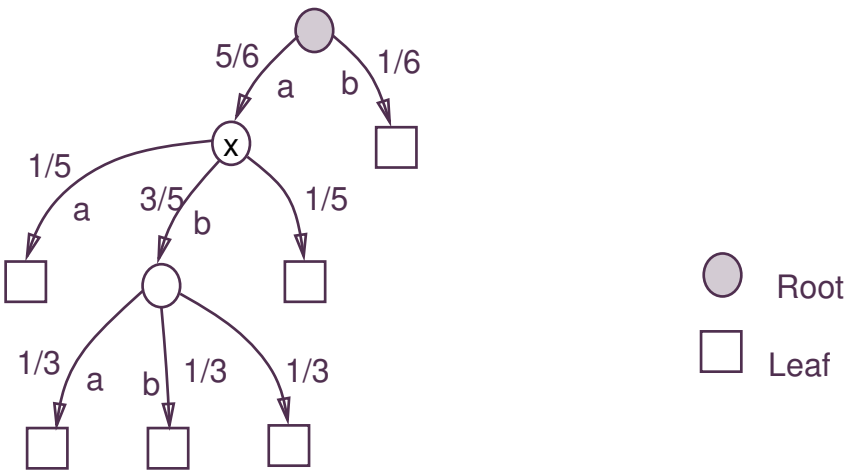


Figure 3.1: The parse tree constructed by the character-based encoder $\mathcal{E}$ for the text "aaaababaabbbabaa", as described in Example 3.1.

is a word-based data compression algorithm. The Ziv-Lempel encoder breaks the input string into blocks of relatively large length n, and it encodes these blocks using a block-to-variable code in the following way: It parses each block of size n into distinct substrings $x_0 = \lambda$, x_1, x_2, ..., x_c such that for all $j \geq 1$ substring x_j without its last character is equal to some x_i, for $0 \leq i < j$. It encodes the substring x_j by the value i, using $\lceil \lg j \rceil$ bits, followed by the ascii encoding of the last character of x_j, using $\lceil \lg \alpha \rceil$ bits.

Arithmetic coding [HoV, Lanb, WNC] is a coding technique that achieves a coding length equal to the entropy of the data model. Sequences of probability p are encoded using $\lg(1/p)$ bits. Arithmetic coding can be thought of as using "fractional" bits, as opposed to the suboptimal Huffman coding in which all code lengths must be integral. The Ziv-Lempel encoder can be converted from a word-based method to a character-based algorithm $\mathcal{E}$ by building a probabilistic model that feeds probability information to an arithmetic coder [BCW, Lana], as explained in the example below. It has been shown that the coding length obtained in this character-based approach is at least as good as that obtained using the word-based approach [BCW, HoV, Lana]. Hence, the optimality results in [ZiLb] hold without change for the character-based approach.

Example 3.1 Assume for simplicity that our alphabet is $\{a, b\}$. We consider the page request sequence "aaaababaabbbabaa...". The Ziv-Lempel encoder parses this string as "(a)(aa)(ab)(aba)(abb)(b)(abaa)...". Each substring in the parse is encoded as a pointer followed by an ascii character. In particular, the match "aba" of the seventh substring "abaa" is encoded using $\lceil \lg 6 \rceil$ bits with a value 4, since the match "aba" is the fourth substring, and the last character "a" is encoded using $\lceil \lg 2 \rceil$ bits, since the alphabet size α is 2.

In the character-based version $\mathcal{E}$ of the Ziv-Lempel encoder, a probabilistic model (or parse tree) is built for each substring when the previous substring ends. The parse tree at the start of the seventh substring is pictured in Figure 3.1. There are five previous substrings beginning with an "a" and one beginning with a "b". The page "a" is therefore

assigned a probability of 5/6 at the root, and "b" is assigned a probability of 1/6 at the root. Similarly, of the 5 substrings that begin with an "a", one begins with an "aa" and three begin with an "ab", accounting for the probabilities of 1/5 for "a" and 3/5 for "b" at node x, and so on. Any sequence that leads from the root of the model to a leaf traverses a sequence of probabilities $p_1, p_2, p_3, \ldots$ whose product $\prod_i p_i$ equals 1/6. The arithmetic coder encodes the sequence with $\sum_i \lg(1/p_i) = \lg(1/\prod_i p_i) = \lg 6$ bits. Note that the unnamed transition with probability 1/5 at node x will never be traversed, and having this arc can only increase the code length (since this probability of 1/5 could otherwise be distributed at node x between "a" and "b" increasing their respective probabilities and reducing their code lengths). Hence, the encoding length using the character-based approach $\mathcal{E}$ will be at least as good as that obtained using the word-based Ziv-Lempel algorithm. □

Our prefetcher $\mathcal{P}$ is based on the character-based version $\mathcal{E}$ of the Ziv-Lempel encoder as follows: At the start of each substring, $\mathcal{P}$'s current node is set to be the root of $\mathcal{E}$'s parse tree. (See Figure 3.1.) Before each page request, $\mathcal{P}$ prefetches the pages with the top k estimated probabilities as specified by the transitions out of its current node. On seeing the actual page requested, $\mathcal{P}$ resets its current node by walking down the transition labeled by that page and gets ready to prefetch again. In addition, if the page is not in memory, a page fault is generated. When $\mathcal{P}$ reaches a leaf, it fetches in k pages at random. The next page request ends the substring, and $\mathcal{P}$ resets its current node to be the root. At the end of n page requests, for some appropriately large n, $\mathcal{P}$ throws away its model and starts afresh.

Notice that instead of storing the probability associated with a transition, we can store the count corresponding to the number of times the transition was traversed. Updating the model, which corresponds to incrementing the count of each traversed transition by one, can be done dynamically while $\mathcal{P}$ traverses it.

3.3 Analysis of our Prefetching Algorithm

Our analysis of the fault rate achieved by our prefetcher $\mathcal{P}$ builds on an analysis of the compression achieved by the Ziv-Lempel character-based encoder $\mathcal{E}$ that $\mathcal{P}$ is based on. In Section 3.3.1 we show that $\mathcal{E}$ is optimal in terms of compression for almost all strings emitted by a Markov source. In Section 3.3.2, we build on Section 3.3.1 and show $\mathcal{P}$ is optimal in terms of prefetching for almost all strings emitted by a Markov source.

3.3.1 Bounds on Compression

We let σ denote a (possibly infinite) sequence from the alphabet A, and we use the notation σ_i^j to denote the subsequence of σ starting at the ith character up to and including the jth character; in particular σ_1^n denotes the first n characters of σ. For convenience, we use $p_{z,i}$ to denote $p(z,i)$, $g(z,\sigma_1^n)$ to denote the state reached by a probabilistic FSA when processing string σ_1^n starting in state z, and $\mathbf{Pr}(z,\ell)$ to denote the probability that the source M is in state z after emitting ℓ characters.

Definition 3.2 [Gal] Let M be a Markov source. The minimum average encoding length per character of M for input sequences of length n is given by

$$H_M(n) = \frac{1}{n} \sum_{z \in S} \left(\sum_{\ell=0}^{n-1} \mathbf{Pr}(z,\ell) \right) \left(\sum_i p_{z,i} \lg \frac{1}{p_{z,i}} \right).$$

If we take the limit of $H_M(n)$ as $n \to \infty$, we get the *entropy* H_M of M.

We now examine the performance of the Ziv-Lempel character-based encoder $\mathcal{E}$ under our probabilistic model of sources and coders. Note that an arithmetic coder can use a probabilistic FSA as a model to perform data compression, and hence probabilistic FSAs can be considered as encoders.

Definition 3.3 Given an encoder C, we define C's *compression* (or number of output bits per character) of σ_1^n by

$$Compression_{C,n}(\sigma_1^n) = \frac{L(y_1^n)}{n}, \tag{3.2}$$

where $L(y_1^n)$ is the length of C's encoding of σ_1^n. Let $M(s)$ be the set of all probabilistic FSAs with $|A| = \alpha$ and $|S| \leq s$. We denote $\min_{C \in M(s)}\{Compression_{C,n}(\sigma_1^n)\}$ by $Compression_{M(s),n}(\sigma_1^n)$.

In particular, if we use a Markov source M to generate the sequence σ_1^n and also to encode σ_1^n (via arithmetic coding), the average compression achieved is equal to the entropy of M; that is,

$$E(Compression_{M,n}) = H_M(n). \tag{3.3}$$

The compression definitions in Definition 3.3 above are similar to those of Ziv and Lempel [ZiLb], except that they define $M(s)$ to be a class of "information lossless" non-probabilistic FSA encoders, use ρ in place of *Compression*, and use $n \lg \alpha$ in place of n in (3.2) to get a ratio of output length to input length.

We generalize Ziv and Lempel's main result [ZiLb] to our model $M(s)$ of probabilistic FSAs, using an iterative analysis based on arithmetic coding, to get the following theorem:

Theorem 3.4 *The compression of $\mathcal{E}$ on σ_1^n is no worse than the best probabilistic FSA in the limit as $n \to \infty$. In particular,*

$$Compression_{\mathcal{E},n}(\sigma_1^n) \leq Compression_{M(s),n}(\sigma_1^n) + \delta_s(n), \qquad \lim_{n\to\infty} \delta_s(n) = 0.$$

To prove Theorem 3.4, we first prove some facts about arithmetic coding and the way the character-based encoder $\mathcal{E}$ works. The first step is to lowerbound $Compression_{M(s),n}(\sigma_1^n)$, for which we need the following lemmas.

Lemma 3.1 *Suppose that σ_1^n can be parsed into c substrings, such that no substring is a prefix of another. We have*

$$Compression_{M(s),n}(\sigma_1^n) \geq \frac{c}{n} \lg \frac{c}{s^2},$$

Proof: The working of an arithmetic coder using a probabilistic FSA as its model can be explained in the following way: The arithmetic coder associates a unit interval $[0, 1]$ with each state of the model. This interval is partitioned into distinct subintervals, one per character, the size of the subinterval associated with a character proportional to the probability of its transition out of the state. Any string that takes the coder from state x to state x' of the model defines implicitly a subinterval of the original $[0, 1]$ interval at x; also, this subinterval uniquely characterizes the string. The size of this subinterval is clearly the product of the probabilities of the transitions taken by the arithmetic coder while processing the string and all the arithmetic coder's output has to do is to identify this subinterval. To

identify a subinterval of length u, an arithmetic coder has to output at least $\lg(1/u)$ bits; for more details refer [HoV, Lanb, WNC]. As an example, consider the probabilistic FSA of Figure 3.1 being used as a model by an arithmetic coder. Starting at node x, the string "`ba`" would cause the interval to shrink by a factor of $3/5$ from $[0,1]$ to $[0.2, 0.8]$ and then by a factor of $1/3$ from $[0.2, 0.8]$ to $[0.2, 0.4]$. To identify "`ba`", the arithmetic coder needs to output at least $-\lg(0.4 - 0.2)$ bits (which is the same as $\lg((3/5)(1/3))$).

It is clear that if there are c' distinct substrings processed in which M starts from state x and ends in state x', these c' substrings define c' distinct subintervals of the original $[0,1]$ interval at x. If these c' substrings are such that no one is a prefix of another, the subintervals corresponding to them are non-overlapping, and the sum of the lengths of these subintervals is at most 1. By convexity arguments, for these c' substrings, the arithmetic coder has to output an average of at least $\lg c'$ bits per substring, giving a total output length of at least $c' \lg c'$ bits for these substrings.

To calculate the output length of M on σ_1^n, we can trace σ_1^n through M and sum up the output lengths for each substring in the parsing of σ_1^n. If σ_1^n can be parsed into c distinct substrings, no one being a prefix of another, by convexity arguments the code length is minimized when the c substrings are distributed equally over the s^2 state pairs (x, x'), where x is the state of M when the substring is about to be processed and x' is the state of M after the substring is processed. Substituting $c' = c/s^2$ gives us the desired bound. □

When the Ziv-Lempel encoder parses a string σ_1^n, it breaks up this string into distinct substrings that are closed under the prefix operation; that is, if σ' is one of the substrings in the parse, then every prefix of σ' is also one of the substrings in the parse. The substrings in the parse can be denoted by a parse tree, like the one pictured in Figure 3.1. The nodes of the parse tree correspond to the substrings in the parse and node i is a child of node j via the edge labeled "`a`" if substring j appended with character "`a`" gives substring i. For two nodes in a parse tree, if neither is an ancestor of the other, then neither can be a prefix of the other; in particular, the set of leaves of a parse tree is a maximal set of such "prefix-closed" substrings.

The internal path length of a tree is defined as the sum over all nodes of the length of the path from the root to that node. For a parse tree, the internal path length is the length of the original string. The branch factor of a tree is the maximum number of children that any node of the tree can have. For a parse tree, the branch factor is the alphabet size α. We now show a lower bound on the number of leaves in a general tree.

Lemma 3.2 *Any tree with c nodes and internal path length n has at least $c^2/20n$ leaves.*

Proof: Let the branch factor of the tree be denoted by α. The case $\alpha = 1$ is trivial, so let us assume that $\alpha \geq 2$. We call the number v *feasible* if there exists a tree with v leaves that has at least c nodes and internal path length at most n. For any feasible v, if we create a tree T with at least c nodes and v leaves and with minimum internal path length, then T's internal path length must be at most n. This tree T must accommodate as many nodes as possible close to its root so as to minimize its internal path length. Hence T must consist of a "full tree part" T' sitting above a "strand part" S'.

The full tree part T' has c' nodes and v leaves, where every internal node, except possibly one, has α children. It follows that T' has at most $c' \leq v\alpha/(\alpha - 1) \leq 2v$ nodes. The strand part S' has the remaining $c - c'$ nodes distributed as v strands, each strand of length u or

$u+1$ and hanging from a leaf of T'. The number c of nodes in T is the sum of the number of nodes in T' and S', which is bounded by $2v + v(u+1) \leq vu + 3v$. Hence, we have

$$vu + 3v \geq c. \tag{3.4}$$

The contribution of S' to the internal path length of T is at least $vu^2/2$, so we have $vu^2/2 \leq n$, which implies that $vu \leq \sqrt{2nv}$. Since $v \leq n$, we have $3v \leq 3\sqrt{nv}$. Substituting these bounds into (3.4), we get $\sqrt{nv}(\sqrt{2}+3) \geq c$, from which it follows that $v \geq c^2/20n$. □

We now use Lemmas 3.1 and 3.2 to get a lower bound on the compression of σ_1^n by any finite state encoder.

Lemma 3.3 *For any string σ_1^n, we have*

$$\begin{aligned} \mathit{Compression}_{M(s),n}(\sigma_1^n) \;\geq\; & \frac{1}{n}\Big(2c(\sigma_1^n)\lg c(\sigma_1^n) - c(\sigma_1^n)\lg n \\ & - c(\sigma_1^n)\lg s^2 - c(\sigma_1^n)\lg\left(20(2\alpha+2)^4\right)\Big), \end{aligned}$$

where $c(\sigma_1^n)$ is the maximum number of nodes in any parse tree for σ_1^n.

Proof: Consider a parse of σ_1^n into c distinct prefix-closed substrings. The corresponding parse tree for σ_1^n has c nodes. Recall that the v leaves of this tree (which are substrings of σ_1^n) are such that no one substring is a prefix of another, and by Lemma 3.1, any encoder requires at least $v\lg(v/s^2)$ bits to encode these leaves (substrings). We can strip off this layer of leaves from the tree and recursively lowerbound the encoding length for the remaining nodes of the tree.

To analyze this recursive process, we consider the stripping procedure to work in phases. Each phase involves the stripping of one or more complete layers of leaves. For $1 \leq i \leq r$, the ith phase ends when the number of nodes remaining in the tree is $c_i \leq c_{i-1}/2$. By definition, we have $c_0 = c$ and $c_r = 0$. Let the number of leaves at the end of the ith phase be v_i. If we denote by $v_{i,j}$ the number of leaves removed in the jth layer of stripping within phase i, then by Lemma 3.1 the encoding length by any finite state encoder of the nodes stripped off in the ith phase is at least

$$\sum_j v_{i,j}\lg\frac{v_{i,j}}{s^2} \geq \sum_j v_{i,j}\lg\frac{v_i}{s^2} = (c_{i-1}-c_i)\lg\frac{v_i}{s^2}.$$

By Lemma 3.2, we have $v_i \geq c_i^2/20n$. Hence,

$$\begin{aligned} \mathit{Compression}_{M(s),n}(\sigma_1^n) \;&\geq\; \frac{1}{n}\sum_{i=1}^{r}(c_{i-1}-c_i)\lg\frac{v_i}{s^2} \\ &\geq\; \frac{1}{n}\sum_{i=1}^{r}(c_{i-1}-c_i)\lg c_i^2 - \frac{\lg 20ns^2}{n}\sum_{i=1}^{r}(c_{i-1}-c_i). \end{aligned} \tag{3.5}$$

By the definition of when a phase ends, we have $c_{i-1} - c_i \geq c_{i-1}/2$. Since the branch factor of the parse tree is the alphabet size α, we get the upper bound $c_{i-1} - c_i \leq c_{i-1}/2 + c_i\alpha$. This gives us

$$c_i \geq \frac{c_0}{(2\alpha+2)^i}.$$

Hence, we have

$$\sum_{i=1}^{r}(c_{i-1}-c_i)\lg c_i^2 \geq 2\sum_{i=1}^{r}(c_{i-1}-c_i)\lg c_0 - 2\lg(2\alpha+2)\sum_{i=1}^{r} i(c_{i-1}-c_i). \tag{3.6}$$

Applying simple telescoping to (3.6) and substituting the result in (3.5) we get

$$\mathit{Compression}_{M(s),n}(\sigma_1^n) \geq \frac{1}{n}\left(2c\lg c - c\lg n - c\lg s^2 - c\lg\left(20(2\alpha+2)^4\right)\right). \tag{3.7}$$

Since (3.7) is true for any c, it is true when $c = c(\sigma_1^n)$, the maximum number of nodes in any parse tree for σ_1^n. □

We are now ready to present the proof of Theorem 3.4.

Proof of Theorem 3.4: It has been shown in [ZiLb] that

$$\mathit{Compression}_{\mathcal{E},n}(\sigma_1^n) \leq \frac{c(\sigma_1^n)+1}{n}\lg(2\alpha(c(\sigma_1^n)+1)), \tag{3.8}$$

where $c(\sigma_1^n)$ is the maximum number of nodes in any parse tree[1] for σ_1^n. It is shown in [LeZ] that

$$0 \leq c(\sigma_1^n) < \frac{n\lg\alpha}{(1-\epsilon_n)\lg n}, \qquad \lim_{n\to\infty}\epsilon_n = 0. \tag{3.9}$$

Theorem 3.4 is clearly true when $c(\sigma_1^n) = o(n/\lg n)$ since $\mathit{Compression}_{\mathcal{E},n}(\sigma_1^n) \sim 0$ as $n \to \infty$. When $c(\sigma_1^n) = O(n/\lg n)$, using the lower bound for $\mathit{Compression}_{M(s),n}(\sigma_1^n)$ from Lemma 3.3 and the upper bound for $\mathit{Compression}_{\mathcal{E}}(\sigma_1^n)$ from (3.8), we get by simple arithmetic that

$$\lim_{n\to\infty}\left(\mathit{Compression}_{\mathcal{E},n}(\sigma_1^n) - \mathit{Compression}_{M(s),n}(\sigma_1^n)\right) = 0.$$

By (3.9), no more possibilities exist for $c(\sigma_1^n)$ and the theorem stands proved. □

If our Markov source M has τ states, it clearly belongs to the set $M(\tau)$, and M compresses no better than the best automaton in $M(s)$, $s \geq \tau$, for all sequences σ_1^n produced by it. Using this fact in Theorem 3.4, taking the expected value of both sides of the inequality, and using expression (3.3) we get the following corollary that the encoder $\mathcal{E}$ compresses as well as possible, achieving the entropy of the source M in the limit.

Corollary 3.2 *Let M be a Markov source with s states. Then we have*

$$E\left(\mathit{Compression}_{\mathcal{E},n}\right) \leq H_M(n) + \delta'_s(n), \qquad \lim_{n\to\infty}\delta'_s(n) = 0.$$

[1] This is not the definition of $c(\sigma_1^n)$ in [ZiLb] but it is easy to verify that the proofs in [ZiLb] also hold under this definition of $c(\sigma_1^n)$.

3.3.2 Bounds on Fault Rate

Along the lines of entropy in Definition 3.2, we introduce the corresponding notion of the *minimum expected fault rate* F_M of a Markov source M. It is the expected fault rate achieved in prefetching by the best algorithm that fully knows the source. As mentioned before, with a slight abuse of notation, we denote this best algorithm also by M. When the source is in some state z, M puts into the cache those k pages having the maximum probabilities for state z.

Definition 3.4 Let M be a Markov source. Let $K_z(M)$ be a set of pages with the maximum k probabilities at state z. Then the minimum expected fault rate of M on inputs of length n is defined by

$$F_M(n) = \frac{1}{n} \sum_{z \in S} \left(\sum_{v=0}^{n-1} \mathbf{Pr}(z, v) \right) \left(\sum_{i \notin K_z(M)} p_{z,i} \right) .$$

If we take the limit of $F_M(n)$ as $n \to \infty$, we get the *minimum expected fault rate* F_M of M.

We now come to our goal: to show optimality of our prefetcher $\mathcal{P}$. The challenge is to show the correspondence between converging to the entropy and converging to the page fault rate.

Definition 3.5 Given a Markov source M and a sequence σ generated by M, we define the *fault rate* $Fault_{P,n}(\sigma_1^n)$ of prefetcher P to be the number of page faults incurred by P on σ_1^n, divided by n.

It is easy to prove the following lemma that M (considered as a prefetcher) has the best expected fault rate (namely, F_M) among all prefetchers when the source is M (considered as a Markov source).

Lemma 3.4 *Let M be a Markov source. The expected fault rate of any (deterministic or randomized) online algorithm P on sequences of length n satisfies*

$$E(Fault_{P,n}) \geq F_M(n).$$

Our first task is to show the following important theorem that the expected fault rate of $\mathcal{P}$ is no worse than the best possible expected fault rate F_M on sequences of length n, as $n \to \infty$. This restates in detail the first part of our first main theorem (Theorem 3.1).

Theorem 3.5 *Let M be a Markov source with s states. We have*

$$E\left(Fault_{\mathcal{P},n}\right) \leq F_M(n) + \epsilon_s(n), \qquad \lim_{n \to \infty} \epsilon_s(n) = 0.$$

To prove the above theorem, we use the following lemmas. The first gives a bound on the probability of page fault by $\mathcal{P}$ that is independent of the cache size k.

Lemma 3.5 *Suppose that at time instant θ the Markov source M is at state z and the next page request is i with probability p_i. We can think of M as a prefetching algorithm with request to the probabilities p_i. Suppose that our prefetcher $\mathcal{P}$ thinks that the next page request will be i with probability r_i. Let us denote the probability of page fault by M and $\mathcal{P}$ on the next page request by f_M and $f_{\mathcal{P}}$ respectively. We have, independently of the cache size k,*

$$f_{\mathcal{P}} - f_M \leq \sum_{i=1}^{\alpha} |p_i - r_i|.$$

Proof: Let A be the set of k pages that M chooses to put in cache, let B be the set of k pages that $\mathcal{P}$ chooses to put in cache, and let $C = A \bigcap B$. For any set X of pages, let $p_X = \sum_{i \in X} p_i$ and let $r_X = \sum_{i \in X} r_i$. Then $f_{\mathcal{P}} - f_M = (1 - p_B) - (1 - p_A) = p_A - p_B = p_{A-C} - p_{B-C}$. Let $\sum_{i=1}^{\alpha} |p_i - r_i| = \epsilon$. Then $p_{A-C} = r_{A-C} + \epsilon_1$ and $p_{B-C} = r_{B-C} + \epsilon_2$, where $|\epsilon_1| + |\epsilon_2| \leq \epsilon$. Since $\mathcal{P}$ chooses those k pages that have the top k probabilities amongst the r_i, $1 \leq i \leq \alpha$, we have $r_{B-C} \geq r_{A-C}$. Hence $f_{\mathcal{P}} - f_M = p_{A-C} - p_{B-C} \leq \epsilon_1 + \epsilon_2 \leq |\epsilon_1| + |\epsilon_2| \leq \epsilon$. □

The summation on the right-hand side of the next lemma is the Kullback-Leibler divergence of $(r_1, \ldots, r_\alpha)$ with respect to $(p_1, \ldots, p_\alpha)$.

Lemma 3.6 *Given two probability vectors $(p_1, \ldots, p_\alpha)$ and $(r_1, \ldots, r_\alpha)$, we have*

$$\left(\sum_{i=1}^{\alpha} |p_i - r_i| \right)^2 \leq 2 \sum_{i=1}^{\alpha} p_i \ln \frac{p_i}{r_i}.$$

Proof: The above lemma is well known; however, we reproduce the proof from [AmM] here for the sake of completeness. From the fact that $x \ln x - x + 1 \geq 0$ and that $3(x-1)^2 \leq (4x+2)(x \ln x - x + 1)$, we get $|x - 1| \leq \sqrt{(4x+2)/3}\sqrt{x \ln x - x + 1}$. Using this inequality at each term of the the left hand side of the inequality to be proved, and applying the Cauchy Schwartz inequality, we obtain

$$\begin{aligned}
\left(\sum_{i=1}^{\alpha} |p_i - r_i| \right)^2 &= \left(\sum_{i=1}^{\alpha} r_i \left| \frac{p_i}{r_i} - 1 \right| \right)^2 \\
&\leq \frac{1}{3} \left(\sum_{i=1}^{\alpha} \left(4\frac{p_i}{r_i} + 2 \right) r_i \right) \cdot \left(\sum_{i=1}^{\alpha} \left(\frac{p_1}{r_i} \lg \frac{p_i}{r_i} - \frac{p_i}{r_i} + 1 \right) r_i \right) \\
&= 2 \sum_{i=1}^{\alpha} p_i \ln \frac{p_i}{r_i}.
\end{aligned}$$

The last step above follows from the fact that $\sum_{i=1}^{\alpha} p_i = \sum_{i=1}^{\alpha} r_i = 1$. □

Lemma 3.7 *Let $\sum_{i=1}^{u} \gamma_i x_i$ be a convex linear combination of the nonnegative quantities x_i; that is, $\gamma_i \geq 0$ and $\sum_{i=1}^{u} \gamma_i = 1$. Then*

$$\left(\sum_{i=1}^{u} \gamma_i \sqrt{x_i} \right)^2 \leq \sum_{i=1}^{u} \gamma_i x_i.$$

Proof: The lemma follows from the square root function being concave. □

We are now ready to prove Theorem 3.5.

Proof of Theorem 3.5: The basic idea of the proof is to examine the behavior of the prefetcher $\mathcal{P}$ whenever the Markov source M is in a particular state z. One big difficulty is coping with the fact that the optimal prefetcher M always prefetches the same k pages at state z, whereas our prefetcher $\mathcal{P}$'s probabilities may differ significantly each time M is in state z, since it is context-dependent. To get over this problem, we map the differences over contexts to the state z and weight by the probability of being in state z.

Let z_0 be the start state and S be the set of states of the source M. In the definition of $H_M(n)$ (Definition 3.2), note that $\mathbf{Pr}(z,\ell)$ is just the sum of the probabilities of all length ℓ strings that bring M from z_0 to z. Hence[2]

$$\begin{aligned} H_M(n) &= \frac{1}{n}\sum_{z\in S}\sum_{\ell=0}^{n-1}\sum_{\text{all } \sigma_1^\ell} \mathbf{Pr}(\sigma_1^\ell)\cdot[g(z_0,\sigma_1^\ell)=z]\cdot H_M(z) \\ &= \frac{1}{n}\sum_{z\in S}\sum_{\ell=0}^{n-1}\sum_{\text{all } \sigma_1^\ell} \mathbf{Pr}(\sigma_1^\ell)\cdot[g(z_0,\sigma_1^\ell)=z]\cdot\sum_{i=1}^{\alpha} p_{z,i}\lg\frac{1}{p_{z,i}}, \end{aligned} \tag{3.10}$$

where $\mathbf{Pr}(\sigma_1^0)=1$, $g(z_0,\sigma_1^0)=z_0$, and $H_M(z)$ is the minimum average encoding length of the source at state z and is equal to $\sum_{i=1}^{\alpha} p_{z,i}\lg(1/p_{z,i})$.

Let $\mathit{Compression}_{\mathcal{E}}^{\ell+1}(\sigma_1^\ell, z, \sigma_{\ell+1}^{\ell+1})$ be $\mathcal{E}$'s encoding length for the $(\ell+1)$st character $\sigma_{\ell+1}^{\ell+1}$, given that M is in state z after emitting the first ℓ characters σ_1^ℓ. We have

$$\mathit{Compression}_{\mathcal{E},n}(\sigma_1^n) = \frac{1}{n}\sum_{\ell=0}^{n-1}\mathit{Compression}_{\mathcal{E}}^{\ell+1}(\sigma_1^\ell, z, \sigma_{\ell+1}^{\ell+1}).$$

We would like to express $E(\mathit{Compression}_{\mathcal{E},n})$ in a form similar to (3.10). If we specify the first ℓ characters and leave the $(\ell+1)$st character $\sigma_{\ell+1}^{\ell+1}$ unspecified, we get the random variable $\mathit{Compression}_{\mathcal{E}}^{\ell+1}(\sigma_1^\ell, z)$ with mean $\sum_{i=1}^{\alpha} p_{z,i}\lg(1/r_{\sigma_1^\ell,i})$, where $r_{\sigma_1^\ell,i}>0$ is the probability with which $\mathcal{E}$ expects to see character i next after having processed σ_1^ℓ in which M ends up in state z. We have

$$\begin{aligned} E(\mathit{Compression}_{\mathcal{E}}^{\ell+1}) &= \sum_{z\in S}\sum_{\text{all } \sigma_1^\ell} \mathbf{Pr}(\sigma_1^\ell)\cdot[g(z_0,\sigma_1^\ell)=z]\cdot E(\mathit{Compression}_{\mathcal{E}}^{\ell+1}(\sigma_1^\ell, z)) \\ &= \sum_{z\in S}\sum_{\text{all } \sigma_1^\ell} \mathbf{Pr}(\sigma_1^\ell)\cdot[g(z_0,\sigma_1^\ell)=z]\cdot\sum_{i=1}^{\alpha} p_{z,i}\lg\frac{1}{r_{\sigma_1^\ell,i}}. \end{aligned} \tag{3.11}$$

Summing on ℓ in (3.11), we get

$$\begin{aligned} E(\mathit{Compression}_{\mathcal{E},n}) &= \frac{1}{n}\sum_{\ell=0}^{n-1} E(\mathit{Compression}_{\mathcal{E}}^{\ell+1}) \\ &= \frac{1}{n}\sum_{z\in S}\sum_{\ell=0}^{n-1}\sum_{\text{all } \sigma_1^\ell} \mathbf{Pr}(\sigma_1^\ell)\cdot[g(z_0,\sigma_1^\ell)=z]\cdot\sum_{i=1}^{\alpha} p_{z,i}\lg\frac{1}{r_{\sigma_1^\ell,i}}. \end{aligned} \tag{3.12}$$

Combining (3.12) and (3.10), we get

$$\begin{aligned} &E(\mathit{Compression}_{\mathcal{E},n}) - H_M(n) = \\ &\qquad \frac{1}{n}\sum_{z\in S}\sum_{\ell=0}^{n-1}\sum_{\text{all } \sigma_1^\ell} \mathbf{Pr}(\sigma_1^\ell)\cdot[g(z_0,\sigma_1^\ell)=z]\cdot\sum_{i=1}^{\alpha} p_{z,i}\lg\frac{p_{z,i}}{r_{\sigma_1^\ell,i}}. \end{aligned} \tag{3.13}$$

[2]We use the notation [*relation*] to denote 1 if *relation* is true and 0 if *relation* is false.

Our goal is to express the quantity $E(Fault_{\mathcal{P},n}) - F_M(n)$ in a way similar to (3.13). By analogy to the expression (3.10) for $H_M(n)$, we have

$$\begin{aligned} F_M(n) &= \frac{1}{n}\sum_{z\in S}\sum_{\ell=0}^{n-1}\sum_{\text{all } \sigma_1^\ell} \mathbf{Pr}(\sigma_1^\ell)\cdot[g(z_0,\sigma_1^\ell)=z]\cdot F_M(z) \\ &= \frac{1}{n}\sum_{z\in S}\sum_{\ell=0}^{n-1}\sum_{\text{all } \sigma_1^\ell} \mathbf{Pr}(\sigma_1^\ell)\cdot[g(z_0,\sigma_1^\ell)=z]\cdot \sum_{i\notin K_z(M)} p_{z,i}, \end{aligned} \tag{3.14}$$

where $F_M(z)$ is the expected page fault rate at state z of M, and $K_z(M)$ is a set of pages with the maximum k probabilities at state z.

By further analogy, we define $Fault_{\mathcal{P}}^{\ell+1}(\sigma_1^\ell, z, \sigma_{\ell+1}^{\ell+1})$ be the 0–1 quantity denoting whether prefetcher $\mathcal{P}$ faults on the $(\ell+1)$st page request $\sigma_{\ell+1}^{\ell+1}$, given that M is in state z after emitting the first ℓ page requests σ_1^ℓ. We have $Fault_{\mathcal{P},n}(\sigma_1^n) = (1/n)\sum_{\ell=0}^{n-1} Fault_{\mathcal{P}}^{\ell+1}(\sigma_1^\ell, z, \sigma_{\ell+1}^{\ell+1})$. If we specify the first ℓ page requests and leave the $(\ell+1)$st page request $\sigma_{\ell+1}^{\ell+1}$ unspecified, we get the random variable $Fault_{\mathcal{P}}^{\ell+1}(\sigma_1^\ell, z)$ with mean $\sum_{i\notin K_z(\mathcal{P},\sigma_1^\ell)} p_{z,i}$, where $K_z(\mathcal{P},\sigma_1^\ell)$ is the set of k pages that $\mathcal{P}$ puts into its cache after processing σ_1^ℓ in which M ends up in state z. We have

$$\begin{aligned} E(Fault_{\mathcal{P}}^{\ell+1}) &= \sum_{z\in S}\sum_{\text{all } \sigma_1^\ell} \mathbf{Pr}(\sigma_1^\ell)\cdot[g(z_0,\sigma_1^\ell)=z]\cdot E(Fault_{\mathcal{P}}^{\ell+1}(\sigma_1^\ell, z)) \\ &= \sum_{z\in S}\sum_{\text{all } \sigma_1^\ell} \mathbf{Pr}(\sigma_1^\ell)\cdot[g(z_0,\sigma_1^\ell)=z]\cdot \sum_{i\notin K_z(\mathcal{P},\sigma_1^\ell)} p_{z,i}. \end{aligned} \tag{3.15}$$

Summing on ℓ in (3.15), we get

$$\begin{aligned} E(Fault_{\mathcal{P},n}) &= \frac{1}{n}\sum_{\ell=0}^{n-1} E(Fault_{\mathcal{P}}^{\ell+1}) \\ &= \frac{1}{n}\sum_{z\in S}\sum_{\ell=0}^{n-1}\sum_{\text{all } \sigma_1^\ell} \mathbf{Pr}(\sigma_1^\ell)\cdot[g(z_0,\sigma_1^\ell)=z]\cdot \sum_{i\notin K_z(\mathcal{P},\sigma_1^\ell)} p_{z,i}. \end{aligned} \tag{3.16}$$

Combining (3.14) and (3.16), we get

$$\begin{aligned} &E(Fault_{\mathcal{P},n}) - F_M(n) = \\ &\quad \frac{1}{n}\sum_{z\in S}\sum_{\ell=0}^{n-1}\sum_{\text{all } \sigma_1^\ell} \mathbf{Pr}(\sigma_1^\ell)\cdot[g(z_0,\sigma_1^\ell)=z]\cdot\left(\sum_{i\notin K_z(\mathcal{P},\sigma_1^\ell)} p_{z,i} - \sum_{i\notin K_z(M)} p_{z,i}\right). \end{aligned} \tag{3.17}$$

From Lemma 3.5 and (3.17) we get

$$\begin{aligned} &E(Fault_{\mathcal{P},n}) - F_M(n) \le \\ &\quad \frac{1}{n}\sum_{z\in S}\sum_{\ell=0}^{n-1}\sum_{\text{all } \sigma_1^\ell} \mathbf{Pr}(\sigma_1^\ell)\cdot[g(z_0,\sigma_1^\ell)=z]\cdot\sum_{i=1}^{\alpha}|p_{z,i} - r_{\sigma_1^\ell,i}|. \end{aligned} \tag{3.18}$$

Let us denote the term $\sum_{i=1}^{\alpha}|p_{z,i} - r_{\sigma_1^\ell,i}|$ in (3.18) by $\epsilon(z,\ell,\sigma_1^\ell)$, and let us denote the term $\sum_{i=1}^{\alpha} p_{z,i}\lg(p_{z,i}/r_{\sigma_1^\ell,i})$ in (3.13) by $\delta(z,\ell,\sigma_1^\ell)$. Lemma 3.6 bounds $\epsilon(z,\ell,\sigma_1^\ell)$ by $\sqrt{(\ln 4)\delta(z,\ell,\sigma_1^\ell)}$. By three applications of Lemma 3.7 to (3.13) and (3.18) and by the bound on $E(Compression_{\mathcal{E},n}) - H_M(n)$ from Corollary 3.2, we get our result. The quantity $\epsilon_s(n)$ is bounded from above by $\sqrt{(\ln 4)\delta'_s(n)}$. □

The following theorem is the detailed version of the second part of our first main theorem (Theorem 3.1):

Theorem 3.6 *Let the page request sequence σ of length bn be generated by an ergodic Markov source M. We have*

$$\mathit{Fault}_{\mathcal{P},nb} \to E(\mathit{Fault}_{\mathcal{P},n}) \qquad \text{for almost all sequences } \sigma, \text{ as } b \to \infty$$

where, by Theorem 3.5,

$$\lim_{n\to\infty} E(\mathit{Fault}_{\mathcal{P},n}) = F_M.$$

Proof: Given a sequence σ_1^{bn}, we divide it into b blocks each of length n. The net fault rate $\mathit{Fault}_{\mathcal{P},nb}(\sigma)$ is $\sum_{i=1}^{b} \mathit{Fault}_{\mathcal{P},n}(\sigma_{(i-1)n+1}^{in})/b$. Since we throw away our data structures at the end of each block, each of the b random variables $\mathit{Fault}_{\mathcal{P},n}$, for $1 \le i \le b$, depends only on the start state for each block, and our result follows by the ergodic theorem [Gal]. □

The proof of Theorem 3.2 essentially deals with showing that F_σ converges to F_M for almost all σ as $n \to \infty$. We call two prefetchers M_1 and M_2 *distinct* if there exists some time instant θ and some page request sequence σ such that M_1 and M_2 prefetch different sets of pages at time θ on σ. Let $M_{opt}(s)$ be a maximal set of probabilistic FSAs with s states that are distinct when considered as prefetchers and that are each optimal for some page request sequence. We now see that $|M_{opt}(s)|$ is finite.

Lemma 3.8 *The cardinality of set $M_{opt}(s)$ is dependent only on s, α, and k, and is independent of n.*

Proof: Let $M' \in M_{opt}(s)$ be the best prefetcher for some page request sequence σ_1^n. Let us trace σ_1^n through M', counting the number of times each transition is traversed. It is clear that the strategy of M' would be to prefetch at each state those k pages corresponding to the k maximum count transitions out of that state. Hence M' could be considered as a finite state predictor with at most s states and k transitions out of each state corresponding to the k pages it prefetches at that state. The number of distinct FSAs with at most s states and k transitions out of each state is clearly dependent only on s, α, and k. □

To prove Theorem 3.2, we first show that F_σ approaches F_M for almost all page request sequences σ. By Theorem 3.1, we know that the fault rate by $\mathcal{P}$ approaches F_M for almost all page request sequences. Theorem 3.2 then follows by transitivity.

Proof of Theorem 3.2: Let $M = (S_m, A, g_m, p_m, z_m)$ be the Markov source, and let $M' = (S_{m'}, A, g_{m'}, p_{m'}, z_{m'}) \in M_{opt}(s)$. Let us define prefetcher $X = (S_x, A, g_x, p_x, z_x)$ to be a type of "cross product" of M and M', where $S_x = S_m \times S_{m'}$, $g_x((z_i, z_j), a) = (g_m(z_i, a), g_{m'}(z_j, a))$, and $p_x((z_i, z_j), a) = p_m(z_i, a)$. At state $(z_i, z_j) \in S_x$, X prefetches those k pages that M prefetches at $z_i \in S_m$, that is, the pages with the top k probabilities at (z_i, z_j).

Let us consider a state $z \in S_x$. Given a sequence σ of length n, let f_z be the number of times σ reaches state z, divided by n, and let $f_{z,i}$ be the number of times σ takes transition i out of state z, divided by the total number of transitions taken by σ out of state z. The probability of sequences of length n for which

$$\sum_{z \in S_x, i \in A} f_z \, f_{z,i} \ln \frac{f_{z,i}}{p_{z,i}} \ge \delta$$

is exponentially small in n [Nat, Theorem 2], for $\delta > 0$. By Lemmas 3.5 and 3.6, we have

$$E(Fault_{X,n}) - F_\sigma \geq \sqrt{2\delta}$$

with exponentially small probability. By the definition of fault rate and Lemma 3.4, it follows that $E(Fault_{X,n}) = F_M(n) \leq E(Fault_{M',n})$. Since by Lemma 3.8 $|M_{opt}(s)|$ is finite, it follows that for any $\epsilon > 0$,

$$F_M(n) - F_\sigma \geq \epsilon$$

with exponentially small probability. Thus, F_σ converges to F_M for almost all page request sequences σ.

From Theorem 3.1, $\mathcal{P}$'s fault rate converges to F_M for almost all page request sequences σ. By transitivity, $\mathcal{P}$'s fault rate converges to F_σ for almost all page request sequences σ. □

3.4 The mth order Markov Source Model

It is easier to prefetch optimally under the second model (when M is an mth order Markov source) because we implicitly know the transitions between the states of M. The only problem that remains is to estimate the probabilities on the transitions. The probability of the next character is dependent only on the past m characters. Our online algorithm for prefetching $\mathcal{M}$ builds the finite state machine of the source, the states labeled with m-contexts and the transitions denoting the unique transitions from one m-context to the next. The prefetcher estimates the probability of each transition to be the frequency that it is taken. These frequencies converge to the actual probabilities exponentially fast [Nat].

Since the state structure of the source is known, $\mathcal{M}$ is always in the same state that the source M is in. Let the algorithm $\mathcal{M}$ prefetch the pages with the top k estimated probabilities. The convergence of the fault rate of $\mathcal{M}$ to the fault rate of the source is related to the convergence of the estimated probabilities on all transitions to the actual probabilities, as given in Lemma 3.5. Hence the fault rate convergence also takes place exponentially fast. Theorem 3.3 follows. Knowing the structure of the source thus allows us to prove a faster rate of convergence.

3.5 Discussion

In this chapter, we started with the intuition that prediction is closely related to data compression, and that a good data compressor should be able to predict well for prefetching purposes. We then constructed a universal prefetcher $\mathcal{P}$, based on the Ziv-Lempel data compression algorithm, that prefetches optimally in the limit for almost all sequences emitted by a Markov source. An interesting feature of the proof of optimality is that is not derived from first principles; for proving fault rate convergence, we explicitly use the data compression convergence results. The proofs provide a direct intuition into the close relationship between data compression and prefetching. We shall see in Chapter 5 that this intuition provides us with several practical prefetchers.

Our prefetching algorithms are adaptive and based only on the page request sequence. They do not attempt to take advantage of possible knowledge of the application that is issuing the requests. In practice, when such knowledge is available [PGS], we could combine

our prefetcher with a logical prefetcher based on the semantics of the application, so as to get the best of both worlds. Similar techniques for cache replacement appear in [FKL].

The framework of Abe and Warmuth [AbW], who investigated a quite different learning problem related to FSAs, suggests a static PAC-learning framework for prefetching, in which the prefetcher is trained on several independently generated sequences of a particular length. A harder model is to assume that the prefetcher is trained on one sufficiently long sequence generated by a source. For certain special cases of sources, like mth order Markov sources, we expect that the optimal prefetcher is PAC-learnable. An interesting related model is that of probabilistic concepts [KeS]. We can modify the model so that page requests are labeled by whether or not the optimal machine M faulted. (In real life, though, we wouldn't have this feedback.) The requests are generated by a Markov source rather than independently; the distance measure corresponds to the difference between the expected fault rate and the optimal fault rate.

It is important to note that the type of analysis presented in this chapter is similar to, but not exactly the same as, competitive analysis. Similar types of analysis should prove useful in establishing the goodness of online algorithms for problems that intuitively cannot admit a competitive online algorithm.

Chapter 4

Optimal Prefetching in the Worst Case

In Chapter 3, we looked at the relationship between data compression and prefetching. We proved the optimality of our pure prefetcher $\mathcal{P}$ derived from the Ziv-Lempel data compressor by assuming that the user can be modeled by powerful probabilistics models. In this chapter, we look at the much stronger form of worst-case analysis and derive a randomized pure prefetching algorithm that we prove analytically converges almost surely to the optimal fault rate in the *worst case* for every sequence of page requests with respect to the important class of finite state prefetchers. In particular, we make no assumption about how the sequence of page requests is generated. As mentioned in Chapter 3, any implementable algorithm for cache-replacement or prefetching must clearly be online. The analysis model we study in this paper can be looked upon as a generalization of the competitive framework, different from the analysis strategy in Chapter 3, in that it compares an online algorithm in a worst-case manner over all sequences against a powerful yet non-clairvoyant opponent.

An important computational requirement of prefetching (and demand fetching) algorithms is that the time spent deciding which pages to fetch into (or evict from) cache must be minimal. We show that apart from our prefetcher achieving optimality in terms of fault rate, we simultaneously achieve the computational goal of implementing our prefetcher in optimal constant expected time per prefetched page, using the optimal dynamic discrete random variate generator of [MVN].

Cache replacement has been studied by Karlin, Phillips, and Raghavan [KPR] under a different stochastic version of the competitive framework; the sequence of page requests is assumed to be generated by a Markov chain (a subset of Markov sources). A PAC learning framework incorporating Markov sources of examples is developed in [AlV].

Pure prefetching can be looked upon as the following prediction problem: Given an arbitrary alphabet of size α (the set of α pages in the database) and a sequence of (page) requests drawn from this alphabet, at each time instant we have to predict the best k choices for the k pages to prefetch into cache. Randomness is required in order for a predictor or prefetcher to be optimal [Cov]. In information theory and in statistics [Bla, CoS, FMG, Han] interesting algorithms for binary sequences (corresponding to an alphabet size of $\alpha = 2$ pages) that make one prediction for the next page (corresponding to cache size $k = 1$) have been developed, but the $\alpha = 2$, $k = 1$ case is clearly unsuitable for our prefetching scenario. The procedure in [Han] may be generalizable to the arbitrary alphabet case $\alpha \geq 2$ for cache

size $k = 1$, but it cannot possibly make a prediction in constant time independent of α, and the $k > 1$ case is open. In [MeF], predictors are developed for various continuous loss functions, but they are not relevant to the harder-to-analyze discontinuous 0–1 loss functions associated with cache replacement and prefetching.

In the light of binary predictors, our algorithm for prefetching can be looked upon as a predictor that achieves the optimal fault rate almost surely in the limit against the class of finite-state prefetchers and that is simultaneously optimal in terms of running time, for the general case of $\alpha \geq 2$ pages and cache size $k \geq 1$. ("Almost surely" means that the probability that convergence does not occur for an arbitrary sequence converges to 0 as the sequence length n gets larger.)

Our analysis model and main results are summarized in Section 4.1. In Section 4.2, we present our core prefetcher algorithm P_1, which makes use of sampling without replacement, and we analyze it in Section 4.3 by comparing it against the best one-state prefetcher. In Section 4.4 we draw on ideas from information theory [CoT] applied to predicting [CoS, FMG, MeF] and generalize P_1 to get a universal prefetcher P that is optimal in the limit against a general finite state prefetcher. The resulting optimal prefetcher P is a blend of P_1 and the prefetcher $\mathcal{P}$ from Chapter 3 based on the Ziv-Lempel data compressor [HoV, Lana, ZiLb] (on which the UNIX *compress* program is based). We show in Section 4.5 how to implement the prefetcher in constant expected time per prefetched page, independent of alphabet size α and cache size k, by use of the newly developed optimal dynamic algorithm for generating discrete random variates of Matias, Vitter, and Ni [MVN], which uses a table lookup method of Hagerup, Mehlhorn, and Munro [HMM]. In Section 4.6, we look at an interesting proof technique that shows the optimality of predictor P_1', an algorithm closely related to P_1, for general α, when $k = 1$. Other related issues are discussed in Section 4.7.

4.1 Analysis Model and Main Results

We denote the cache size by k and the total number of different pages (or alphabet size) by α. We use the notation σ_i^j to denote the subsequence of a (possibly infinite) sequence σ starting at the ith page request up to and including the jth page request; in particular, σ_1^n denotes the first n page requests of σ. Given a parsing of σ_1^n into subsequences, we will denote the jth subsequence by σ_j.

Definition 4.1 A *finite state prefetcher* (FSP) is represented as a quintuple (S, A, g, D, z_0), where S is finite set of states, $A = \{0, 1, 2, \ldots, \alpha - 1\}$ is a finite alphabet with $|A| = \alpha$, g is a deterministic "next state" function that maps $S \times A$ into S, D is a (possibly randomized) decision strategy function that maps S into a k-tuple A^k, and $z_0 \in S$ is the start state. The FSP prefetches at state $z \in S$ the k pages specified by $D(z)$, and upon seeing the next page request i, it changes state from z to $g(z, i)$. We denote the set of all FSPs with at most s states by $\mathcal{F}(s)$.

We next define the best fault rate achieved on a sequence by the class of FSPs:

Definition 4.2 Given a FSP F and a sequence σ_1^n, we denote by $\mathit{Fault}_F(\sigma_1^n)$ the fault rate of F on σ_1^n, that is, the number of faults of F on σ_1^n (expected number of faults if F has a randomized decision strategy), divided by the length n of the sequence. We define $\mathit{Fault}_{\mathcal{F}(s)}(\sigma_1^n)$ to be $\inf_{F \in \mathcal{F}(s)} \mathit{Fault}_F(\sigma_1^n)$. With a little abuse of notation we also denote by $\mathit{Fault}_B(\sigma_1^n)$ the fault rate of a non-finite-state prefetcher B.

Intuitively, we can think of $Fault_{\mathcal{F}(s)}(\sigma_1^n)$ as being given by an optimal *offline* algorithm restricted by the finite state requirement. This means that although a FSP does not know the sequence σ_1^n beforehand, it knows exactly how many times each of its transitions will be traversed when it is used to prefetch on the sequence σ_1^n. By simple convexity arguments it can be verified that the optimal FSP F for σ_1^n will, when at state z, deterministically prefetch the k pages corresponding to the k transitions out of z that are traversed the maximum number of times. (Hence $Fault_{\mathcal{F}(s)}(\sigma_1^n) = \min_{F \in \mathcal{F}(s)} Fault_F(\sigma_1^n)$, the minimum fault rate achieved by any FSP with at most s states on σ_1^n.) For example, $Fault_{\mathcal{F}(1)}(\sigma_1^n)$ is attained by the following one-state (zero-order) prefetcher F_1: Count the number of times page i, $0 \leq i \leq \alpha - 1$ appears in σ_1^n. Let $C_1(\sigma_1^n) = \{i_1, i_2, \ldots, i_k\}$ be k pages with the maximum k counts. At every time t, $1 \leq t \leq n$, predict the next page to be one of the k pages in $C_1(\sigma_1^n)$ (that is, we always keep the same k pages in cache).

We develop an online randomized prefetcher P_1 that achieves on the average the best single-state (zero-order) prefetching fault rate $Fault_{\mathcal{F}(1)}(\sigma_1^n)$ on every sequence σ_1^n of length n, in the limit as $n \to \infty$.

Theorem 4.1 *For every sequence σ_1^n of length n drawn from A, the fault rate of prefetcher P_1 on σ_1^n converges almost surely to $Fault_{\mathcal{F}(1)}(\sigma_1^n)$ as $n \to \infty$. In particular,*

$$Fault_{P_1}(\sigma_1^n) \leq Fault_{\mathcal{F}(1)}(\sigma_1^n) + O(\log n/\sqrt{n}\,). \tag{4.1}$$

The main difficulty in developing P_1 and its proof of optimality is that the alphabet size α and the cache size k are arbitrary. We note that even for the $\alpha = 2, k = 1$ case, the convergence rate cannot be faster than $O(1/\sqrt{n}\,)$ [Cov].

The importance of the above theorem lies in its generalization to higher order using techniques from information theory [CoT]. The approach of [FMG] allows us to combine P_1 with a prefetcher [ViK] based on the Ziv-Lempel data compressor [HoV, Lana, ZiLb] to get a prefetcher P that is optimal in the limit against the class of finite state prefetchers.

Theorem 4.2 *For every sequence σ_1^n of length n drawn from A, and any $s \geq 0$, the fault rate of prefetcher P on σ_1^n converges almost surely to $Fault_{\mathcal{F}(s)}(\sigma_1^n)$ as $n \to \infty$.*

The expected running time for prefetcher P can be made optimal, by use of the optimal dynamic random variate generator of [MVN]:

Theorem 4.3 *The prefetcher P runs in constant expected time (independent of α and k) for each page prefetched; that is, it requires an average of $O(k)$ time to determine which k pages to prefetch.*

The rate of convergence of Theorems 1 and 2 depends on the alphabet size α. For example, the error term is $O(\alpha k^2 \log n/\sqrt{n}\,)$ in Theorem 4.1; for simplicity we suppress the αk^2 term in our discussion since it is insignificant with respect to n in the limit. (Note that in general $k \ll \alpha$.) However, the constant time bound for each prediction is entirely independent of α and k, which is important from a computational point of view.

4.2 The Prefetching Algorithm P_1

In this section, we give the algorithm P_1 that matches the best one-state prefetcher in the limit. Before introducing P_1, we present the more intuitive algorithm P_1' upon which algorithm P_1 is based.

Let t be the current time and let σ_1^t be the sequence of t pages requested until now. Let $f_i(\sigma_1^t)$, $0 \leq i \leq \alpha - 1$, denote the number of times page i appears in σ_1^t. Define $r_t = 2^j$, when $4^{j-1} < t \leq 4^j$. That is, the integer r_t is "close to" $\sqrt{t}$; it doubles at discrete time steps (when t is one greater than a power of 4).

The key idea that prefetcher P_1' (and prefetcher P_1) uses is to reduce the problem of prediction to the problem of generating random variates. Intuitively P_1' should choose for the cache the page i with the highest or nearly highest frequency count $f_i(\sigma_1^t)$. (Randomness in the picking is required, so it does not suffice to simply pick the page with the highest frequency count.) In P_1' we get a similar effect by "boosting" the frequency counts of the page by a large power and then choosing a page with probability proportional to its boosted count. (The boosted counts will be very large, but can be represented with $O(\log n)$ bits, using the scheme discussed in Section 4.5.) Efficient random variate generation with dynamically changing weights can be done using [MVN], as discussed in Section 4.5.

The algorithm P_1' is a simple randomized weighting algorithm that makes k predictions at each time step for the next page request. At each time t and $0 \leq i \leq \alpha - 1$, P_1' assigns to page i a probability $p_{i,t}$ proportional to the boosted frequency count

$$\left(f_i(\sigma_1^t)\right)^{r_t}. \tag{4.2}$$

It predicts k items for the next request by choosing without replacement from the distribution $p_{0,t}, p_{1,t}, \ldots, p_{\alpha-1,t}$.

Example 4.1 Let $\alpha = 3$, $k = 2$. If the sequence of $t = 9$ pages σ_1^t seen until now is `210011102`, we have $f_0(\sigma_1^9) = 3$, $f_1(\sigma_1^9) = 4$, $f_2(\sigma_1^9) = 2$, and $r_t = 2^2 = 4$. Algorithm P_1' assigns probabilities to the pages as $p_{0,9} = 3^4/(3^4 + 4^4 + 2^4)$, $p_{1,9} = 4^4/(3^4 + 4^4 + 2^4)$, and $p_{2,9} = 2^4/(3^4 + 4^4 + 2^4)$. It predicts two pages out of these three by choosing without replacement based on the above probability distribution. □

We can show that algorithm P_1' is optimal against the best one-state machine for general $\alpha \geq 2$, but only when $k = 1$. (A sketch of this proof appears in Section 4.6.) We can modify algorithm P_1' to get algorithm P_1 that is optimal against the best one-state machine for general $\alpha \geq 2, k \geq 1$.

Definition 4.3 We define $\sigma_j = \sigma_1^2$ for $j = 0$ and $\sigma_j = \sigma_{4^{j-1}+2}^{4^j+1}$ for $j \geq 1$. We call σ_j the *jth r-subsequence* of σ_1^n.

Notice from the definition of r_t that P_1' predicts each page in an r-subsequence using the same value for r_t in (4.2), when $t > 2$.

Algorithm P_1 works like algorithm P_1', except that the frequency counts for the pages are reset to 0 at the start of each r-subsequence. That is, at time $t > 1$, $4^{j-1} < t \leq 4^j$, algorithm P_1 assigns to page i a probability $p_{i,t}$ proportional to

$$(f_i(\sigma_{4^{j-1}+2}^t))^{r_t}.$$

Example 4.2 As in Example 4.1, let $\alpha = 3$, $k = 2$, and the sequence of $t = 9$ pages σ_1^t seen until now be `210011102`. We have $\sigma_0 =$ `21`, $\sigma_1 =$ `001`, and the portion of σ_2 seen until now is `1102`. The counts of the pages in the current r-subsequence are 1, 2, and 1 respectively for the pages `0`, `1`, and `2`, and $r = 2^2 = 4$. Algorithm P_1 assigns probabilities to the pages as $p_{0,9} = 1^4/(1^4+2^4+1^4)$, $p_{1,9} = 2^4/(1^4+2^4+1^4)$, and $p_{2,9} = 1^4/(1^4+2^4+1^4)$. It predicts two pages out of these three by choosing without replacement based on the above probability distribution. □

This regular throwing away of past information by algorithm P_1 makes the proof of optimality more elegant. Algorithm P_1 may also perform better than algorithm P_1' in practice, since it captures the effect of locality of reference found in page request sequences [Bel, BIR, Den, IKP, KPR, ShT].

4.3 One-State Case: Optimality of P_1 vs. $\mathcal{F}(1)$

In this section we prove an important special case of Theorem 4.1, namely, that the expected value of P_1's fault rate $Fault_{P_1}(\sigma_1^n)$ converges to $Fault_{\mathcal{F}(1)}(\sigma_1^n)$; the almost-sure convergence follows by using the Borel-Cantelli lemma. As pointed out in Section 4.1, given a sequence σ_1^n, the following prefetcher $F_1 \in \mathcal{F}(1)$ attains the minimum fault rate: Count the number of times page request i, $0 \leq i \leq \alpha - 1$ appears in σ_1^n. Let $C_1(\sigma_1^n) = \{i_1, i_2, \ldots, i_k\}$ be the pages with the maximum k counts. For each time instant t, $1 \leq t \leq n$, F_1 prefetches the k pages in $C_1(\sigma_1^n)$. We have

$$Fault_{\mathcal{F}(1)}(\sigma_1^n) = 1 - \frac{f_{i_1}(\sigma_1^n) + f_{i_2}(\sigma_1^n) + \cdots + f_{i_k,n}(\sigma_1^n)}{n}. \tag{4.3}$$

We now define a *balanced form* of a subsequence, and an *approximately balanced* form of a sequence. This is useful in showing the optimality of P_1.

Definition 4.4 A subsequence $\widehat{\sigma}_a^b$ is a *balanced form* of σ_a^b if

1. $\widehat{\sigma}_a^b$ has the same composition of pages as σ_a^b, that is, for all $0 \leq i \leq \alpha - 1$, page i appears the same number of times in $\widehat{\sigma}_a^b$ as it does in σ_a^b.
2. For each $a \leq t \leq b$, page $\widehat{\sigma}_t^t$ occurs the maximum number of times in $\widehat{\sigma}_a^t$.

For example, if $\sigma_1^{10} = \texttt{1111211321}$, a balanced form is $\widehat{\sigma}_1^{10} = \texttt{123121211}$.

Definition 4.5 A sequence $\widetilde{\sigma}_1^n$ is an *approximately* or *piecewise balanced* form of σ_1^n if

$$\widetilde{\sigma}_1^n = \widehat{\sigma}_0 \widehat{\sigma}_1 \widehat{\sigma}_2 \cdots,$$

where $\widehat{\sigma}_j$ is a balanced form of σ_j, and σ_j is the jth r-subsequence of σ_1^n as defined in Definition 4.3.

By (4.3) and the first condition in Definition 4.4, we have $Fault_{\mathcal{F}(1)}(\sigma_1^n) = Fault_{\mathcal{F}(1)}(\widetilde{\sigma}_1^n)$. Our strategy to show optimality of P_1 (Theorem 4.1) is a two-step process described by the following two theorems. First, we show that the fault rate of P_1 on σ_1^n is never more than the fault rate of P_1 on the approximately balanced $\widetilde{\sigma}_1^n$.

Theorem 4.4 *For every sequence σ_1^n, we have*

$$Fault_{P_1}(\sigma_1^n) \leq Fault_{P_1}(\widetilde{\sigma}_1^n). \tag{4.4}$$

We then compute the fault rate of P_1 on the approximately balanced $\widetilde{\sigma}_1^n$ and show that it is close to the fault rate of the best one-state machine for σ_1^n.

Theorem 4.5 *For every sequence σ_1^n, we have*

$$Fault_{P_1}(\widetilde{\sigma}_1^n) - Fault_{\mathcal{F}(1)}(\sigma_1^n) = O(\log n/\sqrt{n}). \tag{4.5}$$

The proofs of the above two theorems are dealt with in the next two subsections.

4.3.1 The Approximately Balanced Sequence is Sufficiently Worst-Case

In this subsection we prove Theorem 4.4 using an interesting extension of the switch analysis of [FMG] in conjunction with the important notion of boosted frequency counts (4.2).

We denote the jth r-subsequence σ_j by π_1^η, where $\eta = 4^j - 4^{j-1}$ is the length of σ_j. The sequence π_1^η can be converted to a balanced form $\widehat{\pi}_1^\eta$ by an iterative balancing strategy. Without loss of generality, let the $(\tau+2)$nd page request in π_1^η be 1, and let the $(\tau+1)$st page request of the balanced sequence $\widehat{\pi}_1^{\tau+1}$ be 0. We use f_0 to denote the number of 0s in $\widehat{\pi}_1^\tau$, and f_1 to denote the number of 1s in $\widehat{\pi}_1^\tau$. We consider the following iterative balancing strategy to convert $\widehat{\pi}_1^{\tau+1}$ to $\widehat{\pi}_1^{\tau+2}$:

Balancing Strategy: Since $\widehat{\pi}_1^{\tau+1}$ is balanced, $f_1 \leq f_0 + 1$. If $f_1 \geq f_0$, then $\widehat{\pi}_1^{\tau+1}$ appended with a 1 gives $\widehat{\pi}_1^{\tau+2}$. If $f_1 < f_0$, we perform a "`01`" $\rightarrow$ "`10`" *switch at position* $(\tau+1, \tau+2)$ by moving the 1 in front of the 0. We continue this process of "bubbling" the 1 forward through $\widehat{\pi}_1^\tau$ by performing similar switches, until the subsequence of the first $\tau+2$ page requests of π_1^η is balanced.

Our proof of Theorem 4.4 consists in showing that each switch in the balancing strategy does not lower the page fault rate of the entire sequence.

A similar but simpler idea worked in the binary case for a different algorithm [FMG], in which the sequence did not need to be broken up into subsequences, and the sequence $\widehat{\sigma}_1^n$ could be shown to be strictly worst-case. We break σ_1^n into subsequences as part of our method for achieving optimal computational efficiency (as discussed in Section 4.5).

We now show that a switch within an r-subsequence can only increase the fault rate for algorithm P_1. The fact that we allow $k \geq 1$ predictions before each page request makes the probability terms in the analysis to be conditional on the previous prefetches at that time step, and that complicates the analysis.

Lemma 4.1 *Each switch involved in converting π_1^η to $\widehat{\pi}_1^\eta$ creates a subsequence on which P_1 has a larger fault rate (that is, switches within an r-subsequence increase the fault rate).*

Proof: Let us denote by A and B respectively the probabilities of predicting the 0 and the 1 in $\pi_1^\tau 01 \pi_{\tau+3}^\eta$, where the 01 are in the same r-subsequence. Similarly, denote by C and D the probabilities of predicting the 1 and the 0 in $\pi_1^\tau 10 \sigma_{\tau+3}^\eta$. The number of faults algorithm P_1 makes on the portions π_1^τ and $\pi_{\tau+3}^\eta$ will be the same before and after the switch, since the probability of fault by P_1 at position $(\tau+1)$ of π_1^η depends only on the composition of pages in π_1^τ. (Recall that P_1 throws away all previous counts for pages at the beginning of an r-subsequence.) To show that a switch increases the fault rate, we must show that the increase in the number of faults caused by moving the 0 to later in the sequence overshadows the decrease in the number of faults caused by moving the 1 to earlier in the sequence; that is, we need show that $(1-A)+(1-B) \leq (1-C)+(1-D)$. This is equivalent to showing that

$$A - D \geq C - B. \tag{4.6}$$

If P_1 makes only one prediction at each step, the proof of (4.6) is easy. (The proof follows directly from (4.7) for the special case $k = 1$.) However, P_1 makes $k \geq 1$ predictions at each time instant.

Let $A = A_1 + A_2 + \cdots + A_k$, where A_i is the probability of predicting a 0 in $\pi_1^\tau 01 \pi_{\tau+3}^\eta$ in the ith prediction. The probabilities B, C, D are similarly partitioned. Each A_i can be further broken up into a "good part" G_i^A and a "bad part" R_i^A. The good part G_i^A is the probability of predicting a 0 in $\pi_1^\tau 01 \pi_{\tau+3}^\eta$ in the ith prediction given that none of the previous $i-1$ predictions was a 1. The bad part R_i^A is the probability of predicting a 0 in $\pi_1^\tau 01 \pi_{\tau+3}^\eta$ in the ith prediction given that a 1 was predicted in one of the previous $i-1$ predictions. The quantities $G_i^B, R_i^B, G_i^C, R_i^C, G_i^D, R_i^D$ are similarly defined. We now show the following:

Fact 1: $G_i^A - G_i^D \geq G_i^C - G_i^B$, for $1 \leq i \leq k$;

Fact 2: $(G_i^A - G_i^D) + (R_{i+1}^A - R_{i+1}^D) = 0$, and $(G_i^C - G_i^B) + (R_{i+1}^C - R_{i+1}^B) = 0$, for $1 \leq i \leq k-1$.

In other words, the good parts of the ith prediction maintain the relationship we want. The bad parts of the ith prediction *exactly* cancel the gain from the good parts of the $(i-1)$st prediction. By definition, $R_1^A = R_1^D$, and $R_1^C = R_1^B$. The lemma follows from the above two facts:

$$A - D = \sum_{1 \leq i \leq k} A_i - D_i = \sum_{1 \leq i \leq k} (G_i^A - G_i^D) + (R_i^A - R_i^D) = G_k^A - G_k^D,$$

by repeated application of Fact 2. Similarly, $C - B = G_k^C - G_k^B$. By Fact 1, $G_k^A - G_k^D \geq G_k^C - G_k^B$, which implies $A - D \geq C - B$.

We now prove Facts 1 and 2 by induction. Let $d = \sum_0^{\alpha-1} f_i^r$, $d_1 = d - f_1^r + (f_1+1)^r$, and $d_0 = d - f_0^r + (f_0+1)^r$. These quantities are involved in the denominators of the rational expressions for the predictions. Let $u_i^A = u_i^A(x_1, \ldots, x_{i-1})$ be the term in G_i^A that corresponds to predicting $x_1, \ldots, x_{i-1}$, none of them a 0 or a 1 as the first $i-1$ predictions and 0 as the ith prediction. (The order of the first $i-1$ predictions is important. For example, when $i = 3$, the probability of predicting a 0 following x_1, x_2 is different from the probability of predicting a 0 following x_2, x_1.) The quantities u_i^B, u_i^C, u_i^D are similarly defined. The expressions in Fact 1 can be expressed in terms of the u's; for example,

$$G_i^A - G_i^D = \sum_{\substack{x_1, \ldots, x_{i-1} \\ \text{distinct}, \neq 0, 1}} u_i^A - u_i^D.$$

Let $v_i^A = v_i^A(x_1, x_2, \ldots, x_{i-1})$ be the term in R_i^A that corresponds to predicting a 0 in the ith prediction given that a 1 was one of the first $i-1$ predictions and the other $i-2$ predictions were $x_1, \ldots, x_{i-2}$, none of them a 0 or a 1. (The order of these $i-1$ predictions are important, as they are with u_i, but the relative point at which the 1 is predicted is arbitrary. In other words, v_i^A is the sum of $i-1$ probability terms corresponding to the $i-1$ positions at which a 1 can be predicted given that the other $i-2$ predictions were $x_1, \ldots, x_{i-2}$.) The quantities v_i^B, v_i^C, v_i^D are similarly defined. The expressions in Fact 2 can be expressed in terms of the u's and the v's; for example,

$$(G_i^A - G_i^D) + (R_{i+1}^A - R_{i+1}^D) = \sum_{\substack{x_1, \ldots, x_{i-1} \\ \text{distinct}, \neq 0, 1}} (u_i^A - u_i^D) + (v_{i+1}^A - v_{i+1}^D).$$

Let $den(d, i-1)$ be the $i-1$-term falling product $d(d - f_{x_1}^r) \cdots (d - f_{x_1}^r - \cdots - f_{x_{i-2}}^r)$.

Proof of Fact 1: It suffices to show by induction that $u_i^A - u_i^D \geq u_i^C - u_i^B$. For the base case when $i = 1$, $u_1^A - u_1^D = f_0^r/d - f_0^r/d_1$, and $u_1^C - u_1^D = f_1^r/d - f_1^r/d_0$. Hence,

$$\frac{u_1^A - u_1^D}{u_1^C - u_1^B} = \frac{f_0^r}{f_1^r} \times \frac{(f_1+1)^r - f_1^r}{(f_0+1)^r - f_0^r} \times \frac{d_0}{d_1} = \frac{(1+1/f_1)^r - 1}{(1+1/f_0)^r - 1} \times \frac{d_0}{d_1}. \tag{4.7}$$

Since the function $g(x) = x^r$ is convex and $f_0 \geq f_1$, the above quantity is at least 1. From the induction hypothesis that $u_{i-1}^A - u_{i-1}^D \geq u_{i-1}^C - u_{i-1}^B$ we get

$$\begin{aligned}(f_{x_1} f_{x_2} \cdots f_{x_{i-2}} f_0)^r \left(\frac{1}{den(d, i-1)} - \frac{1}{den(d_1, i-1)}\right) \\ \geq (f_{x_1} f_{x_2} \cdots f_{x_{i-2}} f_1)^r \left(\frac{1}{den(d, i-1)} - \frac{1}{den(d_0, i-1)}\right).\end{aligned} \tag{4.8}$$

By a technique similar to the one used to verify (4.7) we can show

$$\begin{aligned}\frac{(f_{x_1} f_{x_2} \cdots f_{x_{i-1}} f_0)^r}{den(d_1, i-1)} \left(\frac{1}{(d - f_{x_1}^r - \cdots - f_{x_{i-1}}^r)} - \frac{1}{(d_1 - f_{x_1}^r - \cdots - f_{x_{i-1}}^r)}\right) \\ \geq \frac{(f_{x_1} f_{x_2} \cdots f_{x_{i-1}} f_1)^r}{den(d_0, i-1)} \left(\frac{1}{(d - f_{x_1}^r - \cdots - f_{x_{i-1}}^r)} - \frac{1}{(d_0 - f_{x_1}^r - \cdots - f_{x_{i-1}}^r)}\right).\end{aligned} \tag{4.9}$$

Multiplying (4.8) by $f_{x_{i-1}}^r/(d - f_{x_1}^r - \cdots - f_{x_{i-1}}^r)$ and adding to (4.9) gives us $u_i^A - u_i^D \geq u_i^C - u_i^B$.

Proof of Fact 2: To prove that $(G_i^A - G_i^D) + (R_{i+1}^A - R_{i+1}^D) = 0$, it suffices to show that $(u_i^A - u_i^D) + (v_{i+1}^A - v_{i+1}^D) = 0$. For the base case when $i = 1$, $u_1^A - u_1^D = f_0^r(1/d - 1/d_1)$, and

$$v_2^A - v_2^D = \frac{f_1^r f_0^r}{d(d - f_1^r)} - \frac{(f_1+1)^r f_0^r}{d_1(d_1 - (f_1+1)^r)}.$$

Using the facts that $d - f_1^r = d_1 - (f_1+1)^r$, and that $d_1 - d = (f_1+1)^r - f_1^r$, it follows from simple algebra that $v_2^A - v_2^D + (u_1^A - u_1^D) = 0$.

For the inductive step, recall that v_{i+1}^A is a sum of i probability terms. (The terms are each rational expressions with different denominators.) By carefully combining terms in $v_{i+1}^A - v_{i+1}^D$, we get that $v_{i+1}^A - v_{i+1}^D$ equals

$$\frac{(f_{x_1} f_{x_2} \cdots f_{x_{i-1}} f_0)^r}{(d - f_{x_1}^r - \cdots - f_{x_{i-1}}^r - f_1^r)} \left(\frac{f_1^r}{den(d, i)} - \frac{(f_1+1)^r}{den(d_1, i)} + \frac{v_i^A - v_i^D}{(f_{x_1} f_{x_2} \cdots f_{x_{i-2}} f_0)^r}\right). \tag{4.10}$$

By the induction hypothesis, $v_i^A - v_i^D = -(u_{i-1}^A - u_{i-1}^D)$. The value for $u_{i-1}^A - u_{i-1}^D$ is the expression on the left hand side of (4.8). Substituting for $u_{i-1}^A - u_{i-1}^D$ in (4.10) we get that

$$v_{i+1}^A - v_{i+1}^D = \frac{(f_{x_1} f_{x_2} \cdots f_{x_{i-1}} f_0)^r}{(d - f_{x_1}^r - \cdots - f_{x_{i-1}}^r - f_1^r)} \times U, \tag{4.11}$$

where

$$U = \frac{f_1^r}{den(d, i)} - \frac{(f_1+1)^r}{den(d_1, i)} - \frac{(d - f_{x_1}^r - \cdots - f_{x_{i-1}}^r)}{den(d, i)} + \frac{(d_1 - f_{x_1}^r - \cdots - f_{x_{i-1}}^r)}{den(d_1, i)}.$$

The quantity $u_i^A - u_i^D$ can be expressed as

$$u_i^A - u_i^D = (f_{x_1} f_{x_2} \cdots f_{x_{i-1}} f_0)^r \left(\frac{1}{den(d,i)} - \frac{1}{den(d_1,i)} \right).$$

Adding the above expression to (4.11) and simplifying we find that $(u_i^A - u_i^D) + (v_{i+1}^A - v_{i+1}^D) = 0$. A similar analysis shows that $(G_i^C - G_i^B) + (R_{i+1}^C - R_{i+1}^B) = 0$. □

4.3.2 $Fault_{P_1}(\widetilde{\sigma}_1^n)$ is Close to $Fault_{\mathcal{F}(1)}(\sigma_1^n)$

In this subsection we prove Theorem 4.5. Let $F_1 \in \mathcal{F}(1)$ be the best one-state prefetcher for σ_1^n. Let $F_1^j \in \mathcal{F}(1)$ be the best one-state prefetcher tuned for the jth r-subsequence σ_j. Let $NumFaults_B(\sigma_a^b)$ be the number of faults incurred by algorithm B on subsequence σ_a^b.

It is clear by definition that prefetcher F_1^j incurs fewer faults on σ_j than F_1 does on σ_j. In other words,

$$NumFaults_{F_1}(\widehat{\sigma}_j) = NumFaults_{F_1}(\sigma_j) \geq NumFaults_{F_1^j}(\sigma_j) = NumFaults_{F_1^j}(\widehat{\sigma}_j). \quad (4.12)$$

Equation 4.12 directly implies the following lemma:

Lemma 4.2 *The fault rate incurred for σ_1^n by using prefetcher F_1^j to prefetch for the jth r-subsequence σ_j, for each $j \geq 0$, is no greater than $Fault_{\mathcal{F}(1)}(\sigma_1^n)$. That is,*

$$Fault_{\mathcal{F}(1)}(\sigma_1^n) = Fault_{\mathcal{F}(1)}(\widetilde{\sigma}_1^n) = \frac{NumFaults_{F_1}(\widehat{\sigma}_1^n)}{n} \geq \frac{\sum_{j \geq 0} NumFaults_{F_1^j}(\widehat{\sigma}_j)}{n}.$$

The above lemma is useful since it is easier to compare algorithm P_1 to algorithm F_1^j on page request sequence $\widehat{\sigma}_j$ than it is to compare P_1 against F_1.

Lemma 4.3 *The number of faults that algorithm P_1 incurs on $\widehat{\sigma}_j$ is close to the number of faults of the optimal one-state machine tuned for $\widehat{\sigma}_j$. In particular,*

$$NumFaults_{P_1}(\widehat{\sigma}_j) - NumFaults_{F_i^j}(\widehat{\sigma}_j) = O((\alpha k^2) j 2^j).$$

From Lemmas 4.2 and 4.3, we get

$$\begin{aligned} Fault_{P_1}(\widetilde{\sigma}_1^n) - Fault_{\mathcal{F}(1)}(\widetilde{\sigma}_1^n) &\leq \sum_j \frac{NumFaults_{P_1}(\widehat{\sigma}_j) - NumFaults_{F_1^j}(\widetilde{\sigma}_j)}{n} \\ &= O\left(\frac{\alpha k^2 \sum_j j 2^j}{n} \right) \\ &= O(\alpha k^2 \log n / \sqrt{n}). \end{aligned} \quad (4.13)$$

Theorem 4.5 follows from (4.13) and the observation following Definition 4.5.

We now give the proof of Lemma 4.3.

Proof of Lemma 4.3: For simplicity, we denote by π_1^η the balanced jth r-subsequence $\widehat{\sigma}_j$, where $\eta = 4^j - 4^{j-1}$ is the length of $\widehat{\sigma}_j$. Divide π_1^η into α subsequences $\pi_0, \pi_1, \ldots, \pi_{\alpha-1}$, where exactly $\alpha - i$ different pages appear in π_i. (Some of the π_i's may be empty.) We compute

explicitly the difference between the expected number of faults of P_1 and the number of faults of F_1^j, for each subsequence π_i. We need to be careful with the asymptotics involved, since the counts for the pages are small in the earlier part of π_1^η, but $r_t = O(\sqrt{\eta})$ is relatively larger. For simplicity, we drop the subscript t from r_t in the following discussion.

Let $|\pi_i|$ be the length of subsequence π_i. Algorithm F_1^j incurs $|\pi_i| \times \max\{0, 1-k/(\alpha-i)\}$ faults on π_i. Define L_i as

$$L_i = \begin{cases} \frac{|\pi_0|}{\alpha} + \frac{|\pi_1|}{\alpha-1} + \cdots + \frac{|\pi_{i-1}|}{\alpha-(i-1)} & \text{if } i \geq 1 \\ 0 & \text{if } i = 0. \end{cases}$$

The expected number of faults incurred by algorithm P_1 on π_i is

$$\mathit{NumFaults}_{P_1}(\pi_i) = |\pi_i| - \sum_{v=0}^{\alpha-i-1} \sum_{u=1}^{|\pi_i|/(\alpha-i)} \sum_{k_1=1}^{\min\{k,\alpha-i\}} \mathbf{Pr}(u, v, k_1), \tag{4.14}$$

where $\mathbf{Pr}(u, v, k_1)$ is the probability of predicting the $((u-1) \times (\alpha - i) + v + 1)$st page request of π_i in the k_1th prediction. It can be verified that

$$\mathbf{Pr}(u, v, k_1) \geq (\alpha - i - 1)^{\underline{k_1-1}} \prod_{w=0}^{k_1-1} \frac{(L_i + u - 1)^r}{(\alpha - i - w)(L_i + u)^r + \sum_{q=0}^{i-1} L_{q+1}^r}, \tag{4.15}$$

where the expression $x^{\underline{y}}$ stands for the "falling power" $x(x-1)\cdots(x-y+1)$. The bound in (4.15) is conservative since the terms involving the i pages not appearing in π_i are disregarded. With some manipulations to (4.15), we get

$$\mathbf{Pr}(u, v, k_1) \geq \frac{(\alpha - i - 1)^{\underline{k_1-1}}}{(\alpha - i)^{\underline{k_1}}} \left(1 - (\delta_1(u, i) + \delta_2(u, i))\right)^{k_1}, \tag{4.16}$$

where

$$\delta_1(u, i) = \frac{(L_i + u)^r - (L_i + u - 1)^r}{(L_i + u)^r}, \quad \text{and } \delta_2(u, i) = \frac{i}{\alpha - i} \times \frac{(L_i)^r}{(L_i + u)^r}.$$

With the expression for $\mathbf{Pr}(u, v, k_1)$ from (4.16) it is easy to verify that the leading term of $\mathit{NumFaults}_{P_1}(\pi_i)$ is $|\pi_i| \times \max\{0, 1 - k/(\alpha - i)\}$, which is the number of faults incurred by F_1^j on π_i. The error term $\epsilon = \mathit{NumFaults}_{P_1}(\pi_i) - \mathit{NumFaults}_{F_1^j}(\pi_i)$ equals

$$\frac{1}{\alpha - i} \sum_{v=0}^{\alpha-i-1} \sum_{u=1}^{|\pi_i|/(\alpha-i)} \sum_{k_1=1}^{\min\{k,\alpha-i\}} \epsilon(u, i, k_1) \tag{4.17}$$

where

$$\epsilon(u, i, k_1) = (1 - \delta_1(u, i) - \delta_2(u, i))^{k_1} - 1.$$

The following facts can be verified by using the asymptotic techniques from [GKP, Chapter 9]:

1. $\delta_1(u, i) \leq r/(L_i + u - 1)$ if $L_i + u - 1 \geq r$.
2. $\delta_2(u, i) \leq (i/(\alpha - i)) \times \exp(-ru/2L_i)$ if $u \leq L_i$; and $\delta_2(u, i) \leq (i/(\alpha - i)) \times 2^{-r}$ if $u \geq L_i$.

The lower order terms arising from the binomial expansion of $(1 - (\delta_1(u,i) + \delta_2(u,i))^{k_1}$ from (4.17) (i.e., terms of degree 2 or greater) can be disregarded if $L_i + u - 1 \geq kr$ and $u \geq \sqrt{3\eta}\log\alpha k$. When $L_i + u - 1 < kr$ or $u < \sqrt{3\eta}\log\alpha k$, we can bound $\epsilon(u,i,k_1)$ by 1; the net contribution of this to ϵ is $O(\alpha r \ln \alpha k)$. Disregarding lower order terms in (4.17), and using the expressions from Facts 1 and 2 above, we get $\epsilon = O(\alpha k^2 r \ln \eta + \alpha k^2 \eta / r) = O(\alpha k^2 \sqrt{\eta} \log \eta) = O(\alpha k^2 j 2^j)$. □

4.4 Generalizing P_1 to Get P

In this section we prove Theorem 4.2 by constructing our optimal prefetcher P. We obtain prefetcher P by combining prefetcher P_1 and the character-based version of the Ziv-Lempel algorithm for data compression. As we saw in Section 3.2, the original Ziv-Lempel algorithm is a word-based data compression algorithm that parses the input string x_1^n into distinct substrings x_0, x_1, x_2, ..., x_c such that, for all $j \geq 1$, substring x_j without its last character is equal to some x_i, for $0 \leq i < j$. (We use the convention that $x_0 = \lambda$, the empty substring.) It encodes the string one substring at a time. Since the substrings are prefix-closed, they can be represented by a dynamically growing tree (the "LZ tree"), with the nodes of the tree representing the substrings, and node x_i being an ancestor of node x_j if substring x_i is a prefix of substring x_j; λ is the root of the tree. An example of the LZ tree is given in Figure 4.1a.

Let $x(z)$ be the sequence of pages seen until now by P when at state z. At the end of a parse, prefetcher P positions itself at the root of the LZ tree. It looks at the subsequence $x(z)$ at its current state z, and simulates P_1 on $x(z)$ to prefetch for the next page. (Algorithm P_1 breaks $x(z)$ into r-subsequences and prefetches based on the current r-subsequence at state z as described in Section 4.2. Note that P_1 does not have to maintain $x(z)$ explicitly; it only has to maintain counts for the different pages.) On observing the next page request j, it updates $x(z)$, moves down the transition labeled by j, and prefetches the next page similarly by simulating P_1 on the sequence of pages seen at the new current state. On reaching a leaf state, it prefetches k pages at random, and the next request ends a parse. The important point is that although the counts for some or all of the transitions can be 0 (since algorithm P_1 resets the counts for all pages to 0 at the beginning of an r-subsequence), the transitions themselves are retained in the tree. An example snapshot of the data structure of P is given in Figure 4.1b.

We now briefly explain why P is optimal against an arbitrary s-state machine (Theorem 4.2), using the interesting approach of [FMG]. An mth-order Markov prefetcher predicts its k choices for the next page based solely on the previous m page requests of the sequence. The prefetcher P can be looked upon as a Markov prefetcher of growing order. If we let m be large, an mth-order Markov prefetcher achieves, for every sequence σ_1^n and any s, a fault rate close to the fault rate of the best s-state prefetcher. In the limit as $n \to \infty$, prefetcher P performs, for any m, as well as the best mth-order Markov prefetcher. Hence it achieves the fault rate of the best s-state prefetcher for any s. Given that P_1 is optimal against $\mathcal{F}(1)$ (Theorem 4.1), to show optimality of P against $\mathcal{F}(s)$ by the approach described above, we need to extend some results of [FMG] to hold for the prefetching problem. These extensions are simple as described below in Section 4.4.1. The intuition why these extensions are simple is because the comparisons are primarily between two "offline" algorithms, and the online algorithm is not much involved, as opposed to the more complex analysis of Section 4.3.

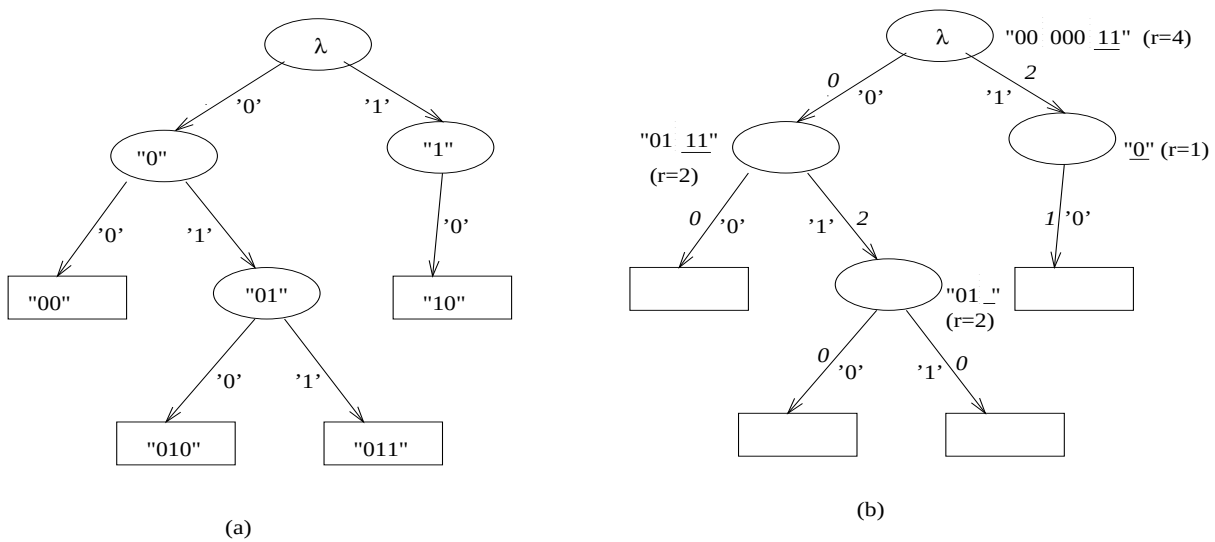


Figure 4.1: Snapshot of data structure for Algorithm P. Assume for simplicity that our alphabet is $\{0, 1\}$. We consider the page request sequence x_1^t = "00001010011110...". The Ziv-Lempel encoder parses this string as "(0)(00)(01)(010)(011)(1)(10)...". The tree that is built at the end of the seventh parse is pictured above in (a). In (b), next to each node/state z of the tree we give the sequence of page requests $x(z)$ seen at that state. For example, for any page request sequence x_1^t that is parsed by the Ziv-Lempel data compressor into distinct substrings λ, x_1, x_2, ..., x_c, the first page of each substring x_i, $1 \leq i \leq c$, forms $x(\lambda)$, the sequence of pages requested when the current state is the root of the tree. In (b), $x(\lambda) = 0000011$. The dotted vertical lines in the sequences delimit the r-subsequences, and the underlined portion is the current r-subsequence. The counts (given in italics) on the transitions out of each state z are the counts obtained by simulating P_1 on $x(z)$.

4.4.1 Proof of Theorem 4.2

In this section we give the required extensions to the results of [FMG] to prove optimality of prefetcher P.

Definition 4.6 Given an α-probability vector $\vec{p} = (p_0, \ldots, p_{\alpha-1})$, we denote by $\min_{\alpha-k}(\vec{p})$ the sum of the minimum $\alpha - k$ elements of $\vec{p}$.

In other words if $p_0 \geq p_1 \geq \cdots \geq p_{\alpha-1}$, $\min_{\alpha-k}(\vec{p}) = \sum_{i=k}^{\alpha-1} p_i$. We now prove the following two lemmas:

Lemma 4.4 *Given two α-probability vectors $\vec{p}$ and $\vec{q}$, we have*

$$\min_{\alpha-k}(\vec{p}) - \min_{\alpha-k}(\vec{q}) \leq \sum_{i=0}^{\alpha-1} |p_i - q_i|.$$

Proof: Without loss of generality assume $p_0 \geq p_1 \geq \cdots \geq p_{\alpha-1}$. Let $X = \{0, 1, \ldots, k-1\}$, and let $Y = \{k, k+1, \ldots, \alpha-1\}$. Hence, $\min_{\alpha-k}(\vec{p}) = \sum_{i \in Y} p_i$. Let $Z = \{i_1, i_2, \ldots, i_{\alpha-k}\}$ be the $\alpha - k$ pages with minimum count in $\vec{q}$. Let $U = Z \cap Y$, and $V = Z \cap X$. By definition, $\min_{\alpha-k}(\vec{p}) - \min_{\alpha-k}(\vec{q}) = \sum_{i \in Y} p_i - \sum_{i \in Z} q_i$. Since by assumption $p_0 \geq p_1 \geq \cdots \geq p_{\alpha-1}$, we have

$$\min_{\alpha-k}(\vec{p}) - \min_{\alpha-k}(\vec{q}) \leq \sum_{i \in U}(p_i - q_i) + \sum_{i \in V}(p_i - q_i) \leq \sum_{i \in A} |p_i - q_i|,$$

where A is the alphabet as given in Definition 4.1. □

Definition 4.7 An *m-th order Markov prefetcher* prefetches for its next page based solely on the previous m page requests of the sequence. Using the notation from Definition 4.1, an m-th order Markov prefetcher has α^m states, where each state is (represents) an m-context $(x_1, x_2, \ldots, x_m)$, and $g((x_1, \ldots, x_m), u) = (x_2, \ldots, x_m, u)$. We denote the fault rate of an m-th order prefetcher by $\mathit{Fault}_{M(\alpha^m)}(\sigma_1^n)$, where $M(\alpha^m) \subseteq \mathcal{F}(\alpha^m)$.

To verify Theorem 4.2, we need to show that Lemma 1, Theorem 2, and Theorem 4 from [FMG] hold for prefetching. We make the following observations:

1. In [FMG, Lemma 1, Theorems 2, 4], uniformly replace $\min\{p_0, p_1\}$, where p_0 and p_1 are the probabilities of a 0 and a 1, by $\min_{\alpha-k}(\vec{p})$, where $\vec{p}$ is the corresponding α-probability vector for prefetching. Also, replace summations over $\{0, 1\}$ by summations over the alphabet A.

2. Lemmas 4.4 and 3.6 replace the utility of [FMG,(B.1)]in the proof of [FMG, Lemma 1]. It can be verified now that [FMG,Theorem 2] holds for the prefetching problem; in particular,

$$\mathit{Fault}_{M(\alpha^m)}(\sigma_1^n) \leq \mathit{Fault}_{\mathcal{F}(s)}(\sigma_1^n) + O\left(\sqrt{\frac{\log s}{(m+1)}}\right). \tag{4.18}$$

3. Although P_1 (the online algorithm for prefetching described in Section 4.2) is very different from the algorithm for prediction of binary sequences in [FMG], algorithm P has the properties used in the proof of [FMG, Theorem 4]. In particular, the same arguments as in the proof of [FMG,Theorem 4] imply that for every page request sequence σ_1^n, and any $m \geq 0$,

$$\mathit{Fault}_P(\sigma_1^n) \leq \mathit{Fault}_{M(\alpha^m)}(\sigma_1^n) + \delta(n, m), \tag{4.19}$$

where for a fixed m, $\delta(n, m) = O(\log\log n / \sqrt{\log n}\,)$.

Theorem 4.2 follows from (4.18) and (4.19).

4.5 Constant-Time Prediction

In this section we prove Theorem 4.3 by showing that our prefetcher P runs in constant time (independent of α, k) on the average for each of the pages it prefetches into cache.

In Section 4.4, we show that it suffices to consider one-state prefetchers; the prefetcher at each step uses the appropriate P_1 to generate random variates according to a dynamically changing set of weights. We showed earlier that P_1's prediction strategy is optimal, in which we successively pick a page at random (without replacement) with probabilities in proportion to the boosted frequency counts $(f_0)^r$, $(f_1)^r$, ..., $(f_{\alpha-1})^r$, where $r \approx \sqrt{t}$. (Actually, we use $r = 2^j$, where $4^{j-1} < t \leq 4^j$, so that r seldom changes. The frequency counts f_i are reset to zero when r changes.)

The general problem of generating a random variate with a value in the range {0, 1, 2, ..., $\alpha - 1$} according to a set of α dynamically changing weights is solved optimally by Matias, Vitter, and Ni [MVN, Section 5] using the table lookup procedure of Hagerup, Mehlhorn, and Munro [HMM]. The idea at an intuitive level is to group the weights into ranges according to their values. Range j stores weights with value in the range $[2^j, 2^{j+1})$. Each range is said to have a weight equal to the sum of the weights it contains. With high probability, the individual weight chosen during the generation will be within the first $O(\log \alpha)$ ranges, so each successive group of $O(\log \alpha)$ ranges should be processed in a recursive data structure according to the weights of the ranges. The use of the rejection method [Knub] is used to adjust the probabilities of generation appropriately, since the weights in each bucket may vary by a factor of 2. After two recursive levels, the problem reduces to generating one of $O(\log \log \alpha)$ weights, each in the range $[1, \log \alpha]$, which can be done dynamically in constant time by the clever table lookup method of [HMM].

There is also extensive concern in [MVN] about the choice of hashing parameters in the universal hashing schemes used to get linear space, since no *a priori* bound on the key values is known. (In fact, a constant-time solution to the general dictionary problem is proposed in [MVN].) The model of computation allows arithmetic computation and truncated logarithms of quantities up to value $O(W)$, where W is the maximum weight.

In our application, the computation assumption of [MVN] is unreasonable. We make the strong requirement that constant-time computations must operate on operands of at most $O(\log n)$ bits, where n is the length of the sequence of page requests. However, the boosted weights $(f_i)^r$ used in the random variate generation can be as large as $n^{\sqrt{n}}$ in value, which cannot be manipulated efficiently. Fortunately, we can determine the bucket j that contains $(f_i)^r$ in constant time using operations on $O(\log n)$ bits by noting that j can be represented with only about $\lg \lg((f_i)^r) = \lg r + \lg \lg f_i \leq 2 \lg n$ bits. (By definition $\lg r$ is always an integer.) The range j can be computed therefore in constant time using $O(\log n)$-bit arithmetic.

The rejection method needed for determining whether to accept (as opposed to reject) a choice of page i must be done with acceptance probability $(f_i)^r / 2^{j+1} \geq 1/2$. This can be done conceptually by generating a uniform random integer U in the range $[1, 2^{j+1})$ and testing if $U \leq (f_i)^r$, but handling quantities of that magnitude is infeasible, as mentioned above. It suffices to determine if $\lg U \leq r \lg f_i$. This can be done in constant time by generating the exponentially distributed random variate $\lg U$ directly using finite-precision [Knub, page 128]. The expected number of bits needed before the acceptance or

rejection is determined is a small constant, so finite precision suffices. This completes the proof of Theorem 4.3.

4.6 Optimality of P_1' for $\alpha \geq 2,\ k = 1$

In this section we present a proof for the optimality of algorithm P_1'; algorithm P_1' was described in Section 4.2. The optimality of P_1' can be shown by a two-step process. First, we show that for every sequence σ, the fault rate of P_1 on σ is close to the fault rate of P_1 on $\widehat{\sigma}$. (The same notations as defined in Section 4.3 are used in this section.)

Theorem 4.6 *For every sequence σ_1^n, when $\alpha \geq 2,\ k = 1$, we have*

$$Fault_{P_1}(\sigma_1^n) - Fault_{P_1}(\widehat{\sigma}_1^n) = O(\log n/\sqrt{n}\,). \tag{4.20}$$

We then compute the fault rate of P_1 on $\widehat{\sigma}_1^n$ and show that it is close to the fault rate of the best one-state machine for σ_1^n.

Theorem 4.7 *For every sequence σ_1^n,*

$$Fault_{P_1}(\widehat{\sigma}_1^n) - Fault_{\mathcal{F}(1)}(\sigma_1^n) = O(1/\sqrt{n}\,). \tag{4.21}$$

Without loss of generality, let the $(t+2)$nd page request σ_{t+2}^{t+2} be 1, and the $(t+1)$st page request of the balanced sequence $\widehat{\sigma}_1^{t+1}$ be 0. Let f_0 be the number of 0s in $\widehat{\sigma}_1^t$, and f_1 be the number of 1s in $\widehat{\sigma}_1^t$. An iterative balance strategy (like the one described in Section 4.3.1) would convert $\widehat{\sigma}_1^{t+1}$ to $\widehat{\sigma}_1^{t+2}$. If we show that a switch produces a sequence on which P_1 has roughly the same fault rate, we know that the balanced sequence $\widehat{\sigma}_1^n$ is a near worst-case sequence. This is similar to the idea from Section 4.3.1.

The important difference is that in this case switches can occur across an r-subsequence *or across an r-boundary.* (Please see the discussion in Section 4.3.1.) From Lemma 4.1, a switch in an r-subsequence increases the fault rate. We show in Lemma 4.6 that all the switches that occur across an r-boundary cause a negligible decrease in fault rate. These two facts together prove Theorem 4.6. The proof of Theorem 4.7 is along the lines of the proof of Lemma 4.3 and is omitted.

To show that all the switches that occur across an r-boundary cause a negligible decrease in fault rate (Lemma 4.6) we use the following lemma.

Lemma 4.5 *For any ℓ, the number of switches at position $(\ell, \ell+1)$ is $\leq \ell \ln \alpha$.*

Proof: When a switch occurs at position $(\ell, \ell+1)$, the sequence σ_1^ℓ is already balanced. The page i_1 that occurs the maximum number of times in $\widehat{\sigma}_1^\ell$ will never move to the left into the ℓth position. The page i_2 that moves to the left into the ℓth position the maximum number of times has to end up after an i_1; hence the number of times a switch at position $(\ell, \ell+1)$ involves page i_2 moving to the left into the ℓth position is at most $\ell/2$. Similarly, the page that moves to the left into the ℓth position the jth maximum number of times does so at most ℓ/j times. The total number of switches is bounded by the total number of pages that move left into the ℓth position, which is bounded by $\sum_{j=2}^{\alpha-1} \ell/j \leq \ell \ln \alpha$. □

Lemma 4.6 *The net decrease in fault rate caused by all switches across all r-boundaries, involved in converting σ_1^n to $\widehat{\sigma}_1^n$ is at most $O(\log n/\sqrt{n}\,)$ when $\alpha \geq 2,\ k = 1$.*

Proof: Let $\ell = 4^j$. We consider the switches at position $(\ell+1, \ell+2)$. That is, in $\sigma_1^\ell 01 \sigma_{\ell+3}^n$, the prediction for 0 uses $r_\ell = r = 2^j$, and the prediction for 1 uses $r_{\ell+1} = 2r = 2^{j+1}$. A crucial point to note is that in the balance strategy (mentioned in Section 4.3.1), when a switch occurs, the sequence σ_1^ℓ appended with a 0 is balanced. This implies for all $1 \leq x \leq \alpha - 1$ that $f_x \leq f_0 + 1$.

The important idea of the proof is to consider the ratio $\rho_x = f_x/f_0$ for each page x. Intuitively, if this ratio is small or close to 1, the effect it has on the change of fault rate is negligible. If it is a constant fraction less than 1, the effect is important; however such a situation seldom happens.

We say that page x is in Region A if $\rho_x \leq 1/r^{1/r}$, in Region C if $\rho_x \geq 1$, and in the "bad" intermediate region B otherwise. When page x is in Regions A and B, the ratio $\rho_x = f_x/f_0$ for switches at position $(\ell+1, \ell+2)$ is non-decreasing with respect to time. Once page x enters region C, it remains in Region C, though ρ_x may fluctuate up or down between the values of 1 and $1 + 1/f_0$. The "width" of Region B is the length along the real line of the values ρ_x can take in Region B; this is at most $1 - 1/r^{1/r} = O(\log r/r)$. We look at two disjoint cases and bound the decrease in the number of faults in switches at an r-boundary in both cases.

Case 1. If there is some page x such that x is in Region B, we can bound the decrease in the number of faults by 2 (the maximum it can ever go down by). This can happen at most $O(r \log r)$ times at this r-boundary by the following reasoning: From the balance procedure, $\ell/\alpha - 1 \leq f_0 \leq \ell$. After α switches at this r-boundary, f_0 has to decrease by at least 1. The ratio ρ_x for page x goes up by at least $f_x/f_0 - f_x/(f_0 - 1)$. Since $f_x \geq f_0/r^{1/r}$, the increase in ρ_x is $\Delta\rho_x = \Omega(1/f_0 r^{1/r}) = \Omega(1/\ell)$. The total number of switches when some f_x is in Region B is at most $O(\text{"width" of interval B}/\Delta\rho_x) = O((\log r/r)/(1/\ell))$ Since $r = \Theta(\sqrt{\ell}\,)$, the net decrease in the number of faults is at most $O(\sqrt{\ell}\log\ell)$.

Case 2. In the case when all pages x lie in Regions A and C, the average number of faults can decrease by at most $O(1/r)$ for each exchange when $k = 1$. The idea is that the decrease in the number of faults is a finite sum of quantities of the form

$$\left(\frac{f_x}{f_0}\right)^r - \left(\frac{f_x}{f_0}\right)^{2r} = O\left(\frac{1}{r}\right). \tag{4.22}$$

when $k = 1$. The above does not hold when $k > 1$. By Lemma 4.5, the number of switches at this r boundary is $O(\ell)$. By (4.22), each switch decreases the number of faults by $O(1/r)$. The net decrease in the total number of faults at this r boundary in this case is therefore $O(\ell/r) = O(\sqrt{\ell}\,)$.

Cases 1 and 2 exhaust all possibilities. The net decrease in the number of faults at this r-boundary is bounded by the sum of the decreases in Cases 1 and 2; this is $O(\sqrt{\ell}\log\ell)$. Summing over all $O(\log n)$ r-boundaries and normalizing by dividing by n to get the decrease in fault rate gives us our result; that is, the net decrease in fault rate is $O(\log n/\sqrt{n}\,)$. □

Lemma 4.7 below gives us the motivation for converting algorithm P_1' to algorithm P_1 (i.e., to reset the frequency counts for all pages at the beginning of an r-subsequence).

Lemma 4.7 *When $\alpha > 2$, $k > 1$, there exists a series of switches across an r-boundary that decrease the fault rate by $\Omega(1)$.*

Proof: (Sketch) We give a brief idea of the proof. Let us use the notation from Lemma 4.6. We can create a situation where 0 is the only page in Region C, there are no pages in Region B, and there are two pages x, y with non-zero counts of at least $\sqrt{\ell}$ in Region A. We can ensure that $f_x > f_y$, and $(f_x/f_y) = O(1/b^{1/r})$, $b = O(1)$, for $O(\ell)$ switches involving 0 and y. In each of these switches, the number of faults decreases by $\Omega(1/b)$. □

4.7 Discussion

In this chapter, we have continued our study from Chapter 3 of the problem of prediction of sequences (of pages requests, for example) drawn from a finite but arbitrary alphabet of cardinality α, in which we can make, at each time step, k predictions for the next item (page). This corresponds to the problem of pure prefetching in databases. We have developed a simple randomized weighting algorithm P_1 and have combined it with the prefetcher $\mathcal{P}$ based on the Ziv-Lempel data compressor to get an efficient prefetcher P. We have shown analytically that P's fault rate converges almost surely to that of the best FSP for every (worst-case) sequence of page requests. This extends our results from Chapter 3 where we considered the sequence of pages requested as being generated by a general Markov source. It has been shown in [Cov] that any optimal algorithm for the binary alphabet case has to be necessarily randomized. By the way our algorithm is designed, we need spend at most constant expected time in making the random choices for each prediction, which is optimal. Thus, the algorithm is simultaneously optimal with respect to fault rate and running time.

An open problem is to study if there are stronger analysis models closer to the competitive model that would permit the study of prediction problems like prefetching. It would also be interesting to improve the convergence bounds while maintaining optimal running time. We conjecture that algorithm P_1' is optimal in the limit for arbitrary $k > 1$.

Chapter 5

Practical Prefetching via Data Compression[1]

The idea of using data compression techniques for prefetching was presented in Chapters 3 and 4. The intuition is that data compressors typically operate by postulating (either implicitly or explicitly) a dynamic probability distribution on the data to be compressed. Data expected with high probability are encoded with few bits, and unexpected data with many bits. Thus, if a data compressor successfully compresses the data, then its probability distribution on the data must be realistic and can be used for effective prediction. Under the pure prefetching assumption, we saw theoretically that any optimal character-by-character data compressor (for example, one derived from the Ziv-Lempel compressor for sequences of page requests generated by a finite state Markov source) can be converted to a prefetcher that has an optimal fault rate. This was extended to worst-case page request sequences in Chapter 4.

In this chapter we analyze the practical issues of using data compression techniques for prefetching. Although the pure prefetching assumption in Chapters 3 and 4 may be valid in some hypertext applications, in general the time between consecutive user page requests will not allow k prefetches at a time. It may actually be prudent in practice to prefetch less than k pages even if there is time (e.g., to avoid burning disk bandwidth). It is therefore imperative to meld good cache replacement techniques with good pure prefetchers. We address this problem in this chapter.

In our study of practical prefetching using data compression techniques, we concentrate on prefetching for only the next page request. We do not consider the compiler-directed prefetching approach (that was described in Section 2.1) of trying to time the prefetches. We discuss possible generalizations to our model in Section 5.6.

As we have seen in the previous chapters, the process of converting character-based data compressors to pure prefetchers is quite simple. However, the practical issues in prefetching are *much different* from the ones in data compression; in prefetching, time and memory issues are more significant. In this chapter we look at these problems of practical prefetching and develop solutions for them.

We look at three data compressors that perform well in practice and build simple, deterministic, universal prefetchers based on them. (A universal prefetcher makes no assumptions about the application or data representation.) Older virtual memory prefetchers that

[1]Some of the material described in this chapter is the subject of a pending patent application.

prefetch pages in sequence, that is, prefetch page $i+1$ when page i was being requested, are not universal. The usefulness of universality is extremely significant in current databases [Sal]. Any specific knowledge about the sequence of page requests can be utilized to improve the performance further using the techniques of [FKL].) We simulate our prefetchers on page request sequences derived from the Object Operations (OO1) benchmark [CaS], the OO7 benchmark [CDN], and from CAD applications used at Digital Equipment Corporation (DEC). We find that the page fault rate (number of page faults divided by the length of the request sequence) decreases significantly compared to that of demand fetching, in which the cache is organized using the least-recently used (LRU) heuristic or using the optimal offline algorithm, OPT for cache replacement [Bel] (in which the page evicted from cache is the one whose next request is furthest in the future). The reduction in fault-rate is also better than that of recent proposed schemes for prefetching [PaZb].

In Section 5.1 we describe the system environment. We describe our three prefetchers in Section 5.2. In Section 5.3 we look closely at problems stemming from memory and time restrictions unique to prefetching in some systems. We propose solutions to these problems and bound their worst-case behavior. In Section 5.4 we present our simulation environment. In Section 5.5 we give a brief description of the page request traces and present our simulation results. Other related issues are discussed in Section 5.6.

5.1 System Environment

The model that we use is the *client-server* paradigm of computing in which the client is the database user (or application) and the server manages the database. Clients make requests for data from the server and the server fulfills these requests. The client typically runs on a workstation with a modest amount of main memory (cache) and local secondary storage. Data used by an application must be in cache to be accessible. Secondary storage, which can be accessed faster than server storage, is used to store the local operating system, application programs, and is used as swap space by the workstation. The client is connected to the server over a network for communication.

The server necessarily has to manage the database because of its size, its distributed nature, and for consistency control. The server manages the database and handles requests from a number of clients. The obvious benefits of such a distributed system are well known. Prefetching reduces the effect of network latency by anticipating the client's future requests and making such requests when the network is idle. The client-server architecture is shown in Figure 5.1.

The server has the ability to handle *demand* read requests from the application and *prefetch* read requests from the prefetcher. The server gives priority to the client's requests, flushing prefetch requests in its queue when a demand request arrives. Such provisions are generally available in prefetching systems [GrR, PaZa].

The prefetcher can be either part of the application or a separate entity distinct from the application. It works by processing the sequence of the client's previous page requests and making requests for data from the server. If more specific information is available about the client's pattern of page requests, prefetching performance can be improved further. In this chapter, though, we consider prefetching based only on previous page requests.

Due to the diverse nature of user's request patterns, the improvement in fault rate will be best when each instance of an application (i.e., each user) on the client runs a copy of the prefetcher which takes into account only *its* request sequence.

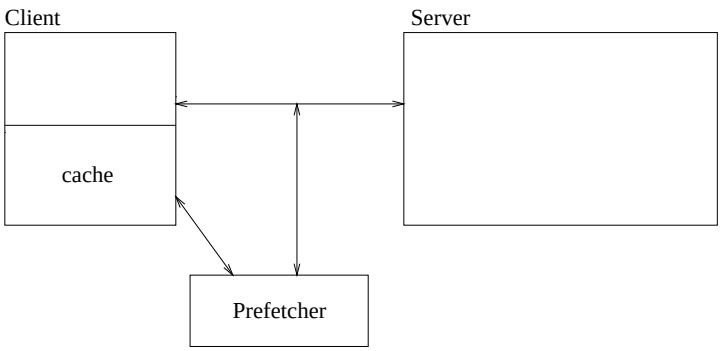


Figure 5.1: The client-server model

5.2 Algorithms for Prefetching

Let α be the alphabet size (total number of pages in the database) and k be the cache size. In typical databases, α is large and $k \ll \alpha$.

In this section, we describe our three simple, deterministic prefetching algorithms based on practical data compressors. (An elegant discussion of the data compressors appears in [BCW].) We describe our prefetchers in Sections 5.2.1–5.2.3 in their "generic" form, as pure prefetchers that can store their entire data structure in cache. These prefetchers make k suggestions for prefetch ordered by their relative merit. To make these suggestions the algorithms use $O(k)$ time. Sometimes the algorithms may have information to make $k_1 < k$ suggestions. In such cases, the remaining $k - k_1$ locations of cache are left undisturbed.

In Section 5.2.4 we look at the modification to the generic algorithm in which we must prefetch fewer than k pages at a time instant. This occurs when the time between page requests is small, or as mentioned earlier, when the prefetcher makes only $k_1 < k$ educated choices. This partial prefetching automatically introduces the problem of cache replacement; our decision strategy on which pages are evicted from cache becomes important. It is implicit in our discussion that the page the application is working on is left undisturbed; hence the actual number of pages in cache is $k + 1$. Other changes to the generic algorithms in situations that arise in practice (for example, when the data structure cannot be stored entirely in cache) are discussed in Section 5.3.

5.2.1 Algorithm LZ

Algorithm LZ is almost exactly the same as prefetcher $\mathcal{P}$ described in Section 3.2, with one difference: algorithm LZ uses a heuristic that parallels the Welsh implementation [BCW] of the Ziv-Lempel data compressor. While LZ is at a leaf, instead of fetching in k pages at random, it resets its current node to be the root (that is, it goes to the root one step early). However, it updates the transition counts for both the leaf node *and* the root.

The data structure used for prediction is a tree with at most one pointer into each node. (See Figure 3.1.) Instead of maintaining explicit probabilities on each transition, we instead maintain an (integer) count of the number of times the transition is "traversed." For example, in Figure 3.1, at node x we can store counts of 1, 3, and 1 at the three transitions (instead of the probabilities). The same comment holds for the PPM and FOM algorithms described below.

5.2.2 Algorithm PPM

Although the prefetcher $\mathcal{P}$ on which the LZ prefetcher is based is theoretically optimal in the limit, convergence to optimality is slow. This motivates us to adapt for prefetching the prediction-by-partial-match (PPM) data compressors, which perform better in practice for compression of text than the Ziv-Lempel algorithm. The motivation for algorithm PPM comes from Theorem 3.3. Although we cannot be sure that the user is an mth order Markov source, we expect that an mth order Markov source will approximate the user well.

A *jth-order Markov predictor* on page request sequence σ uses statistics of contexts of length j from the sequence to make its predictions for the next page request.

Example 5.1 Let $j = 2$, and let the page request sequence σ encountered so far be "`abbababab`". The next character is predicted based on the current context, that is, on the last $j = 2$ characters "`ab`" of σ. In σ, an "`a`" follows an "`ab`" twice, and a "`b`" follows an "`ab`" once. Hence "`a`" is predicted with a probability of 2/3, and "`b`" is predicted with a probability of 1/3. Note that if $j = 0$, each character is predicted based on the relative number of times it appears in the request sequence. □

A PPM prefetcher of order m (which we also denote as PPM-m) maintains jth-order Markov predictors (on the page request sequence seen till now) for all j, $0 \leq j \leq m$. It prefetches the k pages with the maximum k probabilities giving preference to pages predicted by higher order contexts. Formally, it executes the following loop:

for $j = m$ **down to** 0 **do**
begin
  let n_j be the number of pages in the current jth-order
    context not selected in an earlier loop;
  $t_j := \min(n_j, k - n_m - n_{m-1} - \cdots - n_{j+1})$;
  $t_j := \max(t_j, 0)$;
  select the t_j pages not selected in an earlier loop that have
    the highest probabilities in the current jth-order context;
  prefetch the selected pages not already in cache;
end

The above technique is based on the prediction by partial match algorithm for data compression using exclusion [BCW]. In our simulations we use PPM of order 3 and order 1.

The various jth-order Markov predictors, $j = 0, 1, \ldots, m$, can be represented and updated simultaneously in an efficient manner using a forward tree with vine pointers [BCW]. An example of a forward tree with vine pointers is given in Figure 5.2. The data structure is "almost" a tree; there can be more than one edge into a node because of vine pointers.

5.2.3 Algorithm FOM

Algorithm FOM is a limited memory prefetcher designed so it can always fit in a small cache. It takes as parameter a quantity w, the window size. Algorithm FOM with window size w maintains a 1st-order Markov predictor on the page request sequence formed by the last w page requests. (The 1st-order Markov predictor is explained in Section 5.2.2.) It prefetches the k pages with the maximum k probabilities as given by this 1st-order Markov predictor. We use $w = 5000$ in our experiments reported in Section 5.5.2. We would expect

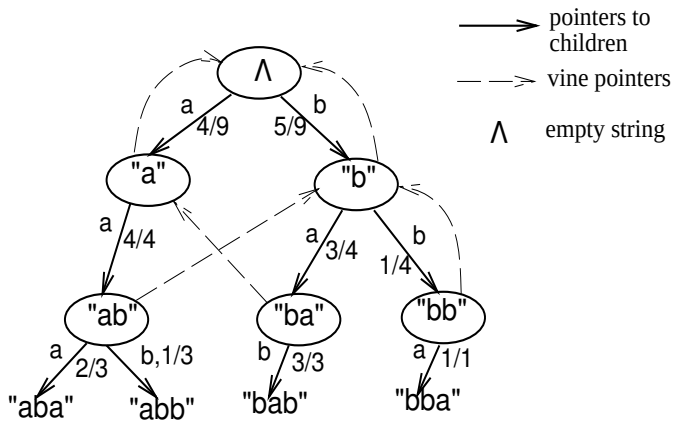


Figure 5.2: Snapshot of the data structure for PPM of order 2 when the request sequence is "`abbababab`". The vine pointer from node "`ab`" to node "`b`" indicates that if the current 2-character context is "`ab`", then the current one-character context is "`b`".

FOM with $w = \infty$ to be "close to" PPM of order 1 in performance. (Note that unlike FOM, algorithm PPM of order 1 uses an additional order-0 context for prediction.)

5.2.4 Cache Replacement Issues

Cache replacement issues automatically arise when we prefetch less than k pages; we need to decide which pages to evict from cache to make space for incoming pages. Any cache replacement algorithm can be suitably modified to work with the "generic" prefetchers described earlier. In particular, we can use the probabilities of the generic prefetcher to determine what to evict from cache, or adapt strategies like the MLP replacement strategy from [PaZb], or adapt well-known cache replacement algorithms like FIFO or LRU. In our simulations, we use a version of LRU suitably modified to handle prefetched pages. Prefetched items are put into cache as if they were demand fetched. They are marked as most recently used items, with more probable pages marked as more recently used. Prefetched data replace the least recently used pages which, if modified, are written back to disk.

5.3 Restricted Memory Environment

Our descriptions of the algorithms in Section 5.2 assume that the data structures of the prefetcher fit in cache. In some applications this is justified. However, we cannot expect all systems to have this facility.

Several techniques are known for limiting data structure size in data compressors [Stob]. An explicit upper bound M is placed on the size of the data structure. The data structure is either frozen when its size reaches M, flushed and rebuilt when its size reaches M, or frozen when its size reaches $M/2$ and a new one is built while the old one is used for prefetching. There are also more sophisticated techniques that use an LRU-type strategy on the data structure to maintain its size [BuB]. Our ongoing work studies these techniques

in the prefetching context. (We shall see later in Section 5.5 though that PPM of order 1 performs better than FOM; this suggests that placing explicit bounds on the data structure size degrades performance.)

We present the following new scheme to prefetch in a restricted memory environment.

5.3.1 Paging the Data Structure

The data structures used by our prefetchers are essentially trees (see Figure 3.1 and 5.2). Each node of the tree maintains information about its children (their counts, addresses, etc.). This information is required to make predictions for the next request. It is reasonable to assume that every node of the tree (except maybe the root) fits in at most one page of memory. (This can be ensured by simple schemes.)

We maintain some of the nodes of the tree in cache using one of many heuristics (like LRU) to decide what to evict from cache. In particular, the root is always maintained in cache. We *page in a node of the tree* when it is required. This scheme works smoothly if each node is given its own page and at least two extra I/Os can be performed between two requests (to write out the evicted node and read in the desired node).

It is more space-efficient to compact several "small" nodes into a single page and to allocate only "big" nodes to a page by themselves. In such cases, nodes may have to be moved when they threaten to overflow a page. For a pure tree data structure as in LZ (Figure 3.1), it can be verified that nodes can be reallocated to "less crowded" pages in a lazy fashion using one extra I/O for the movement, and no subsequent extra I/Os. In PPM, the node of the data structure can have many (vine) pointers into it. (See Figure 5.2.) In this case, when a node moves, it leaves back a "forwarding address," and when a vine pointer is traversed, this forwarding address pointer is "short-circuited." In the worst case there may be one extra I/O per vine pointer per reallocation (although in practice we see few reallocations and few short-circuiting of pointers). Simulations show that this technique significantly reduces paging for the data structure.

5.3.2 Sequence of Fast Page Requests

The scheme explained in Section 5.3.1 solves the limited memory problem by using disk space efficiently but creates a new "timing" problem of *fast page requests* (page requests that arrive quickly so that no I/O can be performed between them). When the data structure is always in cache, it can be updated every time even when there is no time to prefetch between page requests. If the data structure is paged, a sequence of fast page requests σ can force us to disregard important sequence information.

We have proposed and investigated the following strategy to cope with this problem: In both LZ and PPM, the counts for the pages requested in the fast sequence σ are incremented at *the current node* (that is, the node used for prediction just before σ started). We explain our scheme with an example for the LZ algorithm. (A similar scheme is used with the PPM algorithm.)

Example 5.2 Consider a subsequence "`abba...`" of a request sequence. Let the relevant nodes in the subtree for the LZ data structure be as shown in Figure 5.3a. If the subsequence of page requests is "slow" (i.e., if there is sufficient time to prefetch between requests), the data structure would look as in Figure 5.3b after this subsequence.

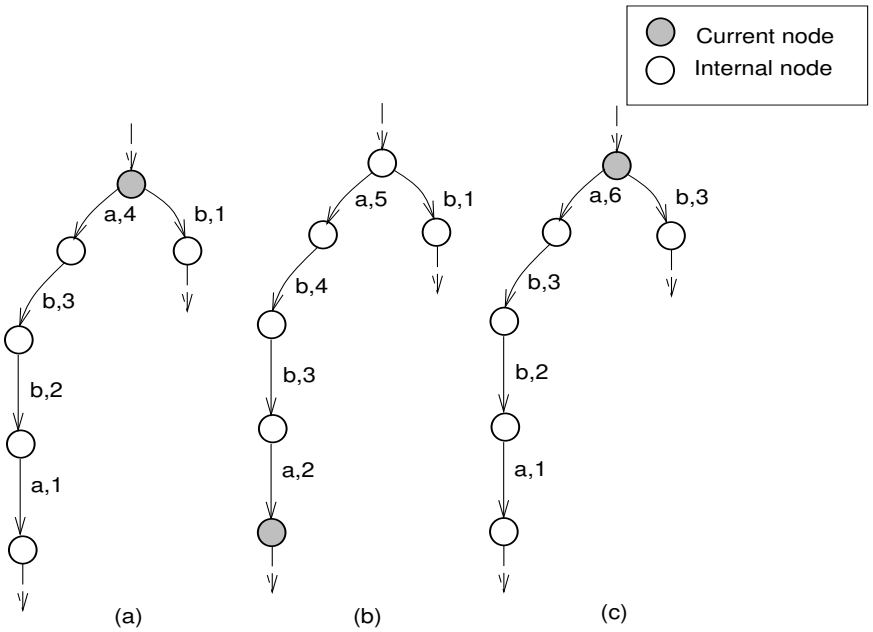


Figure 5.3: Effect of updating the LZ data structure for the subsequence "`abba...`". The transition between nodes is labeled with the page identifier and the reference count. (a) Before the subsequence. (b) After the subsequence assuming slow page requests. (c) After the subsequence assuming fast page requests.

Consider now the case where the page requests in the subsequence are fast. The current node does not change during the subsequence of fast requests. The reference counts for a and b are incremented at the current node, which is accessible to the prefetcher in cache. By assumption, a node fits on a page, so no page faults are required to update the data structure. The updated data structure is shown in Figure 5.3c. □

The intuition behind this scheme is that if the sequence of fast requests is context-dependent, accumulating statistics at the current node will aid in prefetching the correct pages in the future before the start of a fast request subsequence. By this updating strategy we encapsulate information at a node about not just the next page request but a sequence of future fast page requests.

5.4 Simulation Environment

In this section we describe the simulation environment we developed to evaluate our prefetchers. We first look at the assumptions we make for our simulation and then describe the method used for simulations.

5.4.1 Simplifying Assumptions

We bound the complexity of the simulator with the following assumptions about the application being analyzed: We assume that pages do not change their identity during a run and that they are of fixed size. As a rule of thumb, the cache size is chosen to be about 1/100 to 1/1000 of the number of distinct pages in the trace. Most of our simulations are performed

on page request traces. We also perform one set of simulations on object reference traces to aid in comparison with other prefetchers.

5.4.2 Simulation Method

Each trace (described in more detail in Section 5.5.1) is a sequence σ of page numbers requested by a database. We perform two types of simulations on each page request sequence:

Uniform Prefetching. For each page request sequence σ, we simulate each of our prefetchers from Section 5.2 on σ, prefetching d pages at each prefetch step. From Section 5.2.4 it follows that when $d = 0$ the prefetcher works as an LRU cache. This provides a basis for comparison against our prefetcher. For the FOM algorithm, we used a window size of $w = 5000$.

We measure the page fault rate for a request sequence using each of the prediction algorithms and for each value of d from 0 up to k. Statistics about the number of faults and the size of the prefetch data structure when it is allowed to grow unbounded are reported.

To analyze the situation when the data structure is paged using our strategy from Section 5.3.1, we associate with each node of the data structure a logical page number used for caching the nodes of the tree. We page the data structure just as we page the actual database, evicting (and writing out) the least-recently-used page and replacing it with the page containing the node needed by the prefetcher. The fault rate statistics are the same as without paging. We additionally report the number of data structure page I/Os. (Strictly speaking, when we page the data structure, prefetching d pages at each time step implies that we prefetch d pages and do any required data structure I/Os.)

Fast Page Request Prefetching. Fast page requests preempt any prefetching at a step in the execution of the simulator. We use our strategy from Section 5.3.2 to deal with fast requests. In order to simulate fast requests, we need either traces with detailed timing information or, alternatively, a probabilistic approach to decide when and if prefetching can occur, and if it can occur, how much data can be prefetched. Reliable timing information for purposes of prefetching is difficult to obtain. A probabilistic approach is simpler and more widely applicable and was our method of choice. Unfortunately, it removes the relationship between the previous context and the occurrence of fast requests we expect in practice and thus it provides a *conservative* estimate of prefetching performance. In practice, we expect that our prefetching algorithms will perform even better.

We supply our simulator with the (raw) page request sequence σ (used in the uniform prefetching case) and two probability parameters p, q, $0 \leq p, q \leq 1$. The parameters p and q are used to simulate a workload in a computer system. At each request, the simulator tosses a (biased) coin that lands a "head" with probability p. A "head" signifies that prefetching can be done. If the first coin lands a "head," the second (biased) coin (that lands a "head" with probability q) is repeatedly tossed until we get a "tail" or get $k - 1$ "heads." The number of "heads" from the second coin plus one gives the number of pages we can prefetch at this time instance. Setting p and q to a real number close to zero simulates fast page requests while setting p and q to a real number close to one simulates a lightly loaded system. (It can be verified that (The expected number of pages prefetched at each time step is $p(kq^k + \sum_{1 \leq t \leq k} tq^{t-1}(1-q))$.)

In this context we simulate only the LZ and the PPM algorithms. (The FOM data structure can always be updated since its data structure is always in cache.)

5.5 Experimental Results

This section presents the results of simulating our prefetcher on request traces generated by a CAD application, the Object Operations Benchmark (OO1), and the OO7 benchmark written at the University of Wisconsin [CDN]. We first describe the request traces and then present our results. In Section 5.5.4 we analyze the results; this also gives the intuition for the particular format in which the results are presented.

5.5.1 Description of the Traces

We used CAD and database traces[2] to test our prefetching algorithms. Statistics are given in Table 5.1.

CAD1 and CAD2 are object ID (UID) traces from a CAD tool written at Digital's CAD/CAM Technology Center in Chelmsford MA. We include them here as a comparison to the Fido [PaZb] algorithm that analyzed prefetching on the same traces.

The OO1 database benchmark, also known as the "Sun Benchmark," was run on the DEC Object/DB product[3] to generate page fault information for all phases of the benchmark. The more interesting phases include traversal of the structure in both the forward and reverse directions. The OO1 benchmark tests aspects of a DBMS that are critical in computer-aided software engineering (CASE) and computer-aided design (CAD) applications [CaS].

The OO7 benchmark, developed at the University of Wisconsin [CDN], tests critical aspects of object-oriented database systems not covered by other benchmarks. This suite of tests was also run on the DEC Object/DB product used for the OO1 tests. This benchmark includes tests and reports the performance of an object oriented database in the following key areas: pointer traversal, application-DBMS coupling, complex object support and long data items, updates and recovery, path indexing, caching and clustering, queries and optimization, concurrency control, and relationships and versioning. The benchmark performs traversals, associative queries, insert/delete operations, and multiuser tests [CDN]. We tested our prefetcher running with traces from the traversal query portion of the benchmark.

5.5.2 Prefetch Results for Uniform Prefetching[4]

The simulation method for uniform prefetching was described in Section 5.4.2. We depict graphically the performance of our prefetchers on trace CAD1 in Figure 5.4 and on trace OO7_T1 in Figure 5.5. The y-axis denotes the fault rate and the x-axis denotes the parameter d (the number of pages prefetched at each time step) that varies from 0 to k. When $d = 0$, the fault rate generated is exactly the fault rate of an LRU cache and is a basis for comparison with our prefetcher.

In the CAD1 trace, any prefetcher that predicts only pages previously requested must have a fault rate of at least $15,430/73,768 \approx 21\%$, by Table 5.1. The PPM-3 fault rate of 26.7% is therefore close to best possible. The graphs look similar for the other traces (except OO7_T4) and the performance numbers are given in Table 5.2. Our prefetchers' data structure size (when the data structure is allowed to grow unbounded) is given in

[2]The traces were provided as part of the DEC-ERP grant 1139.

[3]DEC Object/DB is a trademark of Digital Equipment Corporation, Maynard MA.

[4]The results reported here use slightly different parameters from the ones reported in [CKV].

Trace name	Pages requested	Unique pages	LRU	OPT
CAD1	73,768	15,430	.853	.809
CAD2	147,344	15,430	.833	.825
OO1_F	11,700	526	.941	.891
OO1_R	11,700	534	.952	.911
OO7_T1	28,103	6,033	.999	.994
OO7_T3A	30,127	6,260	.999	.994
OO7_T4	1,529	1,521	.994	.994

Table 5.1: Trace files and fault rates for LRU and OPT demand caching for cache size $k = 10$.

Trace name	LRU	FOM	LZ	PPM-1	PPM-3
CAD1	.853	.378	.397	.328	.225
CAD2	.833	.464	.358	.316	.160
OO1_F	.941	.791	.805	.781	.768
OO1_R	.952	.842	.837	.823	.784
OO7_T1	.999	.702	.682	.492	.408
OO7_T3A	.999	.723	.689	.505	.418
OO7_T4	.994	.994	.994	.994	.994

Table 5.2: Fault rates of LRU and our prefetchers for uniform prefetching with $d = 1$ (i.e., prefetching one user page between any two page requests). The cache size $k = 10$. The fault rate column for LRU is reproduced from Table 5.1 for ease of comparison.

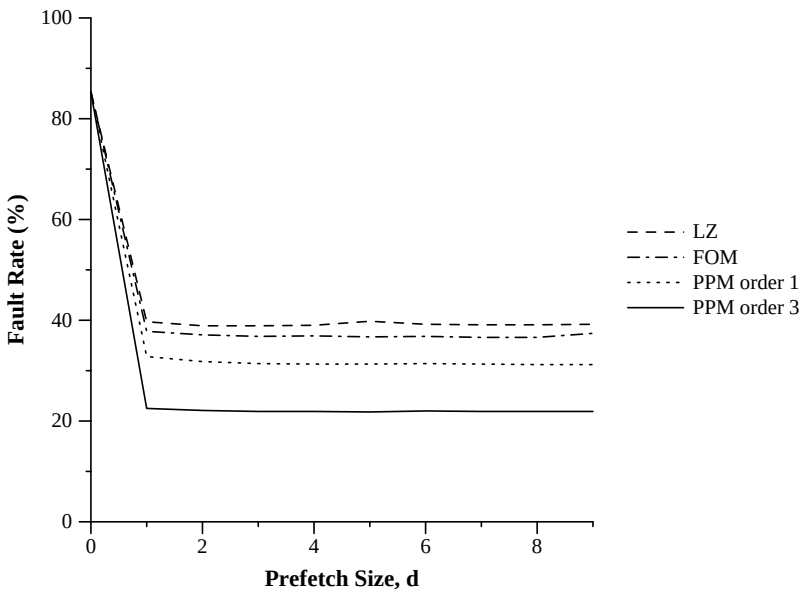


Figure 5.4: The fault rate for prefetching d objects ($0 \leq d < k$) for a fixed cache size $k = 10$ for the trace CAD1. There are 73768 object references and 15430 distinct objects in trace CAD1.

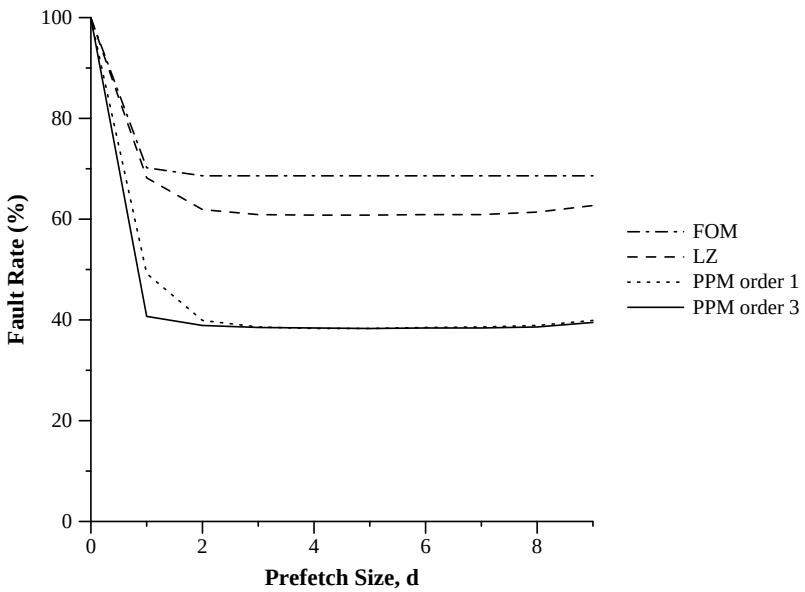


Figure 5.5: The fault rate for prefetching d objects ($0 \leq d < k$) for a fixed cache size $k = 10$ for the trace OO7_T1.

Trace name	LZ	PPM-1	PPM-3
CAD1	28,513	32,871	69,986
CAD2	44,000	35,886	85,440
OO1_F	1,792	8,807	28,127
OO1_R	1,902	8,842	28,837
OO7_T1	13,479	17,462	45,486
OO7_T3A	14,161	18,695	49,650
OO7_T4	1,525	3,048	6,108

Table 5.3: Uniform prefetching memory use in terms of number of nodes for Algorithms LZ, PPM-1, and PPM-3, when the data structure is allowed to grow without bound.

Trace name	LZ	PPM-1	PPM-3
CAD1	28151	42410	69838
CAD2	35470	68575	92499
OO1_F	1430	15254	34362
OO1_R	1540	15900	35332
OO7_T1	13117	23611	48933
OO7_T3A	13799	26128	54136
OO7_T4	1163	491	3551

Table 5.4: Data structure page I/Os for Algorithms LZ, PPM-1, and PPM-3 when 10 out of 20 pages in cache store the prefetch data structure. No optimizations for node size were done.

Table 5.3.

In Table 5.4 we give the number of data structure I/Os performed when 10 out of $k = 20$ cache pages are used for storing the prefetch data structure. (The numbers reported are conservative, since no optimizations were done for storing each node with the minimum number of bits.)

5.5.3 Prefetch Results with Fast Page Requests

Our method for simulating with fast page requests was described in Section 5.4.2. Multiple simulation runs, using different seeds in the random number generator, produced little variation in the results. We present our results of running algorithm PPM-3 on trace OO7_T1 in Figure 5.6. The cache size used is 10 pages. The x-axis denotes the probability q (that ranges from 0.0 to 1.0) and the y-axis denotes the fault rate. The lines represent the fault rate curves for different values of p; one of the lines gives the fault rate performance of LRU (our comparison base).

5.5.4 Analyzing the Results

Improvements in Fault Rate. For each of our traces, our prefetchers achieve a significantly reduced fault rate than that of a pure LRU cache and even of the OPT caching

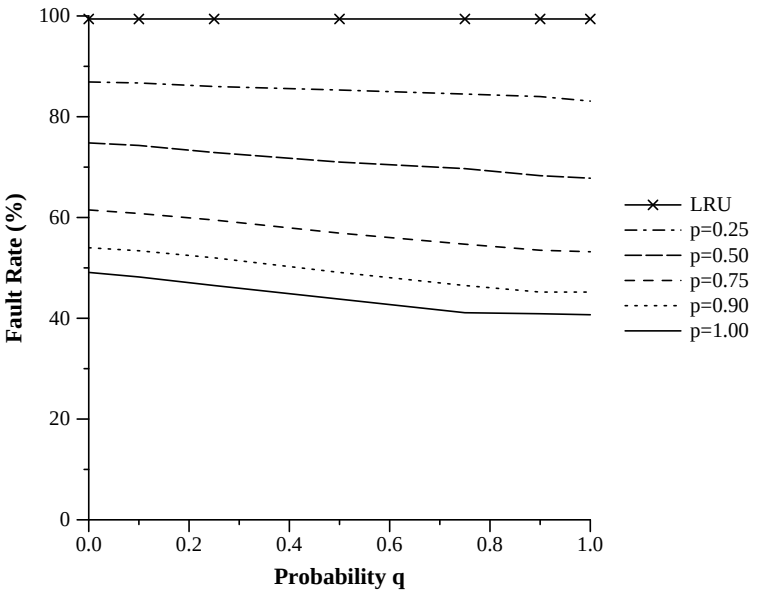


Figure 5.6: The fault rate for prefetching with the fast page request model for a cache size $k = 10$ on the trace OO7_T1 using algorithm PPM of order 3.

strategy. As seen from Tables 5.1 and 5.2, the fault rates reduce by about 60–70% for the CAD application traces, by about 15%–20% for the OO1 traces, and by about 50%–60% for the OO7 traces.

Number of Predictions. In most cases it takes only a small number of predictions (one or two) to greatly reduce the fault rate of the application. This is easily seen from Figures 5.4 and 5.5, and is true for other traces as well; this is the reason we give numbers for only $d = 1$ in Table 5.2.

Relative Performance of the Algorithms. In general, we find that the algorithms' prefetching performance relative to one another parallels their relative performance for data compression in general: $F_{\mathrm{PPM}} < F_{\mathrm{LZ}} < F_{\mathrm{FOM}}$ (where F_A is the *fault rate* for algorithm A). For traces CAD1 and OO1_F, the FOM algorithm has a lower fault rate than LZ.

Cache Size and Data Structure I/Os. Increasing the cache size by a significant factor of 5 (from say, 10 to 50) or a factor of 10 (from 10 to 100) does not lower the fault rate much. (See Table 5.5.) Although it is shown for only trace CAD1 in Table 5.5, it is true for other traces also. Hence LRU with a larger cache can be compared to our prefetcher with a smaller cache (with the remaining cache space used for storing the in-core prefetch data structures), and the gains in fault rate still hold. In addition, the number of data structure I/Os is not much (see Table 5.4); for LZ and PPM-1, for example, that is typically a fraction of a data structure I/O per page request.

Other Observations from Uniform Prefetching. The true test of a prefetcher is when the cache size is small. We have simulated using a cache size that is roughly 1/100–1/1000 of the number of distinct pages in the trace.

k	LRU	LZ	PPM-3
10	.853	.398	.225
50	.817	.391	.222
100	.760	.383	.221

Table 5.5: The effect of different cache sizes on prefetching performance for trace CAD1. The LZ and PPM-3 algorithms prefetch $d = 1$ page between any two user page requests.

The cache replacement strategy used in conjunction with the uniform prefetcher is extremely relevant. Our cache replacement scheme performs very well as seen. Some other cache replacement strategy may give even better improvements.

For comparison with Fido [PaZb], we simulated our algorithms on the same trace (CAD2) with the same cache sizes for LRU (2,000) and the prefetcher (1,500) as used in [PaZb]. Fido decreased the fault rate from 45.8% to about 23.5%. Our improvement (for PPM of order 1) was better; from 45.8% to 18.2%. (In Fido, the predictor is trained on a request sequence, the model is frozen, and it is used for prefetching on request traces from similar applications. This is in contrast to our adaptive approach which continuously learns and predicts for each request sequence. The MLP cache replacement strategy in Fido uses prediction information to both "promote" and "demote" cached pages; this idea can be used in our approach also.)

For comparison with popular heuristics, we analyzed the OO1 traces using sequential prefetching (that is prefetching page $i + 1$ after a request to page i). We found that such an approach decreases the cache fault rate only minimally (by 5%). Some traces (e.g., OO7_T4) are entirely sequential with almost all pages seen only once. In such cases simple heuristics would work well. We observe that heuristics can be melded with our prefetchers to get added performance benefits.

Fast Page Request Prefetching. The fast page request prefetching results (see Figure 5.6) suggest that the load on the system is inversely proportional to the improvement gained by prefetching and that, even under heavy load, a system with prefetching outperforms one without. These results confirm the validity of our methods for modeling fast page requests in the algorithms. The negative slope of the lines suggest that making more than one prefetch at each time step (if possible) has added benefits. This justifies the argument presented at the end of Section 5.3.2.

5.6 Discussion

We started with the theoretical results from Chapters 3 and 4 that predictors based on certain data compressors are optimal in the limit for prefetching. We observed that the practical issues involved in prefetching in databases are much different from the practical issues in data compression, and that the pure prefetching assumption, although valid for some hypertext systems, needs to be relaxed while looking at general databases. Motivated thus, we converted three practical data compressors to get three practical prefetchers. We simulated our prefetchers on page request traces generated from the OO1 and OO7 benchmarks and from CAD applications used in the industry. (The traces were obtained from

DEC.) We observed significant improvements in fault rate in comparison to using an LRU cache, and in comparison to other good prefetchers.

General predictors (except the simplest ones) can be expected to require nontrivial data structures, and these may not fit in cache for some applications. We looked at the data structures used by our algorithms, and suggested techniques for paging in the data structures efficiently with a minimum number of I/Os. We also proposed a solution to the problem of fast requests (when there is insufficient time between requests to update the paged data structures in a normal way).

An interesting result of our simulations is that the prefetching performance of our prefetchers is directly related to the compression ability of the data compressors they are derived from; in particular, algorithm PPM performs better than LZ for both compression and for prefetching. This suggests strongly that the vast research being done in developing good data compressors can be used to develop good prefetchers. Chapters 3–5 also attempt to unite two seemingly different practical fields of research. There is a note of caution required since the issues in data compression are different from the ones in prefetching; significant work is required to convert a data compressor to a prefetcher and vice-versa. We expect that the problems encountered in this task are similar to the ones addressed in this chapter.

One implementation issue we have not discussed in detail in this chapter is how to optimize for the space used by each node of the data structure. There are two promising heuristics that we experimented with prior to [CKV] which reduce the number of bits required to store the count associated with a node (or transition): the *regular halving* and the *move-to-front* heuristics.

In the regular halving heuristic, we fix the maximum count of a transition (or a node), thereby fixing the number of bits used to store the count of a transition. When the count of a transition out of node x threatens to exceed its limit, the count of each transition out of node x is halved, and the ceiling of the resulting value is taken to obtain the new count for the transition. The maximum count can be different for different levels of the tree. (Although not explicitly stated, some of the results reported in [CKV] used the regular halving heuristic; without the regular halving heursitic, the fault rates were slightly lesser than what was reported in [CKV].) In the move-to-front heuristic, instead of prefetching the pages with the maximum k counts out of the current node, we prefetch the k pages that were accessed most recently while at the node. (Notice that the move-to-front heuristic when prefetching one page between any two user page requests is related to the regular halving heuristic if we set the maximum count of each node to 1, and if, like in algorithm P from Chapter 4, we retain transitions of zero count in the model; the count of a node is defined to be the sum of the counts of transitions out of the node.) These heuristics have the intuitive advantage of exploiting locality by giving more weight to recent events while simultaneously keeping track of the contextual dependence of requests as encapsulated by the nodes of the data structure. Our observation from this chapter that the first prefetch is most important strongly suggests that it is likely that one prediction out of a node is sufficiently more likely than the others, and hence these heuristics will work well in practice. It will also be interesting from a theoretical point of view to analyze the effect of these heuristics using an appropriate model of locality as in [KPR, SlT, ViK].

As described earlier, the approach from [PaZb] is based on training the predictor on a page request sequence, fixing the predictor, and using it for subsequent user sessions. Notice that the predictors developed in this paper can also be used from session to session, by saving

the model built in earlier sessions. Independently to our work, Alexander et al. propose an interesting architecture for hardware prefetching [AlK]. Their algorithm for hardware prefetching of memory addresses into cache is closely related to the FOM algorithm using the move-to-front heuristic. For the domain of hardware prefetching, the authors also need to consider low-level timing issues; they ignore lower-order bits of the memory address and hash the address to an entry in a table that stores the predictions. Alexander et al. also study higher-order Markov models for prediction; however the necessity of having a small fixed amount of memory for hardware prefetching and the restrictions imposed by the hashing schemes make it difficult for them to gain extra performance improvements with higher-order models [Ale].

Our model of prefetching described in this chapter concentrates on prefetching for the next page request. Like in compiler-based techniques for prefetching, our model could be generalized to prefetching much in advance of an anticipated page request, by looking "down" into the model. To a certain extent, our fast page request model by accumulating statistics at the current node of the data structure tries to predict further into the future than for just the next page request.

Another important way to achieve better response time is to use clustering. Clustering is in a way dual to prefetching. Clustering algorithms attempt to improve the performance of database systems by placing related sets of objects on the same page in the hope of reducing the average number of I/Os needed to retrieve objects. There has been extensive work in clustering (e.g., [TsN] and references therein). It would be interesting to see the effect of the combination of clustering and our prefetching techniques on response-time performance. Using prefetch data structures for clustering could also be considered.

There are many open problems that this work motivates, both theoretical and practical. Can our strategy of using LRU with prefetching be shown to be optimal in some reasonable models? Otherwise, is there some other provably optimal cache replacement strategy that can be blended with prefetchers? We expect that recent work on caching models in [KPR] may be relevant. We observed that the first prefetch is almost always the most significant in terms of performance improvement. This suggests a simple theoretical model of uniform prefetching, where exactly one prefetch can be made between any two page requests. This model lies strictly between caching and pure prefetching. It would be interesting to develop provably optimal algorithms for this model of limited prefetching. Finally, can our techniques be extended for prefetching in parallel environments?

Chapter 6

Disk Spindown and Rent-to-Buy: Predicting the *When*

When shall we three meet again
In thunder, lightning, or in rain?

—William Shakespeare, *Macbeth*, (1606)

Recently, there has been an explosion in the use of battery-operated portable computers. Energy conservation is critical in these mobile systems. This has motivated both hardware and software approaches for reducing power consumption in these systems. Recent studies show that the disk sub-system on notebook computers is a major consumer of power [DKM, LKH, MDK]. Disk manufacturers are developing special types of drives especially designed for the portable market. These disks have small size, high density, high shock tolerance, and multiple energy states; in a *sleep* or *spundown* state, the disks consume comparatively no energy, but need to be spunup before data can be accessed [Cor, Pac, Qua]. The spinup takes on the order of seconds, as opposed to accessing data from a spinning disk which takes on the order of milliseconds. The disk manager needs to make online decisions on when to spin down the disk to conserve energy, while keeping latency (caused by the time taken to spinup a disk) acceptable.

Most if not all current mobile computers use a fixed threshold to determine when to spin down the disk: if the disk has been idle for some (predetermined) amount of time, the disk is spun down. The disk is spun up again upon the next access. The fixed threshold is typically on the order of many seconds or minutes to minimize the delay from on-demand disk spinups. The Hewlett-Packard Kittyhawk C3014A takes about three seconds to spindown and spin back up; its manufacturer recommends spinning it down after about five seconds of inactivity [Hewb]; most other disks take several seconds for spindown/spinup and are recommended to spindown only after a period of minutes [Del, Zen].

The main intuition underlying the variations among disk spindown policies is identifying periods of inactivity at disk that are "sufficiently large." Fixed threshold policies wait a fixed period of time to be sure that the period of inactivity is large enough. Spinning a disk for just a few seconds without accessing it can consume more power than spinning it down and spinning it up again upon the next access. Spinning down the disk more aggressively may therefore reduce the power consumption of the disk, in exchange for higher latency upon the first access after the disk has been spun down. If we can accurately predict when

the next access at disk is going to happen, we can make an optimal spindown decision. Alternatively, if we can learn a distribution of inter-arrival times at disk by observing past inter-arrival times, we can adaptively vary our spindown threshold.

Our work [DKM, KLV] described in Chapters 7–9 analyzes different methods for disk spindown: simple fixed threshold policies, predictive techniques, and adaptive methods derived from modeling the disk spindown problem by the rent-to-buy framework.

6.1 The Rent-to-Buy Framework

The disk spindown problem can be well-modeled by the the rent-to-buy framework. In the *single rent-to-buy decision* problem, without a priori knowledge of the amount of time a resource will be used, we need to decide when to buy the resource, given that we can rent the resource for \$1 per unit time or buy it once and for all for \$$c$. In the sequential rent-to-buy problem, or just the *rent-to-buy* problem, we are presented with a *sequence* of single rent-to-buy decision problems. To solve the tth single rent-to-buy problem (or the tth *round*), the online algorithm can use what it has learned from the previous $t-1$ rounds.

The disk spindown scenario can be modeled by the rent-to-buy framework as follows. A round is the time between any two requests for data on the disk. For each round, we need to solve the disk spindown problem. Keeping the disk spinning is viewed as renting, since energy is continuously expended to keep the disk spinning. Spinning down the disk is viewed as a buy, since the energy to spindown the disk and spin it back up upon the next request is independent of the remaining amount of time until the next disk access. The cost of the increased latency in serving the next disk access can also be integrated into the cost of the buy, if the algorithm is given as an input the relative importance of conserving energy and responding quickly to disk accesses.

It is easy to verify that an online algorithm that buys after waiting c units of time incurs a cost that is within a factor of two of the cost of the optimal offline algorithm; this is optimal in the worst case under the competitive model of analysis [KMM]. (Competitive analysis was defined in Chapter 3.) For a fixed c, this 2-competitive algorithm corresponds to a fixed threshold algorithm than spins down the disk after waiting c seconds.

The rent-to-buy framework is also useful in studying other related systems problems like the spin/block problem from multiprocessor applications [KLM] and deciding virtual circuit holding times in IP-over-ATM networks [SaK]. We elaborate on this in Chapter 8.

6.2 Background and Related Work

The area of disk spindown and device management has attracted a lot of attention recently. Li et al. have also investigated the issue of disk drive power management [LKH]. They used trace-driven simulation to look at a fixed threshold policy for disk spin-control, and studied buffer cache parameters. Wilkes hypothesized that it would be effective to use a weighted average of a few previous interarrival times at disk to decide when to spindown the disk on a mobile computer [Wil].

Adaptive spindown policies that continually change the spindown threshold based on the perceived inconvenience to the user are studied in [DKB]. Golding et al. [GBS] have studied idle-time detection and prediction, and proposed a taxonomy of idle-time detection

algorithms; they report the effect of their different methods in the context of the TickerTAIP simulation system [RuWb].

Greenawalt [Gre] did an analytical study of disk management strategies, assuming a Poisson process for request arrival. The single rent-to-buy problem has been studied in the worst-case setting and efficient deterministic and randomized algorithms have been developed for the problem by Karlin et al. [KMM].

Some commercial products have recently begun to address the disk spindown issue. For example, while the standard Apple Powerbook control panel only allows the user to choose broadly between "maximum conservation" and "maximum performance," the Connectix Powerbook Utilities (CPU) [Con] provide fine-grained control over such details as the disk spindown threshold and processor speed, as well as feedback on the current state of the disk (such as a count-down to when the disk will spin down).

6.3 Our Approach

In Chapter 7, we study fixed threshold algorithms for the disk spindown problem. Intuitively, we can think of a fixed threshold algorithm that spins down the disk after T seconds as a coarse predictor that predicts a "large idle time" if the idle time is greater than T seconds. We also describe our experiences with predictive disk spindown strategies; these predictive strategies are based on intuition developed from Chapters 3–5.

In Chapter 8, we consider the rent-to-buy framework in detail and develop a computationally efficient rent-to-buy algorithm whose expected cost for the tth resource use (i.e., the tth idle time at disk) converges to optimal for any bounded probability distribution on the resource use times, under the assumption that the resource use times are independently drawn from an unknown probability distribution. In Chapter 9, we formalize our notion of adaptive spindown. We describe exactly how a rent-to-buy algorithm will be used for the disk spindown problem, and present the results of simulating our theoretically optimal rent-to-buy algorithm for the disk spindown problem.

Chapter 7

Fixed Threshold and Predictive Disk Spindown Policies

Minimizing power consumption is important for mobile computers, and disks consume a significant portion of system-wide power. As described in [DKM], disk densities are increasing, making it possible to carry more data. Even though disk densities have increased, the power used by the largest disks has stayed about the same, around 1 Watt for an idle spinning disk. With more efficient management of the other components of the system (e.g., the CPU and the display), the overall system power cost is dropping; the result is that the relative amount of power consumed by the disk sub-system on notebook computers has increased from 9% to 31%. With the exception of the smallest and lightest computers, such as the Hewlett-Packard Omnibook [Hewa], the trend seems to be to carry a larger disk with the same mass rather than a smaller disk with the same number of bytes.

Proper disk management can improve battery life. If the disk drive is used with any frequency, it will have a significant impact on the length of time the computer can operate on a single battery charge. Measurements of power consumed by notebook computers as described in [DKM] suggest that battery life for the Dell 320 could be improved 20–31%, the amount that would be saved if the disk were off all the time. Put another way, a battery that lasts 5 hours could last from 6 to 6.5 hours instead. Since there is a large difference in power consumption between a disk that is spinning and one that is not, systems try to keep the disk spinning only when it must. The system must trade off between the power that can be saved by spinning the disk down quickly after each access and the impact on response time from spinning it up again too often.

To understand this tradeoff, we use trace-driven simulations to evaluate different disk spindown policies for reducing power consumption. In this chapter, we consider threshold policies which are practical to implement, offline algorithms that are optimal in terms of power consumption, and predictive online algorithms that take past history into account. We find that threshold policies that spindown the disk after 1–10 seconds come close to the power consumption of the optimal offline algorithm, which reduces the power consumption using manufacturers' recommended thresholds by about half. However, in some cases the threshold algorithms substantially increase the delays incurred by the user. These delays can be avoided if access times could be predicted accurately enough.

In Section 7.1 we describe a model of the energy states of the disk. In Section 7.2 we discuss various spindown policies. In Section 7.3 we describe the input traces and simulator

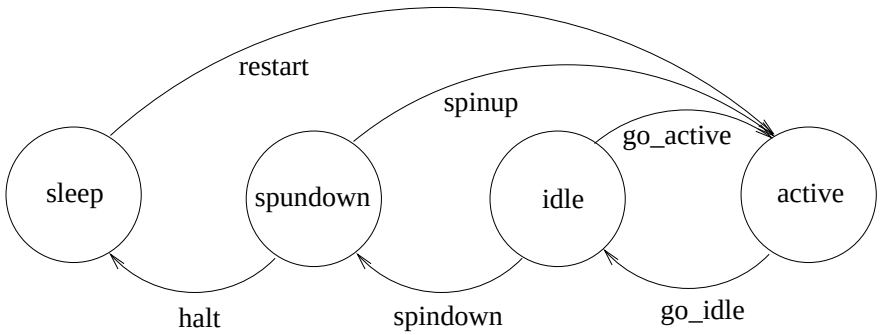


Figure 7.1: Disk state transition diagram.

used in our experiments, and in Section 7.4 we report the results of our simulations. We present our experiences with predictive disk spindown in Section 7.5. Other related issues are discussed in Section 7.6.

7.1 Disk Energy States

As mentioned before, disk drive manufacturers have developed disks with multiple energy states, especially targeted to the portable computer market. Figure 7.1 shows the energy states and transitions of a typical disk. (The "spundown" state is sometimes also called the "standby" state; some manufacturers refer to the "spundown" state as the "sleep" state, and the "sleep" state as the "off" state.)

The disk has to be in the active state to read and write data. In the idle state the disk is spinning, but the heads are parked. In the spundown state, the disk platter is not spinning. The disk has to be powered up from the sleep state.

Typically, the disk consumes a very small amount of power in the spundown state, and little or no energy in the sleep state. Table 7.1 lists the main characteristics of two disk drives for mobile computers, the Hewlett-Packard Kittyhawk C3014A [Pac] and the Quantum Go•Drive 120 [Qua]. (The quantity T_d in Table 7.1 is explained in Section 7.2.1.)

7.2 Policies

In this chapter, we investigate two types of algorithms for spinning a disk up and down: *offline*, which can use future knowledge, and *online*, which can use only past behavior. Offline algorithms are useful as a baseline for comparing different online algorithms, and for identifying where there is room for potential improvement. Online algorithms are implementable. Typically mobile computers spindown their disk based on a simple heuristic; for instance, when it has not been accessed in a predetermined period of time (such as 5 minutes). They spinup the disk when the first access after a spindown occurs.

7.2.1 Offline Policies

Spinup and spindown policies should minimize both power consumption and response time. Power and time are not always optimized by the same policy. It is easy to see that the optimal policy with respect to response time is not necessarily optimal with respect to

Characteristic	HP Kittyhawk C3014A	Quantum Go•Drive 120
Capacity (Mbytes)	40	120
Power consumed, active, (W)	1.50	1.65
Power consumed, idle, (W)	0.63	1.00
Power consumed, spundown, (W)	0.28	0.20
Power consumed, sleep, (W)	0.02	0.20
Power consumed, spinup (W)	2.17	5.50
Normal time to spinup (s)	1.10	2.50
Normal time to spindown (s)	0.55	6.00
Avg time to read 1 Kbyte (ms)	22.5	26.7
Break-even interarrival time T_d (s)	5.0	14.9

Table 7.1: Disk characteristics of the Kittyhawk C3014A and Quantum Go•Drive 120. The Kittyhawk has less capacity than the Go•Drive, but it has significantly lower operating costs, especially the power drawn during disk spinup and the average spinup duration. As a result, the break-even point for the Kittyhawk is about a fourth that of the Go•Drive, making a short spindown threshold much more important for the Kittyhawk. Also, the Kittyhawk spins down in a half a second, while the Go•Drive takes "< 6s" [Qua].

power consumption. Leaving the disk spinning all the time will produce the minimal impact on response time, but will waste power if the disk isn't accessed for long periods of time. Likewise, the optimal policy with respect to power may result in a delay when a new request stalls waiting for the disk to spinup.

An offline policy for spinning down the disk is based on the relative costs of spinning or starting it up. The offline policy will spindown the disk if the inter-arrival time is such that spinning down the disk and spinning it back up would consume lesser energy than keeping the disk spinning. It is easy to verify that, given a disk, there is a breakeven interarrival time T_d depending only on the disk characteristics, such that the offline policy will spindown the disk immediately if the interarrival time is greater than T_d. There are, of course, complications beyond this simple threshold; for instance as seen from Figure 7.1 and Table 7.1, a disk usually has multiple states that consume decreasing amounts of power but from which it is increasingly costly (in time and power) to return to the active state. Table 7.1 lists the values for T_d for the Kittyhawk and Go•Drive disks.

The time to spinup the disk once a new request arrives has a substantial impact on response time. An online algorithm that spins up the disk when a request arrives if the disk is spun down will cause the request to wait until the disk is ready, typically at least 1–2 seconds. This latency is up to a couple of orders of magnitude greater than normal disk access times, and should be avoided whenever possible. The high spinup overhead is the reason why typical thresholds for spinning down a hard disk are often on the order of several minutes even if T_d is just a few seconds: if the disk has not been accessed for several minutes then the overhead of a couple of extra seconds before a new request can be serviced is neither unexpected nor unreasonable. In contrast to the online approach, an offline algorithm can not only spindown the disk when that would save power, it can spinup the disk again just in time for the next request to arrive.

7.2.2 Threshold-based Policies

Threshold-based policies are the standard timeout-based algorithms used in most present systems. If the disk is not accessed within a fixed period of time it is spun down. That timeout value may vary depending on environmental characteristics (e.g., running off battery rather than A/C power) or input from the user, but is normally not modified dynamically by the system. The disk is spun up again upon the next access, and the request must wait for spinup to complete.

7.2.3 Predictive Policies

The predictive policies store historical information and interpret it to predict the interarrival time at disk. There are many possible heuristics for interpreting this data. For instance, Wilkes hypothesized that it would be effective to use a weighted average of a few previous interarrival times to decide when to spindown the disk on a mobile computer. He noted as well that if inactive intervals were of roughly fixed duration, the disk could be spun up in advance of the expected time of the next operation [Wil]. If access patterns are not so consistent, however, these techniques may not prove to be helpful. We use Markov model-based predictors, and discuss more about our predictive policies in Section 7.5

7.2.4 Taxonomy

Here we describe a taxonomy of disk spindown policies, considering both the policy used to decide when to spindown the disk and the one used to spin it up again. The naming scheme indicates the most salient feature of the particular algorithm; the first part of the name denotes the spindown policy, while the second part denotes the spinup policy.

OPTIMAL_OPTIMAL. This is the offline algorithm described in Section 7.2.1, which uses future knowledge to spindown the disk and to spin it up again *prior* to the next access. It provides the lowest power consumption among policies that have minimum response time. Other offline algorithms with slightly lower power consumption may exist, but they will suffer increased response times.

OPTIMAL_DEMAND. An alternative offline approach is to assume future knowledge of access times when deciding whether to spin down the disk but to delay the first request upon spinup. This algorithm will consume about the same amount of power as OPTIMAL_OPTIMAL, but will have poorer response time. This algorithm is relevant because an online algorithm may be better at predicting that the next request will occur more than T_d seconds in the future than predicting exactly when the request will occur; i.e., predicting the correct time to spindown the disk may be easier than predicting when to spin it up again.

THRESHOLD_DEMAND. The disk is spun down after a fixed period of inactivity and is spun up upon the next access. This is the policy used on most systems at present.

THRESHOLD_OPTIMAL. The disk is spun down after a fixed period of inactivity but is spun up just before the next access. If the next access occurs too soon (there is not enough time to spinup the disk before the access) then the access will be delayed. This algorithm is primarily for purposes of comparison and completeness.

PREDICTIVE_DEMAND. Spindown is based on a heuristic that uses information about previous accesses to determine when to spindown the disk. Spinup is performed upon the next access.

PREDICTIVE_PREDICTIVE. Spindown uses the same heuristic as PREDICTIVE_DEMAND, and spinup is based on a predictive heuristic as well.

7.3 Methodology

7.3.1 Traces

To evaluate the effect of the disk spindown policy on power consumption and response time, we used traces from two execution environments: an Apple Macintosh Powerbook Duo 230 and a Hewlett-Packard 9000/845 personal workstation running HP-UX.

The Powerbook traces were gathered at MITL. There were two traces of approximately two hours of activity and one trace of approximately four hours of activity. One of the two-hour traces has characteristics very similar to the four-hour trace, while the other shows more constant use of the disk. In this chapter we report results of simulating the four-hour trace, during which time mostly Microsoft Word, an editor, and Eudora, an electronic mail application, were running.

Disk management on the Macintosh is unusual in several respects. First, its cache behavior is dependent on the size of the cache: a larger disk cache not only increases the number of blocks that can be cached but also increases the maximum size of any given read or write that can be cached. Even with a cache size of 256 Kbytes, the maximum transfer that can be cached is only 8176 bytes [App]. Second, writes that are cached in the buffer cache are later passed to the disk in 512-byte units, which can degrade performance relative to transferring larger files as a unit. Third, a Macintosh may be configured with a "RAM disk" that behaves like a magnetic disk drive but is stored in DRAM. On the Powerbook Duo the RAM disk is not persistent, so it is useful only for storage of temporary files, other noncritical files that can be copied to disk later, or copies of read-only files such as the System folder. The RAM disk thus allows users to get around the deficiencies of the buffer cache, but only for a specific subset of files.

Our Powerbook trace records reflected access at the file-system level (i.e., above the disk cache). Rather than simulating the Macintosh buffer cache as it is implemented, we simulated a simple LRU write-through buffer cache with each 1-Kbyte block handled separately and no maximum per-file limit. The Macintosh enforces a minimum cache size of 32 Kbytes; we varied the cache size from having no cache at all to a maximum of 1 Mbyte. Because a relatively large cache is essential to eliminating enough disk accesses to make spinning down the disk worthwhile [LKH], in this chapter we report results for the 1-Mbyte cache. Also, in our Powerbook traces, about 2% of accesses went to files on the RAM disk, and we ignored these accesses in the simulator. More experienced Powerbook users might have different access patterns, with more accesses to a RAM disk, but in fact one may consider the 1-Mbyte disk cache to be equivalent to a RAM disk of the same size.

In addition, we used traces from an HP-UX workstation, documented by Ruemmler and Wilkes [RuWa]. We used the HP-UX traces for a couple of reasons: first, because there are a number of UNIX-based mobile platforms available, so a UNIX trace might be indicative of actual mobile usage patterns; and second, because UNIX does the sort of aggressive

	Powerbook	HP-UX
Duration	4.1 hours	63 days
Mean interarrival time (s)	0.4	13.1
Standard deviation of interarrival time (s)	1.9	130.3
Maximum interarrival time (s)	269.2	1770.4

Table 7.2: Summary of trace characteristics. An important distinction between the Powerbook and HP-UX traces is that the statistics for the Powerbook trace report the interarrival times as seen by the buffer cache while the ones for the HP-UX trace are as seen by the disk.

caching that is essential for eliminating disk accesses. The HP-UX traces are at the disk level; they represent requests from the HP-UX buffer cache to the disk. As a result, we did not simulate a buffer cache for these traces.

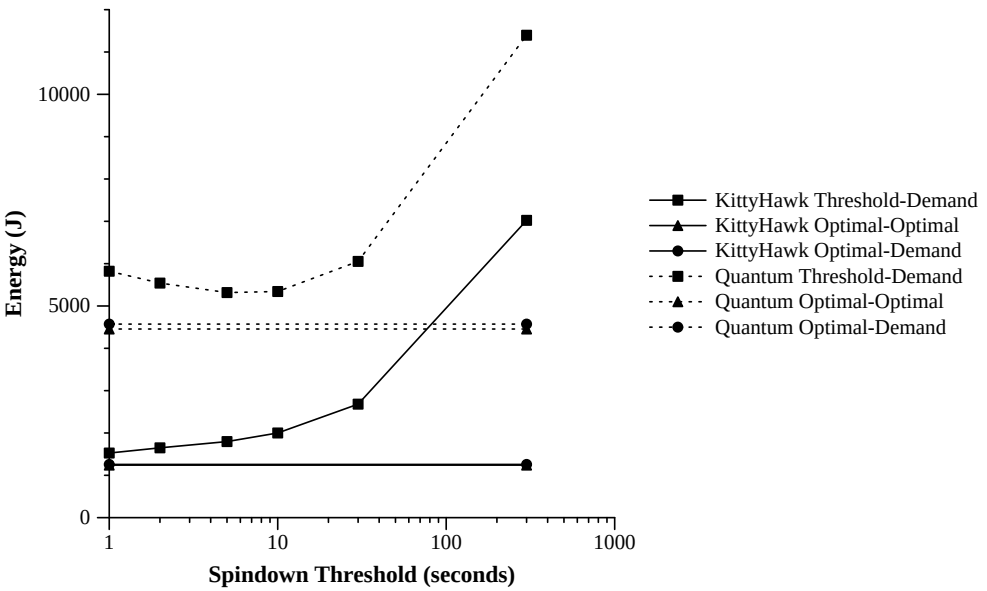
We believe the HP-UX traces should be representative of a mobile environment because the sorts of activities mobile users perform—such as word processing, electronic mail, and spreadsheets—are the same activities as the user in the HP-UX trace would perform in the office. Nevertheless, the HP-UX and Powerbook traces do vary considerably in some respects. Most notably, the HP-UX trace is over a prolonged period of time (about 2 months), with about the same number of accesses as the Powerbook trace had over four hours spread out instead over a week's time. (The simulations in [DKM] were done using only the first week's worth of data; our results in this chapter use the whole trace. The results are qualitatively similar; quantitatively, they are away by roughly a factor of 8–10, suggesting that the distribution of inter-arrival times is similar throughout the trace.) Table 7.2 summarizes some characteristics of the two traces.

7.3.2 Simulator

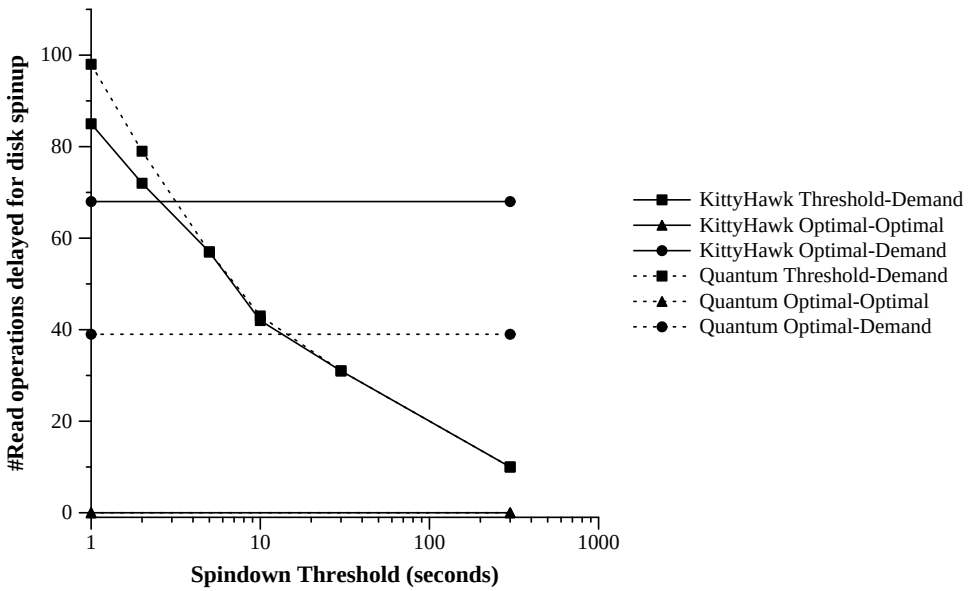
The simulator is a general storage management simulator that models three levels of a storage hierarchy, nominally DRAM, flash memory, and magnetic disk; for these experiments the size of flash was always set to 0. For the HP-UX trace, the size of the DRAM cache was also set to 0, as described above, so all disk requests from the original trace resulted in disk requests in the simulations.

Each disk is represented by a file specifying a number of parameters, one for each state in which the disk might be (reading, writing, active, idling, spundown, or sleeping), and one for each state transition. The configuration files specify how long the disk stays in a given state or transition and how much power, in Watts, is consumed during that time. Several of these parameters are shown in Table 7.1.

We made a number of simplifying assumptions in the simulator. First, a disk access is assumed to take the average time for seek and rotational latency, unless it involves a disk block with a numerical identifier within ϵ of the previous block accessed. Second, all operations and state transitions are assumed to take the average or "typical" time specified by the manufacturer, if one is specified, or else the maximum time. While these assumptions may affect the specific values of energy and response time produced by the simulator, we do not believe they affect the relative differences in energy consumption and response time from using different spindown policies.

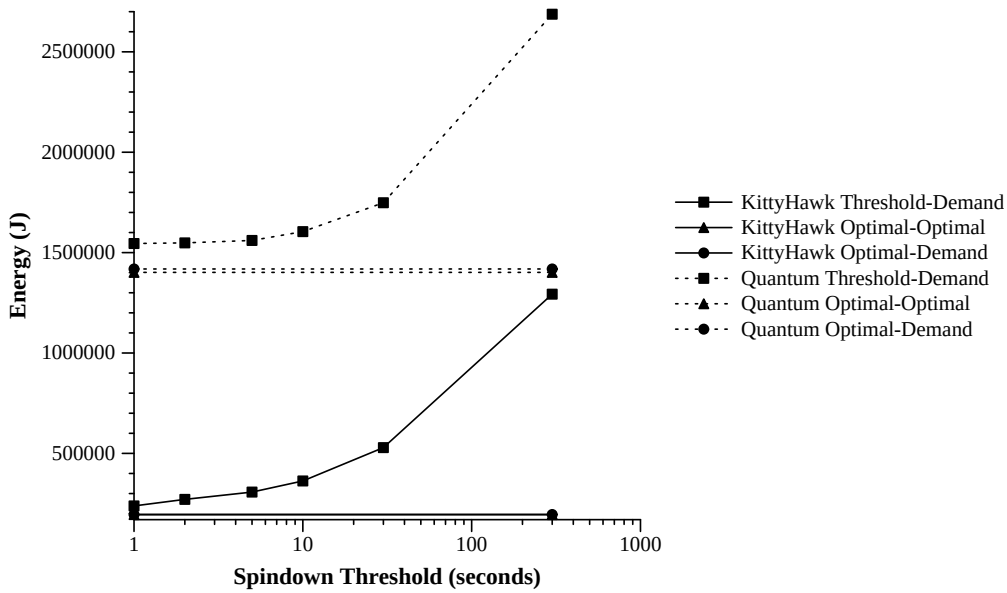


(a) Energy consumption as a function of spindown time.

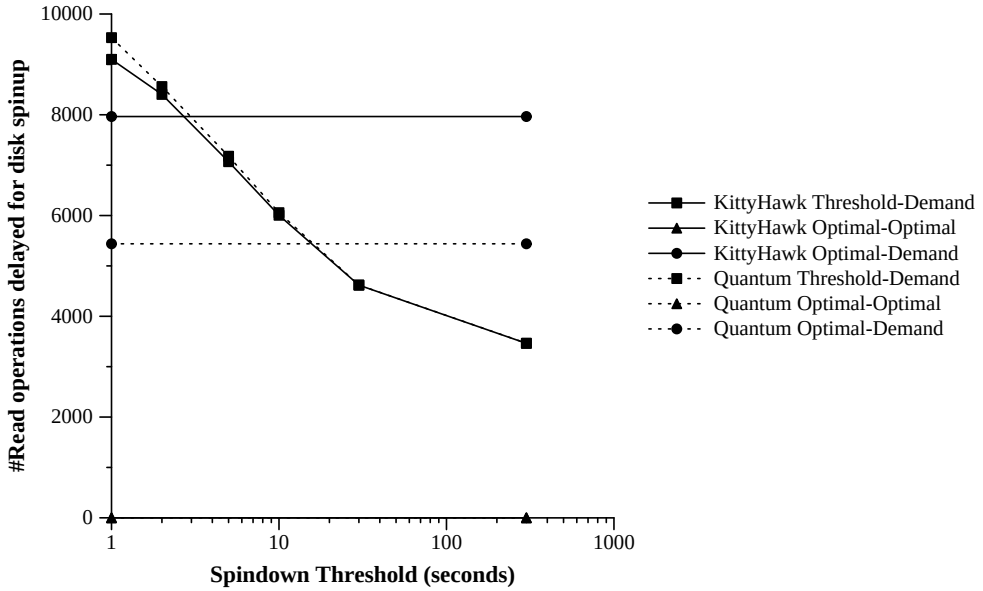


(b) Delays due to spinup on read accesses, as a function of spindown time.

Figure 7.2: Results of simulating the 4-hour Powerbook trace on the Kittyhawk and Go•Drive 120 disks.



(a) Energy consumption as a function of spindown time.



(b) Delays due to spinup on read accesses, as a function of spindown time.

Figure 7.3: Results of simulating the HP-UX trace on the Kittyhawk and Go•Drive 120 disks.

7.4 Results

We simulated a number of threshold-based policies as well as the OPTIMAL_OPTIMAL and OPTIMAL_DEMAND policies, running on both the Powerbook and HP-UX traces, and using both the Kittyhawk and Go•Drive specifications. (Results of using predictive algorithms are discussed in Section 7.5.) In this chapter we consider two metrics:

Energy Consumption The energy (in Joules) consumed by the disk over the course of the simulation.

Read-Spinup Delays The number of times a read operation was delayed to spinup the disk.

Another metric for the impact on read response time might be the average response time across all reads, but that metric is less satisfying: it considers average delay but not the great discrepancy between operations that have no spinup delay and those that are delayed. In fact, the actual delay from spinup varies from about 1 second on the Kittyhawk to 2.5 seconds on the Go•Drive, so the penalty from these undesirable spinup delays is much greater for the Go•Drive. A third possible metric is the number of operations delayed. The metrics of number of reads delayed and number of operations delayed are very close; intuitively, if writes (including synchronous writes) can be decoupled from disk latency with a small amount of nonvolatile memory [BAD, RuWa], then the number of writes delayed can be ignored.

Figure 7.2 shows the energy consumption and read-spinup delays for the Powerbook traces, with a 1-Mbyte cache, and Figure 7.3 shows the same for the HP-UX traces (which has an implicit buffer cache, as discussed above). Both figures show, for both types of disk, several THRESHOLD_DEMAND policies, OPTIMAL_OPTIMAL, and OPTIMAL_DEMAND. For the threshold-based policies, the disk always goes to the idle state after two seconds if it has not already spundown by then, and always goes from the "spundown" state to the "sleep" state immediately. On the Kittyhawk there is marginal overhead in going from "sleep" to "active" relative to going from "spundown" to "active," and on the Go•Drive the "spundown" and "sleep" states are identical.

The most important conclusions one may reach from these figures are:

- The offline OPTIMAL_OPTIMAL algorithm can reduce disk power consumption by 30–60%, compared to the fixed threshold suggested by manufacturers, without adversely affecting response time. (This compares OPTIMAL_OPTIMAL to the 5-second spindown of the Kittyhawk and the 5-minute spindown of the Go•Drive 120.)

- Online THRESHOLD_DEMAND algorithms with shorter than recommended thresholds approach the power consumption of OPTIMAL_OPTIMAL but may increase the number of read-spinup delays substantially.

- The best compromise between power consumption and response time is workload-dependent. For the HP-UX trace on the Kittyhawk, a spindown threshold of 1s consumes 23% less power than the recommended threshold of 5s, and increases delays by 19%, a fair tradeoff. The Powerbook trace on the same hardware shows a 15% improvement in energy with the 1-second threshold but a 50% increase in delays.

- Lastly, the characteristics of the disk make an enormous difference in the appropriateness of an aggressive spindown policy. The high latency to spindown and spinup the Go•Drive 120, compared to the Kittyhawk, results in a higher value for T_d and, for the Powerbook trace, minimal power consumption for a threshold of 5s rather than 1s for the Kittyhawk.

We discuss each type of algorithm in turn, as well as the impact of disk characteristics.

7.4.1 Offline Algorithms

With future knowledge of disk activity, one can both reduce energy consumption and delays due to disk spinup. OPTIMAL_OPTIMAL uses 64% of the energy consumed by the 5-second THRESHOLD_DEMAND policy for the HP-UX trace running on the Kittyhawk; it uses 52% of the energy consumed by the 5-minute policy running on the Go•Drive. For the Powerbook trace, OPTIMAL_OPTIMAL used 69% and 39% of the energy of the recommended thresholds for the Kittyhawk and Go•Drive, respectively. In each case, because the disk was always spinning at the time of the next request, response time would improve as well.

OPTIMAL_DEMAND considers the hypothetical case where one could predict the future well enough to spindown immediately if that would save power, but would not be able to predict the time of the next access precisely. It uses about the same amount of energy as OPTIMAL_OPTIMAL but has a much larger number of read-spinup delays than the 5-minute THRESHOLD_DEMAND policy, since many more read operations result in the disk spinning up. The number of read-spinup delays incurred by this algorithm is a lower bound on the number of delays that a THRESHOLD_DEMAND algorithm with a spindown threshold less than T_d would incur, and an upper bound on the number that a THRESHOLD_DEMAND algorithm with a spindown threshold greater than T_d would incur.

7.4.2 Threshold-Demand Algorithms

The set of THRESHOLD_DEMAND algorithms reported above demonstrate the tradeoffs between energy and delay that arise with any simple threshold-based technique. The differences between the Powerbook and HP-UX traces show how important the workload is in this regard: for the HP-UX trace on the Go•Drive disk, a 1-second threshold performed best out of this class of algorithms, reducing power consumption 45% (within 7% of optimal) compared to the 5-minute threshold, while for the Powerbook traces the 5-second and 10-second thresholds performed comparably, reducing power by about 53%. This is not surprising given the difference in mean interarrival times described in Section 7.3.1.

Regardless of the subtle differences in energy consumption between relatively short thresholds of 1, 2, or 10 seconds, it is clear that any of these short thresholds is much more energy efficient than the manufacturers' commonly recommended spindown threshold of 5 minutes. However, the shorter thresholds introduce more spinup delays. As a result, the most energy efficient threshold may not be the most desirable one. For example, with the Powerbook trace running on the Kittyhawk, moving from a threshold of 1s to 5s increases energy consumption by 18% but reduces the read spinup delays by 33%.

7.4.3 The Impact of Disk Characteristics

Figures 7.2 and 7.3 show the simulation results of using the Kittyhawk and the Go•Drive drives using the same sets of traces and spindown parameters. The lower operating costs

Bucket	Range (seconds)
0	[0.00, 0.10)
1	[0.10, 0.25)
2	[0.25, 0.50)
3	[0.50, 1.00)
4	[1.00, 2.00)
5	[2.00, 3.00)
6	[3.00, 5.00)
7	[5.00, 100.00)
8	[100.00, ∞)

Table 7.3: One of the definitions for buckets used to study predictive spindown.

for the Kittyhawk result in less power consumed for the same policies, and the faster and less power-intensive spinup for the Kittyhawk makes it more feasible to quickly spindown the disk.

Since the Go•Drive consumes more power than the Kittyhawk when operating, and takes much longer to spinup and spindown, its overall power consumption is far greater than for the Kittyhawk on the same workload. The impact of varying the spindown threshold is less than for the Kittyhawk as well. Of course, operations on larger disk drives are expected to consume more power than those on small ones, and spinning them up is especially costly—both in terms of power and response time—by comparison.

7.5 Predictive Algorithms[1]

The gap between the threshold-based policies and the offline policies suggests that predictive algorithms can improve upon existing techniques. We experimented with heuristics for predicting when to spindown based on past history rather than just the time since the last access. The goal of the predictive approach is to predict interarrival times between disk accesses so that the disk is spundown immediately when the next disk access is likely to be far enough in the future to make spinning down the disk worthwhile (i.e., if the idle time at disk is predicted to be greater than T_d).

Our heuristics for predictive disk spindown were suggested from our observations from Chapter 5 of the intimate relationship between data compression and prediction. Our approach was to use a Markov model-based approach to anticipate when the next access will arrive at disk. (Markov models are defined in Section 5.2.2.)

We implemented the Markov model for predictive spindown by mapping ranges of inter-arrival times into *buckets*, and defining the buckets to be our alphabet; in other words, we discretized our domain. Table 7.3 gives an example of the buckets we used in one of our simulations. An inter-arrival time naturally maps on to a bucket. We build an mth order Markov model on the alphabet defined by the buckets. (We studied Markov models of order 1, 2, and 3.) The past m bucket values as defined by the past m inter-arrival times uniquely define the *current state* s of the Markov model. Like in the prefetching problem

[1]Some of the material described in this section is the subject of a pending patent application.

Bucket	0	1	2	3	4	5	6	7	8
0	4905	443	193	135	142	95	132	32	0
1	416	131	51	38	55	21	33	13	0
2	178	49	19	29	48	25	33	6	0
3	141	38	29	19	36	18	45	3	0
4	176	36	41	40	33	16	56	8	0
5	93	17	12	25	19	6	76	14	0
6	142	34	38	38	61	77	1140	47	0
7	26	10	5	5	11	4	62	35	0
8	0	0	0	0	0	0	0	0	0

Table 7.4: Transition table corresponding to the buckets in Table 7.3, obtained by using a first order Markov predictor on the Powerbook trace.

as described in Chapter 5, we predict the next interarrival time to lie in bucket j, where the transition labeled j has the maximum probability (or count) out of the current state s. (Recall that the Markov predictor is updated by incrementing the count of a traversed transition by one.) If the predicted bucket corresponds to a range of interarrival times that is "sufficiently large," we spindown the disk immediately. Table 7.4 gives a snapshot of a first order Markov model obtained by running the Powerbook trace using the bucket definitions from Table 7.3. For our experiments, we defined a predicted interarrival time range to be "sufficiently large" if the minimum of the range was greater than T_d (an aggressive policy). We also looked at conservative strategies, where we spin down the disk only if the minimum of the predicted interarrival range was 2–3 times T_d.

The bucket definitions were chosen based on the level of granularity we desire for our prediction. In other words, we could have had just two buckets, $[0, T_d)$, and $[T_d, \infty)$; the rationale behind having more buckets is that the extra precision can increase the likelihood of making an accurate prediction. We experimented with different granularity of buckets; in general, the specific number of buckets and the ranges for each bucket must be tuned to a particular system.

In PREDICTIVE_DEMAND, we use the predictions obtained as described above to spindown the disk. We spinup the disk upon the next access. One problem that must be addressed is what happens if the prediction is incorrect. If the prediction indicates that the disk should be spundown, and an access occurs within T_d seconds (an *eager* prediction), the disk is spunup again. If the prediction causes the disk to remain spinning, yet no access occurs (a *lazy* prediction), leaving the disk spinning indefinitely might consume too much power, rendering the whole procedure useless. To take care of lazy predictions, we augment PREDICTIVE_DEMAND by using the threshold from the THRESHOLD_DEMAND algorithm: if the disk is not spundown by virtue of a prediction, and T seconds elapse without an access (where T is the threshold used by the THRESHOLD_DEMAND algorithm), the disk is spundown.

In PREDICTIVE_PREDICTIVE, we estimate the inter-arrival time to be the average value of the range given by the bucket predicted by the Markov predictor. (We use finer grained buckets in this case.) We spin the disk up based on the estimated value or upon demand, whichever is earlier.

7.5.1 Experiences with Using PREDICTIVE_DEMAND

We observed some interesting results by simulating PREDICTIVE_DEMAND. When the Powerbook trace was simulated with *no buffer cache*, PREDICTIVE_DEMAND saved 15% of the power consumed without adverse affects (on the number of reads delayed). In this case, the energy consumption of PREDICTIVE_DEMAND was roughly halfway between the optimal energy consumption and the best THRESHOLD_DEMAND policy. However, by using a moderate size buffer cache (i.e., upto 1-MByte cache), all the energy values dropped drastically. (The performance numbers reported in Section 7.4 assumed a buffer cache of 1-MByte for the Powerbook traces.) With a buffer cache, the energy consumption and response time performance of the predictive algorithm was generally slightly worse than the corresponding THRESHOLD_DEMAND algorithm. That is, there were almost no eager predictions and more lazy predictions when the buffer cache was present.

The predictability of the Markov predictor by itself was very good. That is, the fraction of times our model correctly predicted the bucket in which the interarrival time will fall was high; roughly 70%. As illustrated by Table 7.4, some of these predictions were expected: if the previous interarrival time was small (corresponding to bucket 0), then the next interarrival time is also expected to be small. However a significant fraction of the predictions were non-trivial; e.g., Table 7.4 suggests that if the previous interarrival time fell in bucket 6, then the current interarrival time will also be in bucket 6. Using the bucket definitions from Table 7.3, this implies that if the last access came between 5 and 100 seconds after the preceding access, the next access is also likely to come in that range; so if the disk is a Kittyhawk, then based on the T_d value from Table 7.1, we should spindown the disk immediately.

We did not experiment extensively with PREDICTIVE_PREDICTIVE. Intuitively, PREDICTIVE_PREDICTIVE requires a more accurate prediction as compared to PREDICTIVE_DEMAND. Our impression from the study is that the buffer cache tends to increase the entropy of interarrival times as seen by the disk; without a cache, access times are more predictable. Intuitively, having a large number of consecutive hits in cache increases the inter-arrival time at disk. When a buffer cache is present, the predictor sees this effect of the cache hits in the inter-arrival time. Choosing the buckets more efficiently, or developing strategies that better understand the effect of the buffer cache on the inter-arrival times at disk and using these strategies with our schemes might give better performance.

7.6 Discussion

In this chapter, we have studied our results of simulating various threshold-based and predictive techniques for minimizing the power consumption of hard disks on mobile computers by spinning the disk only when necessary. Our offline policy, OPTIMAL_OPTIMAL, demonstrates that significant power savings are possible: there is a potential to reduce disk power consumption by 30–60%, compared to the fixed threshold suggested by manufacturers. Our THRESHOLD_DEMAND policy demonstrates that it is possible for practical policies to get power savings close to that of the offline policy. Threshold policies that spindown the disk after 1–10 seconds come within 7–23% of the offline policy, and consume only 47–85% of the energy consumed by manufacturers' recommended thresholds (5 seconds for the Kittyhawk, and 5 minutes for the Go•Drive 120). In other words, the disk access patterns suggest that an aggressive spindown policy is better with respect to conserving energy.

Threshold policies with a small spindown threshold save energy but degrade response time performance. We do not yet have enough experience to know what the best metric of degradation might be; in this chapter we have compared algorithms based on the number of times a read request is delayed, but as noted in Section 7.4, other metrics are possible. In Chapter 9, we consider the related metric of total number of operations delayed.

As mentioned in Chapter 6, independently to our work, Li et al. have also investigated the issue of disk drive power management [LKH]. They used trace-driven simulation to look at a THRESHOLD_DEMAND policy for disk spin-control, and studied buffer cache parameters. While we considered multiple algorithms for determining when to spindown the disk (offline optimal, THRESHOLD_DEMAND, and predictive), they focused on THRESHOLD_DEMAND. They did a more detailed analysis of the impact of the buffer cache on power consumption and performance. Their basic conclusion regarding THRESHOLD_DEMAND algorithms is the same as ours: short timeouts on the order of a few seconds greatly reduce power consumption by the disk and do not significantly degrade performance.

As mentioned above in Section 7.2.3, Wilkes proposed a predictive algorithm for disk management [Wil]. He suggested adjusting spindown timeouts based on a weighted average of recent interarrival times. Picking the weights may be a difficult task; we attempted to implement this strategy but were unable to out-perform THRESHOLD_DEMAND with the particular sets of weights we tried. To the best of our knowledge, no other implementation of this strategy has been attempted.

The impact of hardware design on software parameters is significant. The Kittyhawk was designed to make the transition between the active and idle states much less prohibitive than the Go•Drive or comparable disk drives. As a result, a shorter threshold is appropriate for the Kittyhawk than for these other disks, although even the Kittyhawk is susceptible to workloads that will result in too much overhead to make frequent spindowns feasible. A better approach than very short spindown timeouts may be better predictive algorithms that exploit past history. An alternative, *adaptive spindown* (defined in Chapter 9), is less aggressive, and effectively exploits past interaccess times in a different way to make better spindown decisions. We study the rent-to-buy framework and adaptive disk spindown via rent-to-buy in the next two chapters.

Chapter 8

Optimal Rent-to-Buy in Probabilistic Environments

In Chapter 7, we studied fixed-threshold algorithms, and some predictive techniques for the disk spindown problem. We concluded that adaptively adjusting the spindown threshold based on our experience from previous inter-arrival times might yield even better results than the improvements observed via aggressive fixed threshold policies. In this chapter, we study the *rent-to-buy problem*, and develop a provably optimal and computationally efficient algorithm for the rent-to-buy problem in probabilistic environments. The rent-to-buy framework models not only the disk spindown problem, but other interesting systems problems as well, specifically, thread blocking decisions during lock acquisition in multiprocessor applications [KLM], and virtual circuit holding times in IP-over-ATM networks [KLP, SaK]. The rent-to-buy model, as we will see in this chapter and in Chapter 9, lies at the heart of our strategy for adaptive disk spindown.

8.1 Online Rent-to-Buy

The *single rent-to-buy decision* problem can be described as follows: we need a resource for an unknown amount of time, and we have the option to rent it for \$1 per unit time, or to buy it once and for all for \$$c$. How long should we rent the resource before buying it? The best algorithm with full prior knowledge of how long the resource will be needed (an offline algorithm) buys the resource immediately if the resource will be needed for at least c time units and rent otherwise. An online algorithm (i.e., one without *a priori* knowledge of how long the resource will be needed) that rents the resource for c units of time and then buys it incurs a cost of at most 2 times the cost of the best offline algorithm. This competitive factor[1] of 2 is the best possible (for deterministic algorithms) in the worst case [KMM]. If we know of a probability distribution on the time the resource is needed, we can usually find a rent-to-buy strategy whose expected cost is substantially less than that of the online algorithm that waits c time units before buying.

In this chapter, we are interested in the rent-to-buy problem described above with two important additional features motivated by practical applications. Many interesting systems problems can be modeled well by a *sequence* of single rent-to-buy problems. To solve the tth single rent-to-buy problem (or the tth *round*), the online algorithm can use

[1] Competitive analysis was defined in Chapter 3.

what it has learned from the previous $t-1$ rounds. (The online algorithm that waits for c time units before buying in each round is still within a factor of 2 of the best possible.) We call this the *sequential rent-to-buy* problem, or just the *rent-to-buy* problem. In these real-life situations we can assume that the time for which the resource is needed in each round is drawn from a probability distribution. However, it is unreasonable to assume that the distribution is known a priori. We now describe three interesting problems modeled by a sequence of rent-to-buy decisions.

The disk spindown problem studied in Chapter 7 is our main motivating application to study the rent-to-buy problem. As mentioned in Chapter 6, the disk spindown scenario can be modeled by the rent-to-buy framework as follows. A round is the time between any two requests for data on the disk. For each round, we need to solve the disk spindown problem. Keeping the disk spinning is viewed as renting, since energy is continuously expended to keep the disk spinning. Spinning down the disk is viewed as a buy, since the energy to spindown the disk and spin it back up upon the next request is independent of the remaining amount of time until the next disk access. The cost of the increased latency in serving the next disk access can also be integrated into the cost of the buy, if the algorithm is given as an input the relative importance of conserving energy and responding quickly to disk accesses. (This is discussed in detail in Chapter 9, and forms an integral component of *adaptive spindown*.) Based on observations of disk access patterns in workstation environments [RuWa], the times between accesses to disk (which define the rounds) can be assumed to be generated by a probability distribution.

The *spin/block* problem, another interesting and important problem from multiprocessor applications, can be modeled by the rent-to-buy framework. The spin/block problem involves threads trying to acquire locks to protect access to shared data [KLM]. A round is defined by a thread requesting locked data and eventually acquiring the lock. In a round, the system can have the thread wait (or *spin*) until the lock is free, incurring a fixed cost per unit time for wasted processor cycles, or block and incur a higher context switch overhead. The spinning can thus be viewed as renting, and a block can be viewed as a buy. In this situation too, practical studies suggest that lock-waiting times can be assumed to obey some unknown but time-invariant probability distribution [KLM].

Deciding virtual circuit holding times in IP-over-ATM networks is another scenario modeled by the rent-to-buy framework [SaK]. When carrying Internet protocol (IP) traffic over an Asynchronous Transfer Mode (ATM) network, a virtual circuit is opened upon the arrival of an IP datagram, and the ATM adaptation layer has to decide how long to hold a virtual circuit open. There are many possible pricing policies for virtual circuit holding times. As described in [SaK, Section 5], in future ATM networks, it is expected that a large number of virtual circuits could be held open by paying a charge per unit time to keep the circuit open. Keeping the virtual circuit open can be thought of as a "rent" while closing it can be considered a "buy." The inter-arrival time of packets on a circuit (i.e., the resource use times in the rent-to-buy model) can be modeled as being drawn independently from a probability distribution [LPR, SaK].

An algorithm for the sequential rent-to-buy problem can be visualized in two ways. In any round, the algorithm can be thought of as making sequential binary decisions of "Should I buy now?" Alternatively, we can think of the algorithm as setting a threshold or *cutoff* on the cost it is willing to accrue before buying, and behaving according to the cutoff. These two views are trivially equivalent; we adopt the second for convenience. There are two important requirements of any good online algorithm for the rent-to-buy

problem: the algorithm should produce good cutoffs, and it should use minimal space and time to output its cutoffs. In this chapter we develop online algorithms for the rent-to-buy problem in probabilistic environments, assuming that the resource use times are independently randomly drawn from a fixed but unknown probability distribution.

The most straightforward solution to the problem [KMM] is to store all past resource use times, and use that cutoff b for the current round which would have had the lowest total cost had we used it in the past. Straightforward application of results of Vapnik [Vapb] implies that the expected rent-to-buy cost of this strategy converges to that of the best fixed cutoff. One can easily see that the cutoff b at any given time falls on (actually, near) one of the past resource use times; however, even taking this into account, this solution is computationally expensive. For the tth round, this solution would need space and time proportional to $O(t)$, and this is unacceptable in system environments.

In this chapter, we develop an algorithm L for the rent-to-buy problem which, for arbitrary probability distributions with support on $[0, M]$, converges to optimal; i.e., the cost of the algorithm converges to the cost of the best algorithm with full prior knowledge of the distribution. More importantly, for the tth round that lasts x_t time, the algorithm uses $O(c\sqrt{t})$ space, generates its cutoffs in $O(1)$ time, and uses $O((\min\{x_t, c\})\sqrt{t} + \log(ct))$ time to update its data structures. Alternatively, our algorithm can be adapted to work in a situation when the space it can use is limited. Presented with $O(s)$ space, our algorithm L_s uses $O(1)$ time to generate cutoffs, $O((\min\{x_t, c\})s + \log(cs))$ time to update its structures, and almost converges to optimal, being away from optimal by an additive factor of $O(\min\{M, c\}/s)$. The $O(x_t)$ component of the time used in updating the data structure can be done "on the fly" as the round is progressing. For example, in the disk spindown scenario, let the tth idle time at disk be $z < c$ seconds. Before the idle period starts, algorithm L_s outputs its recommended spindown threshold using $O(1)$ time, and updates its data structure in $O(zs + \log(ct))$ time. The updates corresponding to the "zs" term can be done while the disk is waiting for the next access.

Most practical situations are well-modeled by bounded distributions. For example, in the disk spindown scenario, any reasonable algorithm will spin down the disk after a few minutes (say, 30 minutes) since the last access. Therefore, all idle times at disk greater than 30 minutes are practically equivalent, and can be assumed to be 30 minutes without loss of generality, resulting in a distribution with bounded support.

In Section 8.2, we describe our main analytical results. We present algorithm A_ϵ in Section 8.3; algorithm A_ϵ lies at the heart of our optimal rent-to-buy algorithms, L and L_s. We analyze algorithm A_ϵ for space used, computational time, and convergence rate in Section 8.4. We describe how algorithm A_ϵ can be used to get algorithms L, L_s in Section 8.5. Other related issues are discussed in Section 8.6.

8.2 Definitions and Main Analytical Results

We denote the reals by $\mathbb{R}$, the nonnegative reals by $\mathbb{R}^+$, and the positive integers by $\mathbb{N}$. An *online rent-to-buy algorithm* is given the relative cost $c \geq 1$ of buying. It works in rounds, where in the tth round, it first formulates a cutoff on the amount of time it will wait before buying, and then gets the tth resource use time. A rent-to-buy algorithm defines a mapping from $\cup_{n\in\mathbb{N}}(\mathbb{R}^+)^n$ (the past resource use times) to $\mathbb{R}^+$ (the cutoff generated). In other words, $A(x_1, x_2, \ldots, x_t)$ is the cutoff generated by algorithm A in the $(t+1)$st round, when the previous resource use times were $x_1,\ x_2, \ldots, x_t$. If the resource use time in any

round is x, then the cost of choosing cutoff b is

$$\text{cost}_c(x,b) = \begin{cases} x & \text{if } x \leq b \\ b+c & \text{otherwise.} \end{cases}$$

For the disk spindown problem, the resource use time in round t corresponds to the tth idle time at disk, and a cutoff is a spindown threshold.

Our first main result is an algorithm L that approaches optimal and is efficient in terms of the space and time it uses.

Theorem 8.1 *For any $c > 1$, $M > 1$, there is a rent-to-buy algorithm L that on round t with resource use time x_t,*

- *uses $O(c\sqrt{t})$ space*
- *outputs its choice of cutoff at the beginning of the round using $O(1)$ time, and updates its data structures in $O((\min\{x_t, c\})\sqrt{t} + \log(ct))$ time, and*
- *incurs a cost that approaches optimal: there exists k such that for any distribution D on $[0, M]$, for all large enough $t \in \mathbb{N}$,*

$$\mathbf{E}_{\vec{x}\in D^t}(\text{cost}_c(x_t, L(x_1, ..., x_{t-1}))) \leq \inf_a \mathbf{E}_{z\in D}(\text{cost}_c(z,a)) + k\sqrt{\frac{\ln t}{t}}.$$

Note that in Theorem 8.1, the same k can used for any distribution with support on $[0, M]$. Further, the time and space bounds are independent of D as well. It is easy to adapt algorithm L to get algorithm L' that successively increases its estimate of M, and converges to optimal for any distribution. However, the convergence rate of algorithm L' would depend on the distribution.

In many practical situations, we would like to fix the amount of space and time used by our algorithm while converging approximately, rather than exactly, to optimal. Algorithm L_s, a restricted space version of Algorithm L, can be used in this scenario.

Theorem 8.2 *When presented with $s > k\ln^2(M+c)\ln\ln(M+c)$ bytes of space, where k is a constant independent of M and c, for the tth round, algorithm L_s outputs its choice of cutoff in $O(1)$ time, updates its data structures in $O((\min\{x_t, c\})s + \log(cs))$ time, and for any probability distribution D on $[0, M]$ for all large enough $t \in \mathbb{N}$, converges approximately to optimal:*

$$\mathbf{E}_{\vec{x}\in D^t}(\text{cost}_c(x_t, L_s(x_1, ..., x_{t-1}))) \leq \inf_a \mathbf{E}_{z\in D}(\text{cost}_c(z,a)) + O\left(\frac{\min\{c, M\}}{s}\right).$$

One obvious approach to attack the rent-to-buy problem in probabilistic environments is to learn the distribution on times for the rounds, calculate the optimal cutoff for the estimated distribution, and output that cutoff for each round. This is unacceptable from the computational standpoint. In our algorithms, we bypass the estimation of the distribution, directly estimating the efficacy of different cutoff points. The analysis is complicated, however, by the fact that there are infinitely many cutoff points to evaluate at any given time on the basis of a finite number of samples from the distribution. We show for the rent-to-buy problem that to get a good solution, it is sufficient to consider a small finite set

of possible cutoff points. The appropriate choice of this set depends on the distribution, and is done using the information gained in early rounds. We call this basic strategy that chooses the appropriate set of possible cutoffs and evaluates them to determine the best cutoff to use in any round as algorithm A_ϵ.

Our algorithm L is based on algorithm A_ϵ. It chooses from among successively larger finite sets of possible cutoff points to converge to optimal. A tree data structure, which is modified dynamically, is used to store the estimated quality of each considered cutoff point. Algorithm L_s sets appropriate parameters based on the available space s, and uses algorithm A_ϵ to converge approximately to optimal.

We first describe algorithm A_ϵ which lies at the heart of our optimal algorithms L and L_s.

8.3 The Main Idea: Algorithm A_ϵ

Algorithm A_ϵ takes as parameters ϵ and M, and attempts to achieve an expected cost on a given round which is at most ϵ greater than the expected cost incurred by the optimal cutoff. In keeping with the terminology commonly used with learning algorithms, we also call a resource use time an "example." Our algorithms estimate optimal cutoffs based on past resource use times; in other words, they estimate optimal cutoffs based on the examples they have seen.

Algorithm A_ϵ works in two stages. In the *first stage*, it uses a small number of examples to generate a small number of candidate cutoffs. (For the small number of rounds that constitute the first stage, the algorithm chooses an arbitrary cutoff, say buying immediately.) It fixes these candidate cutoffs and then starts its *second stage*. For the tth round in the second stage, it evaluates the candidate cutoffs on the past $t-1$ examples, and chooses the cutoff with minimum total cost. The important point is that these small number of candidate cutoffs when generated carefully are sufficient to achieve a small enough cost, as described in Section 8.4.2. Also, updating these cutoffs can be done efficiently, as described in Section 8.4.3. We call an ϵ such that $0 < \epsilon < 1/(\ln^2(M+c)\ln\ln(M+c))$ a *suitable* epsilon; for technical reasons, we assume in our discussions that ϵ is suitable.

Note that the problem is trivial if $c \geq M$, since no reasonable algorithm would ever buy in this case; the case of interest is when $c \ll M$.

8.3.1 First Stage

In the first stage, algorithm A_ϵ generates candidate cutoffs $b_0,\ b_1, \ldots,\ b_v$ by partitioning $[0, M]$ into v intervals. Intuitively, to be accurate in its estimations in the second phase, algorithm A_ϵ wants these candidate cutoffs to be close in one of two senses: either that the probability of a point falling between them is not too large, or in absolute distance. However, for computational efficiency, we do not want too many candidate cutoffs. Hence, algorithm A_ϵ attempts to partition $[0, M]$ into $v \leq \lceil 4c/\epsilon \rceil$ intervals, such that

1. each interval is at least $\epsilon/2$ in length, and
2. if an interval has length $> \epsilon/2$, then the interior of the interval has probability at most $\epsilon/2c$.

The endpoints of the intervals define the candidate cutoffs.

We say that an interval satisfies the *computational criterion* if it is at least $\epsilon/2$ in length, and that it satisfies the *density criterion* if the probability of the interval is at most $\epsilon/2c$. (In other words, at the end of the first stage, algorithm A_ϵ ensures that every interval satisfies the computational criterion, and intervals of length greater than $\epsilon/2$ satisfy the density criterion.) Conceptually, we can think of algorithm A_ϵ as generating v' intervals that each satisfy the density criterion, and then moving the potential cutoffs apart (discarding intervals of size 0) to get $v \leq v'$ intervals such that the computational criterion holds for each interval. As a result of the VC theory [BEHb, VaC], it is easy to partition $[0, M]$ into v intervals satisfying the density criterion with high probability, by storing $\eta = \Theta(v \ln v)$ examples, and calling a procedure `generate_cutoffs`(w, η, σ) on $[0, M]$. The procedure `generate_cutoffs` breaks a specified interval into w intervals by taking a set σ of η examples, and ensuring that in any interval we have η/w examples from σ. (The procedure `generate_cutoffs` can be implemented by sorting σ to get κ and iteratively moving through η/w examples in κ to define the intervals.)

Algorithm A_ϵ implements its first stage in a space efficient manner by storing at most $O(v)$ examples at any time. It performs the first stage in three *phases*. In the first phase, algorithm A_ϵ partitions $[0, M]$ "roughly" into B big intervals, and in the second phase it refines these big intervals one by one into approximately v'/B intervals each. While refining a specific big interval, algorithm A_ϵ discards examples that do not fall in the big interval. In the third phase, algorithm A_ϵ moves potential candidate cutoffs apart to ensure that the computational criterion is met.

Formally, algorithm A_ϵ works as follows. Let $\delta = \epsilon/(4(c + M))$, and let the array σ store the examples being retained by algorithm A_ϵ. In the *first phase*, it divides the interval $[0, M]$ into $B = k \ln(1/\delta)/2$ *big intervals* (where k is a constant independent of ϵ, M, and is as defined in Lemma 8.2). It does this by collecting $\eta_1 = 4kB \ln(2B/\delta)$ examples and calling `generate_cutoffs`(B, η_1, σ) on interval $[0, M]$. The *second phase* consists of B subphases, where in the ith subphase, algorithm A_ϵ divides the ith big interval into $\lceil 4c/(B\epsilon) \rceil$ intervals. It does this by sampling at most $\eta_2' = 4B(\eta_2 + \ln(2B/\delta))$ examples, where $\eta_2 = 4kc \ln(4c/(\epsilon\delta))/(\epsilon B)$, and storing the first η_2 examples that fall within the ith big interval. Let $\eta_{2,i} \leq \eta_2$ be the number of examples stored in the ith subphase. Algorithm A_ϵ calls `generate_cutoffs`$(\lceil 4c/(B\epsilon) \rceil, \eta_{2,i}, \sigma)$ on the ith big interval. (We will see in Section 8.4.1 that $\eta_{2,i} = \eta_2$ with high probability.) At the end of the second phase, we are left with the required $v' \approx \lceil 4c/\epsilon \rceil$ intervals. In the third phase, algorithm A_ϵ ensures that the computational criterion is met. Let the ith interval at the end of the second phase be $[l_i', r_i')$. Algorithm A_ϵ sets $l_0 = 0$, $r_0 = \max(\epsilon/2, r_0')$, and processes the intervals iteratively by setting $l_i = r_{i-1}$, and $r_i = \max(r_i', l_i + \epsilon/2)$. The ith interval is defined to be $[l_i, r_i)$, and intervals such that $\eta_i = r_i$ are discarded. The total number of resulting intervals is $v \leq \lceil 4c/\epsilon \rceil$.

The candidate cutoffs are defined to be $b_i = l_i$, $0 \leq i < v$, $b_v = M$.

8.3.2 Second stage

In the second stage, algorithm A_ϵ repeatedly chooses the cutoff from among those in $\{b_0,\ b_1, ..., b_v\}$ that performed the best in the past. Formally, it formulates its tth cutoff in the second stage as follows. If $x_1,\ x_2, ..., x_{t-1}$ are the resource use times previously

seen in the second stage, for all $i \in \mathbb{N}, 0 \leq i \leq v$, algorithm A_ϵ sets

$$q_i = \sum_{j=1}^{t-1} \text{cost}_c(x_j, b_i).$$

It uses a b_i for which $q_i \leq q_k$ for all $k \in \{0, ..., s\}$ as its cutoff for the tth round.

We now study the performance of algorithm A_ϵ in terms of space used, the convergence rate, and time required for updates.

8.4 Goodness of Algorithm A_ϵ

In Section 8.4.1, we see that algorithm A_ϵ can be implemented with $O(v)$ space, and generates good cutoffs with high probability. In Section 8.4.2 we see that the distance algorithm A_ϵ is away from optimal approaches ϵ as t gets large, and in Section 8.4.3 we see that in the second stage the strategies can be updated efficiently with a tree-based data structure.

8.4.1 Guarantees about the First Stage

Let $\delta = \epsilon/(4(c+M))$, $B = k\ln(1/\delta)/2$, $\eta_1 = 4kB\ln(2B/\delta)$, and $\eta_2 = 4kc\ln(4c/(\epsilon\delta))/(\epsilon B)$ be as defined in Section 8.3.1. From the discussion in Section 8.3.1, it follows that the space used by Algorithm A_ϵ in the first stage is bounded by the number of examples we use at any time plus the number of cutoffs we retain; i.e., the space used is bounded by $B + v + \max\{\eta_1, \eta_2\} = O(v) = O(c/\epsilon)$.

The operations in the third phase of the first stage ensure that every interval satisfies the computational criterion. We say that the first stage *fails* if at the end of the first stage there is an interval of length greater than $\epsilon/2$ not satisfying the density criterion. The event that the first stage fails is a subset of the event that at the end of the second phase, there is some interval that does not satisfy the density criterion.

Let ℓ_ϵ be the *total* number of examples we see in the first stage; i.e., all examples, including the ones we discard. We now see that the first stage fails with low probability (i.e., probability 2δ).

Lemma 8.1 *Let* $\ell_\epsilon = \left\lceil kc\ln^2\left((c+M)/\epsilon\right)/\epsilon \right\rceil$ *be the number of examples seen in the first stage (where* k *is as defined in Lemma 8.2), let* $\delta = \epsilon/(4(c+M))$*, and let* E_1 *be the event that the first stage fails. Then, for any* ϵ *that is suitable,* $\mathbf{Pr}(E_1) \leq \epsilon/(2(c+M))$.

To prove the above lemma, we use a technique due to Kearns and Schapire [KeS]. Lemma 8.2 below follows immediately from the results of Blumer et al. [BEHb] using the techniques of Vapnik and Chervonenkis [VaC]. Informally, Lemma 8.2 says that m points are enough to simultaneously estimate the probabilities of *every* interval.

Lemma 8.2 *Choose* $0 < \alpha, \beta \leq 1/2, c \geq 1$, *and a probability distribution* D *on* $\mathbb{R}^+$. *Then there exists* $k > 0$ *such that if* $m = \left\lceil \frac{k}{\alpha}(\ln\frac{1}{\alpha} + \ln\frac{1}{\beta}) \right\rceil$ *then*

$$\mathbf{Pr}_{\vec{x}\in D^m}\left(\exists a, b \text{ s.t. } \mathbf{Pr}_D((a,b)) \geq 2\alpha \text{ and } \frac{1}{m}\left|\{j : x_j \in (a,b)\}\right| \leq \alpha\right) \leq \beta$$

and

$$\mathbf{Pr}_{\vec{x}\in D^m}\left(\exists a, b \text{ s.t. } \mathbf{Pr}_D((a,b)) \leq \alpha/2 \text{ and } \frac{1}{m}\left|\{j : x_j \in (a,b)\}\right| \geq \alpha\right) \leq \beta.$$

The standard Chernoff bounds will be helpful to prove Lemma 8.1.

Lemma 8.3 (Chernoff) *For t independent Bernoulli trials each of which has a probability of success at least p, let $LE(p,t,r)$ denote the probability that there are at most r successes in the t trials. Then, for $0 < p < 1$, and $0 \leq q \leq p$,*

$$LE(p,t,qt) \leq e^{-(p-q)^2 t/2p}$$

We now present the proof of Lemma 8.1.

Proof of Lemma 8.1: The value for ℓ_ϵ was obtained by assuming that the first phase requires us to look at $\eta_1 = 4kB\ln(2B/\delta)$ examples, and the ith subphase of the second phase requires us to look at a total of $\eta_2' = 4B(\eta_2 + \ln(2B/\delta))$ examples, where $\eta_2 = 4kc\ln(4c/(\epsilon\delta))/(\epsilon B)$, and $B = k\ln(1/\delta)/2$. We bound the probability of the first stage failing by the probability of the event that at the end of the second phase there is some interval that does not satisfy the density criterion.

We say that the first phase fails, if any big interval generated in the first phase has probability greater than $2/B$ or less than $1/2B$. We say that the ith subphase fails if any interval generated in the ith subphase has probability greater than $\epsilon/2c$; the second phase fails if for any i, the ith subphase fails. The lemma is proved if we can bound the probability of the first phase failing or any of the subphases failing by δ/B, since the net failure probability is then bounded by $(\delta/B)\cdot(B+1) \leq \epsilon/(2(c+M))$.

From Lemma 8.2, by setting $\alpha = 1/B$ and $\beta = \delta/(2B)$, we can easily verify that if we look at η_1 examples, the first phase fails with probability at most δ/B. We now assume that the first phase did not fail; i.e., the probability of any big interval is between $1/2B$ and $2/B$. We could fail in the ith subphase if we either do not get η_2 examples in the ith big interval, or if after using the η_2 examples we get an interval with probability $> \epsilon/2c$. From Lemma 8.3, by substituting $p = 1/(2B)$, $r = \eta_2$, and $t = \eta_2'$, we see that the probability that the number of examples that fall in the ith big interval is less than η_2 is at most $\delta/(2B)$. From Lemma 8.2, by setting $\alpha = \epsilon B/(4c)$ and $\beta = \delta/(2B)$, we see that the probability that the η_2 examples did not divide the ith big interval into subintervals with probability $\leq \epsilon/2c$ is at most $\delta/2B$. Hence the probability of the ith subphase failing is at most δ/B. □

8.4.2 Convergence of Algorithm A_ϵ

We have seen that the first stage works with high probability. The main result of this subsection is to bound the performance of A_ϵ.

Theorem 8.3 *Choose M, c such that $M > c \geq 1$. Choose any ϵ that is suitable, and let $\ell_\epsilon = \left\lceil kc\ln^2\left((c+M)/\epsilon\right)/\epsilon \right\rceil$ be the number of examples seen by algorithm A_ϵ in the first stage, where k is defined as in Lemma 8.2. There exists $k_1 > 0$ such that for sufficiently large $t \in \mathbb{N}$, for any distribution D on $[0,M]$,*

$$\mathbf{E}_{(\vec{u},\vec{x})\in D^m \times D^t}\left(\mathrm{cost}_c(x_t, A_\epsilon(u_1,...,u_m,x_1,...,x_{t-1}))\right)$$
$$\leq \left(\inf_a \mathbf{E}_{z\in D}(\mathrm{cost}_c(z,a))\right) + \epsilon + k_1(c+M)\sqrt{\frac{\ln\left((c+M)t/\epsilon\right)}{t}}.$$

To prove the above theorem, we first show that if the first stage was successful, then one of the possible cutoffs b_j generated in the first stage is only $\epsilon/2$ away from optimal

(Lemma 8.4). Intuitively, by choosing the cutoff with minimal cost in the second stage, we are close to b_j in cost. We then bound the error in expected cost resulting from the first stage failing and prove Theorem 8.3.

Lemma 8.4 *Choose $0 < \epsilon \leq 1/2$, $c \geq 1$, $s \in \mathbb{N}$, and a probability distribution D on $[0, M]$. Choose $0 = b_0 < b_1 < ... < b_s = M$. If for all $j \in \{1, ..., s\}$, either $\mathbf{Pr}_D((b_{j-1}, b_j)) \leq \epsilon/2c$, or $b_j - b_{j-1} = \epsilon/2$, then there exists $i^* \in \{0, ..., s\}$ such that*

$$\mathbf{E}_{z \in D}(\text{cost}_c(z, b_{i^*})) \leq \inf_a \mathbf{E}_{z \in D}(\text{cost}_c(z, a)) + \frac{\epsilon}{2}.$$

Proof: Intuitively, if the optimal cutoff lies between b_{j-1} and b_j, the way in which the candidate cutoffs were chosen ensures that the interval (b_{j-1}, b_j) is "small enough" (in probability or absolute size) so that one of b_{j-1} or b_j is close to optimal.

Assume without loss of generality that no b_i is exactly optimal; i.e., for all $\delta > 0$, there exists an $a^* \notin \{b_0, ..., b_s\}$, such that $\text{cost}_c(z, a^*) = \inf_a \mathbf{E}_{z \in D}(\text{cost}_c(z, a)) + \delta$. Choose $\delta > 0$ and fix a^*, $b_{j-1} < a^* < b_j$. We now show that one of $i^* = j - 1$ or $i^* = j$ satisfies the lemma.

Case 1. $\mathbf{Pr}(b_{j-1}, b_j) \leq \epsilon/2c$. In this case, we show that the lemma holds with $i^* = j - 1$. If a resource use time z lies outside of the interval $[b_{j-1}, a^*)$, then the cutoff a^* incurs at least as much cost as the cutoff b_{j-1}, since $a^* > b_{j-1}$. If the resource use time $z \in (b_{j-1}, a^*]$, then the expected extra cost of cutoff b_{j-1} is at most $c \cdot \mathbf{Pr}_D((b_{j-1}, a^*)) \leq c \cdot (\epsilon/2c) \leq \epsilon/2$.

$$\begin{aligned}
\mathbf{E}_{z \in D}(\text{cost}_c(z, b_{j-1})) &\leq \mathbf{E}_{z \in D}(\text{cost}_c(z, a^*) \mid z \notin [b_{j-1}, a^*)) \cdot \mathbf{Pr}_{z \in D}(z \notin [b_{j-1}, a^*)) \\
&\quad + \mathbf{E}_{z \in D}(\text{cost}_c(z, a^*) + c \mid z \in (b_{j-1}, a^*]) \cdot \mathbf{Pr}_D((b_{j-1}, a^*]) \\
&\leq \mathbf{E}_{z \in D}(\text{cost}_c(z, a^*)) + \epsilon/2 \qquad (\text{since } \mathbf{Pr}_D((b_{j-1}, b_j)) \leq \epsilon/2c) \\
&\leq \inf_a \mathbf{E}_{z \in D}(\text{cost}_c(z, a)) + \delta + \epsilon/2.
\end{aligned}$$

Case 2. $\mathbf{Pr}(b_{j-1}, b_j) > \epsilon/2c$. In this case, we show that the lemma holds with $i^* = j$. Note that $b_j - b_{j-1} = \epsilon/2$. For all $c > 1$ and all distributions D, $\mathbf{E}_{z \in D}(\text{cost}_c(z, a))$ viewed as a function of a is Lipschitz bounded in one direction in a sense. (This is in spite of the fact that this function of a has jump discontinuities in general.) That is, if $0 \leq a_1 < a_2$, then

$$\mathbf{E}_{z \in D}(\text{cost}_c(z, a_2)) - \mathbf{E}_{z \in D}(\text{cost}_c(z, a_1)) \leq a_2 - a_1.$$

Hence,

$$\mathbf{E}_{z \in D}(\text{cost}_c(z, b_j)) - \mathbf{E}_{z \in D}(\text{cost}_c(z, a^*)) \leq b_j - a^* \leq b_j - b_{j-1} \leq \frac{\epsilon}{2},$$

which implies that

$$\mathbf{E}_{z \in D}(\text{cost}_c(z, b_{j-1})) \leq \inf_a \mathbf{E}_{z \in D}(\text{cost}_c(z, a)) + \delta + \epsilon/2.$$

Since $\delta > 0$ was chosen arbitrarily, this completes the proof. □

The standard Hoeffding bounds will be useful to prove Theorem 8.3.

Lemma 8.5 (see [Pol]) *Choose $M > 0$, a probability distribution D on $[0, M]$, and $m \in \mathbb{N}$. Then*

$$\mathbf{Pr}_{\vec{x} \in D^m}\left(\left|\frac{1}{m}\sum_{i=1}^{m} x_i - \mathbf{E}_{u \in D}(u)\right| \geq \epsilon\right) \leq 2e^{-2\epsilon^2 m/M^2}.$$

We now present the proof of Theorem 8.3.

Proof of Theorem 8.3: Regardless of what happens in the first stage, for all $j \leq s$ and for all $x \in \mathbb{R}^+$, we have $\text{cost}_c(x, b_j) \leq c + M$. Thus, applying Lemma 8.5, we get for each $j \leq s$, $\alpha > 0$,

$$\mathbf{Pr}_{\vec{x} \in D^m} \left(\left| \frac{1}{t-1} \sum_{i=1}^{t-1} \text{cost}_c(x_i, b_j) - \mathbf{E}_{z \in D}(\text{cost}_c(z, b_j)) \right| \geq \alpha \right) \leq 2e^{-2\alpha^2(t-1)/(c+M)^2}.$$

Approximating $s = \lceil 4c/\epsilon \rceil$ by $8c/\epsilon$, we get

$$\begin{aligned} &\mathbf{Pr}_{\vec{x} \in D^m} \left(\exists (j \leq s) \ s.t. \ \left| \tfrac{1}{t-1} \textstyle\sum_{i=1}^{t-1} \text{cost}_c(x_i, b_j) - \mathbf{E}_{z \in D}(\text{cost}_c(z, b_j)) \right| \geq \alpha \right) \\ &\quad \leq \frac{16c}{\epsilon} e^{-2\alpha^2(t-1)/(c+M)^2}. \end{aligned} \tag{8.1}$$

Let j^* be such that b_{j^*} is the cutoff amongst the candidates with minimum cost; i.e.,

$$\mathbf{E}_{z \in D}(\text{cost}_c(z, b_{j^*})) = \min_j \mathbf{E}_{z \in D}(\text{cost}_c(z, b_j)),$$

and let $\hat{j}^*$ be the index of the cutoff used by A_ϵ in the tth round. Recall that

$$\frac{1}{t-1} \sum_{i=1}^{t-1} \text{cost}_c(x_i, b_{\hat{j}^*}) = \min_j \left\{ \frac{1}{t-1} \sum_{i=1}^{t-1} \text{cost}_c(x_i, b_j) \right\}.$$

Let E_1 be the event that the first stage was successful, i.e., for all intervals (b_{j-1}, b_j) generated in the first stage, $|b_j - b_{j-1}| = \epsilon/2$, or $\mathbf{Pr}_D((b_{j-1}, b_j)) < \epsilon/2c$. We have

$$\begin{aligned} &\mathbf{E}_{(\vec{u},\vec{x}) \in D^m \times D^t}(\text{cost}_c(x_t, A_\epsilon(u_1, ..., u_m, x_1, ..., x_{t-1}))) \\ &\quad = \mathbf{E}_{(\vec{u},\vec{x}) \in D^m \times D^t}(\text{cost}_c(x_t, A_\epsilon(u_1, ..., u_m, x_1, ..., x_{t-1})) \mid E_1) \cdot \mathbf{Pr}(E_1) \\ &\qquad + \mathbf{E}_{(\vec{u},\vec{x}) \in D^m \times D^t}(\text{cost}_c(x_t, A_\epsilon(u_1, ..., u_m, x_1, ..., x_{t-1})) \mid \neg E_1) \cdot \mathbf{Pr}(\neg E_1) \\ &\quad \leq \mathbf{E}_{(\vec{u},\vec{x}) \in D^m \times D^t}(\text{cost}_c(x_t, A_\epsilon(u_1, ..., u_m, x_1, ..., x_{t-1})) \mid E_1) \cdot \mathbf{Pr}(E_1) \\ &\qquad + (c+M) \left(\frac{\epsilon}{2(c+M)} \right) \qquad \text{(Lemma 8.1)} \\ &\quad \leq \mathbf{E}_{(\vec{u},\vec{x}) \in D^m \times D^t}(\text{cost}_c(x_t, A_\epsilon(u_1, ..., u_m, x_1, ..., x_{t-1})) \mid E_1) + \tfrac{\epsilon}{2}. \end{aligned} \tag{8.2}$$

Now, assume $u_1, ..., u_m$ make E_1 true. Fix $\alpha > 0$. Let E_2 be the event that all the estimates of $\mathbf{E}_{z \in D}(\text{cost}_c(z, b_j))$ obtained through $x_1, ..., x_t$ are accurate to within α. Then

$$\begin{aligned} &\mathbf{E}_{\vec{x} \in D^t}(\text{cost}_c(x_t, A_\epsilon(u_1, ..., u_m, x_1, ..., x_{t-1}))) \\ &\quad = \mathbf{E}_{\vec{x} \in D^t}(\text{cost}_c(x_t, A_\epsilon(u_1, ..., u_m, x_1, ..., x_{t-1})) \mid E_2) \cdot \mathbf{Pr}(E_2) \\ &\qquad + \mathbf{E}_{\vec{x} \in D^t}(\text{cost}_c(x_t, A_\epsilon(u_1, ..., u_m, x_1, ..., x_{t-1})) \mid \neg E_2) \cdot \mathbf{Pr}(\neg E_2) \\ &\quad \leq \mathbf{E}_{\vec{x} \in D^t}(\text{cost}_c(x_t, A_\epsilon(u_1, ..., u_m, x_1, ..., x_{t-1})) \mid E_2) \\ &\qquad + \frac{16c(c+M)}{\epsilon} \exp\left(\frac{-2\alpha^2(t-1)}{(c+M)^2} \right), \end{aligned} \tag{8.3}$$

by (8.1). By the triangle inequality, if $\mathbf{E}_{z \in D}(\text{cost}_c(z, b_{\hat{j}^*})) \geq \mathbf{E}_{z \in D}(\text{cost}_c(z, b_{j^*})) + 2\alpha$, then for either $v = j^*$ or $v = \hat{j}^*$,

$$\left| \mathbf{E}_{z \in D}(\text{cost}_c(z, b_v)) - \frac{1}{t-1} \sum_{i=1}^{t-1} \text{cost}_c(x_i, b_v) \right| \geq \alpha.$$

Thus, (8.3) and Lemma 8.4 imply that if E_1 is true, then

$$\mathbf{E}_{\vec{x}\in D^t}(\mathrm{cost}_c(x_t, A_\epsilon(u_1, ..., u_m, x_1, ..., x_{t-1}))) \\ \leq (\inf_a \mathbf{E}_{z\in D}(\mathrm{cost}_c(z,a))) + \tfrac{\epsilon}{2} + 2\alpha + \frac{16c(c+M)}{\epsilon}\exp\left(\frac{-2\alpha^2(t-1)}{(c+M)^2}\right).$$

Combining the above with (8.2) and setting $\alpha = 100(c+M)\sqrt{\ln\left((c+M)t/\epsilon\right)/t}$ completes the proof. □

8.4.3 Computation Time of Algorithm A_ϵ

We now describe how the predictions of A_ϵ are made efficiently. Let $\sigma_t = x_1, x_2, \ldots, x_{t-1}$ be the sequence formed by the first $t-1$ rounds in the second stage, where x_i, for $1 \leq i < t$, is the resource use time seen in round i. Recall from Section 8.3 that for the tth round, algorithm A_ϵ needs to output a strategy b_j that has minimum cost on the rounds in σ_t. Any updates to the data structures used by algorithm A_ϵ need to be made efficiently. We now describe a data structure maintained by algorithm A_ϵ that allows predictions to be output in $O(1)$ time and updates to be made in $O(\min\{x_t, c\}/\epsilon + \log(c/\epsilon))$ time. (Note that in problems of interest, $c \ll M$.)

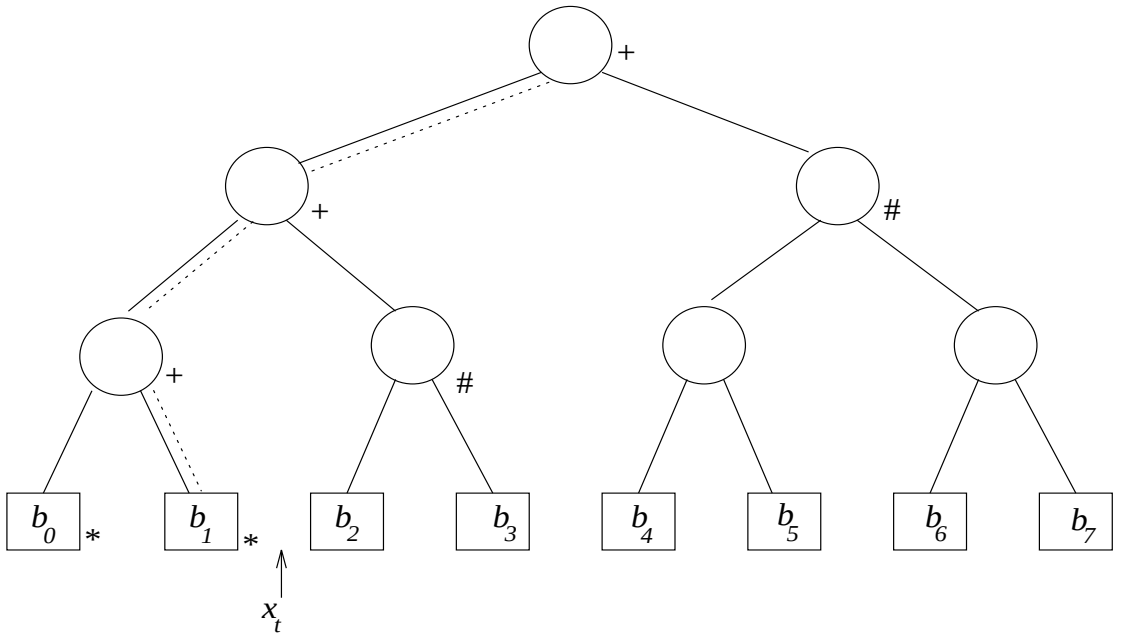


Figure 8.1: Snapshot of the data structure used by algorithm A_ϵ. In the situation depicted above, there are 8 candidate cutoffs labeled $b_0, \ldots, b_7$, appearing as leaves of the tree. The value x_t falls between b_1 and b_2. The path $P(b_1)$ is shown with dotted lines. The *diff* values of all nodes marked with a "$*$" are increased by the value of the cutoff at the node plus c. The *diff* values of the nodes marked with a "#" are increased by x_t. The *min_cutoff* and *min_cost* values of all marked nodes (whether marked with a "$*$" or "#" or "+") are updated.

Algorithm A_ϵ maintains the different candidate cutoffs as leaves of a balanced tree T. (See Figure 8.1.) We label the root of the tree by λ, and the leaves of the tree from left to

right as $0 \ldots v$, such that the jth leaf corresponds to the cutoff b_j. (For simplicity, we use the name b_j for leaf j.) Let $T(x)$ be the subtree of T rooted at node x, and let $P(x)$ be the path from the root to (and including) node x. In particular, T is $T(\lambda)$.

With each (leaf and internal) node x, algorithm A_ϵ maintains three variables, *diff*(x), *min_cost*(x), and *min_cutoff*(x). The algorithm maintains the following invariants for all t before the tth round. (These invariants define the variables.) We refer to the total cost of an algorithm that repeatedly uses a given cutoff over a sequence of resource use times as the cost of that cutoff on the sequence. The cost of using cutoff b_j for σ_t is proportional to the sum of the *diff* values of the nodes in the path from the root to b_j, i.e., the cost of using cutoff b_j for σ_t is proportional to $\sum_{x \in P(b_j)}$ *diff*(x). The variable *min_cutoff*(x) is the cutoff b_j with minimum cost for σ_t amongst all cutoffs that are leaves of $T(x)$. The variable *min_cost*(x) is closely related to the cost of the best cutoff amongst the leaves of $T(x)$; in particular, it is the cost of the best cutoff amongst the leaves of $T(x)$ minus the sum of the *diff* values of the nodes in $P($*parent*$(x))$. Formally, *min_cost*$(x) = \min_{b_l \in T(x)}\{\sum_{1 \le i < t} cost(x_i, b_l)\} - \sum_{i \in P(parent(x))}$ *diff*(i). It is important to note that since two siblings in T have the same parent, the *min_cost* values at the two siblings can be directly compared to get the *min_cutoff* value at the parent.

The tree is initialized appropriately. After round $t-1$ (i.e., at the beginning of round t), algorithm A_ϵ outputs *min_cutoff*(λ) as its cutoff for the tth round. Let $b_j \le x_t < b_{j+1}$. For the data structure to be consistent after request x_t (the tth round), the algorithm needs to increase the cost of each cutoff b_i for $0 \le i \le j$, by $b_i + c$ (which varies with i), and the cost of each cutoff b_i for which $i < m \le s$, by x_t (which is independent of i). As shown in Figure 8.1, the data structure is kept consistent by adding $b_i + c$ to the *diff* value of each of the leaves $0 \ldots j$, and by adding x_t to the *diff* values of each right child of the nodes in $P(b_j)$ that is not itself in $P(b_j)$. (Notice that exactly one *diff* value in the path from each leaf to the root is updated.) Algorithm A_ϵ updates the *min_cutoff* and *min_cost* variables for the nodes whose *diff* values were changed and their ancestors. The *min_cost* values are updated using the relation *min_cost*$(x) = \min\{$*min_cost*$($left_child$(x))$, *min_cost*$($right_child$(x))\} +$ *diff*(x). (The correctness of this update procedure follows by induction.) Also, *min_cutoff*(x) is updated to be the the *min_cutoff* of the child of x that has the smaller *min_cost*.

The number of leaves in the tree is $O(c/\epsilon)$. The time to update the *diff* values of the cutoffs b_i, $0 \le i \le j$ is $O(\min\{x_t, c\}/\epsilon)$, since each $[b_i, b_{i+1}]$ is at least $\epsilon/2$ in size. Updating the other *diff* values takes time proportional to the height of the tree, which is $O(\log(c/\epsilon))$. Hence, the amount of time to make the updates is $O((\min\{x_t, c\})/\epsilon + \log(c/\epsilon))$. The leaves $0 \ldots j$ and (most of) their ancestors can be updated online as time passes, with an extra $O(\log(c/\epsilon))$ processing required at the end.

8.5 Getting Algorithms L and L_s from Algorithm A_ϵ

In this section we prove Theorems 8.1 and 8.2 by developing our algorithms L and L_s.

8.5.1 Algorithm L

Our convergent algorithm L is obtained by running A_ϵ with continually decreasing ϵ. Clearly, if we start $A_{1/\sqrt{t}}$ sufficiently far back in the past and use the cutoffs generated by it for the tth round, we will have an algorithm that converges to optimal. For obvious

computational reasons, we do not want to maintain too many A_ϵ's with different ϵ's at the same time.

Roughly speaking, algorithm L gets over this problem by starting a new A_ϵ with $\epsilon \approx 1/\sqrt{t}$ only in round j, such that $j \approx 4^i$. It "warms up" A_ϵ through 4^{i+2}, evaluating the strategies but not using the cutoffs generated by A_ϵ. When A_ϵ is sufficiently warmed up, algorithm L uses the cutoffs generated by A_ϵ until the 4^{i+3}rd round, and then discards A_ϵ. This continual learning helps algorithm L to converge to optimal, while maintaining only a small number of A_ϵ's at any one time.

Let ℓ_ϵ, the expected number of examples seen in the first stage by algorithm A_ϵ, be as defined in Lemma 8.1. Formally, algorithm L does the following.

Algorithm L
begin
  for each round t with resource use time x_t **do**
  begin
    if there is no *current* A_ϵ **then** use a default threshold
    else use the threshold generated by the *current* A_ϵ
    endif
    if $t = 4^i - \ell_{1/2^{i+2}}$ **then** start a copy of $A_{1/2^{i+2}}$ and call this an *active* A_ϵ **endif**
    if $t = 4^i$ and $i > 2$ **then**
      discard *current* A_ϵ, if one exists
      set *current* A_ϵ to be $A_{1/2^i}$
    endif
    feed resource use time x_t to each *active* A_ϵ
  end
end

At any sufficiently large time t, there are at most three active A_ϵ's; i.e., if $4^i \le t < 4^{i+1}$, the active A_ϵ's are $A_{1/2^i}$, $A_{1/2^{i+1}}$, and $A_{1/2^{i+2}}$. Hence, the space used by algorithm L is at most three times the space used by algorithm $A_{1/2^{i+2}}$, which we know from Section 8.4.1 is $O(c/2^i) = O(c\sqrt{t})$. In round t, $4^i \le t < 4^{i+1}$, algorithm $A_{1/2^i}$ has seen at least $4^i - 4^{i-2} = (15/16) \cdot 4^i$ examples in its second stage; from Theorem 8.3, algorithm $A_{1/2^i}$ is away from optimal by at most

$$\frac{1}{2^i} + k_1(c+M)\sqrt{\frac{\ln(t(c+M)/2^i)}{15 \cdot 4^{i-2}}} = O\left(\sqrt{\frac{\ln t}{t}}\right).$$

The update time bound follows from Section 8.4.3.

8.5.2 Algorithm L_s

Algorithm L_s is exactly A_ϵ, with ϵ set appropriately such that $s = B + v + \max\{\eta_1, \eta_{2,1}\}$. (See Section 8.4.1.) Since $\epsilon = \Theta(c/s)$, Theorem 8.2 follows from the discussion in Section 8.4. The lower bound on s arises from ϵ being suitable.

8.6 Discussion

In this chapter we have looked at the problem of a sequence of single rent-to-buy choices where the resource use times are independently drawn from an unknown probability distribution. We have looked at computationally efficient strategies whose expected cost for the tth resource use converges to optimal as $t \to \infty$ for any bounded probability distribution on the resource use times. We have also looked at a fixed-space algorithm which almost converges to optimal. Our algorithm for the first stage described in Section 8.3.1 provides a space efficient way to partition an interval into approximately equal probability subintervals by sampling, and could be used in other scenarios; specifically, it can be used for defining the buckets required in our predictive strategies from Section 7.5.

It would be interesting to model the resource use times as being generated by a Hidden Markov Model (HMM). Markov models have been effectively used to analyze caching and prefetching algorithms, as discussed in Chapters 3 and 4, and in [KPR].

The single rent-to-buy problem has been studied in the worst-case setting and efficient deterministic and randomized algorithms have been developed for the problem by Karlin et al. [KMM]. In particular, 2-competitive deterministic algorithms and $e/(e-1)$-competitive randomized algorithms have been developed. In [KMM] it was claimed that there is an adaptive algorithm achieving a competitive ratio approaching $e/(e-1)$ on input sequences generated according to any time invariant probability distribution. However, their technique as stated is computationally inefficient.

Karlin et al. [KLM]have studied the spin/block problem empirically, evaluating different spin/block strategies including fixed-threshold and adaptive strategies. The virtual circuit problem has been empirically studied by Saran et al. [SaK], where they propose a Least Recently Used (LRU)-based holding time policy as performing well in their studies. The first LRU-based holding time policy they study is the 2-competitive algorithm described earlier in this chapter, and their second holding time policy involves estimating the mean inter-reference interval with exponential averaging. In [KLP], Keshav et al. empirically study an adaptive policy for the virtual circuit problem that tries to estimate the distribution of inter-arrival times by keeping a histogram of observed inter-arrival times grouped into fixed size buckets. This heuristic does not take into account our concern of keeping the probability of each bucket (i.e., interval) small.

In the next chapter, we describe in detail the notion of adaptive spindown, and precisely model the tradeoff between reponse time and energy. We also study the performance of our algorithms L and L_s for the disk spindown problem.

Chapter 9

Adaptive Disk Spindown via Rent-to-Buy

In the previous chapter, we studied how the disk spindown scenario can be modeled as a rent-to-buy problem, where spinning the disk is equivalent to renting, and a spindown is equivalent to a buy. If energy conservation were the sole consideration of a disk spindown algorithm, the cost of a buy, c, is the ratio of the energy required to spindown the disk and spin it back up to the power (or energy per unit time) to keep the disk spinning. In practice, there are two conflicting goals of a disk spindown policy: conserving energy and preserving response time performance. In *adaptive disk spindown*, the user specifies the relative importance a of latency with respect to conserving energy, and the cost of the increased latency is integrated into c, the cost of the buy. We now describe precisely how this is done.

Let P_s be the power consumed by a spinning disk. Typically, a spundown disk consumes $P_{sd} > 0$ power, where P_{sd} is much smaller than P_s. (See Figure 7.1 and Table 7.1.) Let T be the net idle time at disk.[1] This implies that the disk would consume at least $T \cdot P_{sd}$ energy independent of the disk spindown algorithm. While comparing disk spindown algorithms for how well they do in terms of energy consumed, it is instructive to compare the *excess energy*, $\mathcal{E}_X$, consumed by a disk while using spindown algorithm X; we define $\mathcal{E}_X$ as the total energy consumed by algorithm X minus $T \cdot P_{sd}$. (This is essentially equivalent to saying that the power for keeping the disk spinning is $P_s - P_{sd}$, and the power consumed by a spundown disk is 0.)

The response time delay incurred while waiting for a spinup is proportional to the amount of time required to spinup a spundown disk. A natural measure of the net response time delay is, therefore, the number of operations that are delayed by a spinup. (Another closely related measure of response time delay, the number of read operations delayed, was studied in Chapter 7, and is discussed in Section 9.3.5, Item 4.)

[1] We assume that operations are synchronous, and that every algorithm sees the same sequence of idle times at disk. If this is not true, T can be defined as the minimum taken over all algorithms of the net idle time at disk.

9.1 Adaptive Disk Spindown

In adaptive disk spindown, the user specifies a parameter a, the relative importance of latency w.r.t. conserving energy. Let O_X be the number of operations delayed by a spinup for algorithm X. Given a disk (spindown) management algorithm X, and a user specified parameter a, we define EC_X, the *effective cost* of algorithm X, as

$$\mathrm{EC}_X = \mathcal{E}_X + a \cdot \mathrm{O}_X. \tag{9.1}$$

The goal of the disk spindown algorithm is to minimize the effective cost. The effective cost models the tradeoff between energy and response time in a natural fashion. In particular, a small value of a implies that energy conservation is the more important activity, while a larger value of a implies that response time is more critical.

Minimizing effective cost can be modeled in the rent-to-buy scenario thus. Given the relative importance a, we determine the buy cost c. By definition, the value of c is the ratio of the effective cost for a spindown vs. the effective cost per unit time to keep the disk spinning. Since a spindown delays one operation, the effective cost of a spindown is $E_{sd} + a$, where E_{sd} is the total energy consumed by a spindown and a spinup. The effective cost per unit time to keep the disk spinning is $P_s - P_{sd}$. Hence, $c = (a + E_{sd})/(P_s - P_{sd})$. For a given disk, the buy cost c is linearly related to the relative importance parameter a.

We use the above framework to study our algorithm[2] L from Chapter 8 for the disk spindown problem. We first present the methodology of our simulations in Section 9.2, and then present our results in Section 9.3.

9.2 Methodology

We simulated algorithm L using a disk access trace from a Hewlett-Packard 9000/845 personal workstation running HP-UX. This trace is described in [RuWa], and this trace was also used in our experimental study described in Chapter 7. The trace was obtained by Ruemmler and Wilkes by monitoring the disk for roughly two months; it consisted of 416262 accesses to disk.

We studied our algorithm for two disks, the Kittyhawk C3014A and the Quantum Go•Drive. (The characteristics of the two drives are given in Table 7.1.) For our studies as described in this chapter, we merged the active and idle states of the disk into one active state; notice that a disk can read and write data only in the active state. By merging these two states we ensure that a "buy" corresponds to a spindown. As in Chapter 7, we assumed that a disk access takes the average time for seek and rotational latency. We also assumed that all operations and state transitions take the average or "typical" time specified by the manufacturer, if one is specified, or else the maximum time.

It is difficult to determine from a disk access trace *why* a specific access arrived at disk. We assumed that, if the disk is spundown, the application waits for the disk to spinup and complete the requested operation, and then performs the same sequence of operations as in the original system. In other words, although our simulations used disks that were different from the one on which the trace was collected, in our simulator we maintained the inter-arrival time of events at disk as in the original trace: if, in the original trace, the tth access at disk arrived Δ seconds after the $(t-1)$th access, in our simulation, we assumed

[2]Instead of scheduling a new A_ϵ at $t \approx 4^i$, in our simulations we scheduled a new A_ϵ at $t \approx 2^i$.

that the tth access arrived Δ seconds after the $(t-1)$th access was completed by the disk. The basic problem with any strategy is that data dependency between different operations cannot be derived from the trace.

We performed simulations for different values of a, the relative importance of response time to energy. For each a, we computed the buy cost c using the strategy described in Section 9.1. We compared our algorithm L against the following online algorithms: the *two-competitive algorithm*, which spins down the disk after c seconds of inactivity, and fixed-threshold policies that spindown the disk after 5 seconds, 30 seconds, and 5 minutes of inactivity; we also compared algorithm L against the *optimal offline* rent-to-buy algorithm, which knows the future and spins down the disk immediately if the next access is to take place more than c seconds in the future. For each algorithm X, we computed $\mathcal{E}_X$, the excess energy consumed, O_X, the number of operations delayed by a spinup; from these values we computed EC_X, the effective cost of algorithm X, using (9.1). For the HP trace (see Chapter 7, Table 7.2), the maximum inter-arrival time was 1770.4 seconds; the maximum a we used corresponded to a c of 1770.4.

9.3 Results

In this section we present the results of our simulations. We first see how the effective cost varies with parameter a, and then look at how excess energy and number of operations delayed vary with a. Recall that the parameter a is linearly related to the buy cost c. In particular, for the Kittyhawk disk, $c = 2.54 + a/1.225$, and for the Go•Drive, $c = 10.33 + a/1.45$.

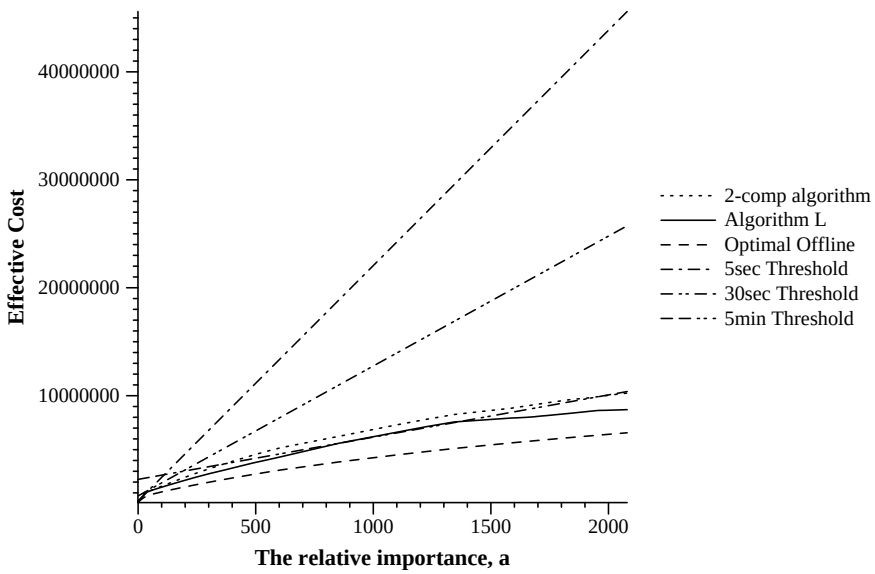
The discussion from Section 9.1 implies that algorithm L and the 2-competitive algorithm try to optimize for effective cost as defined by (9.1). In particular, for really small values of a, algorithm L will essentially try to reduce excess energy, and for really large values of a, algorithm L will essentially try to reduce number of operations delayed.

9.3.1 Effective Cost vs. a

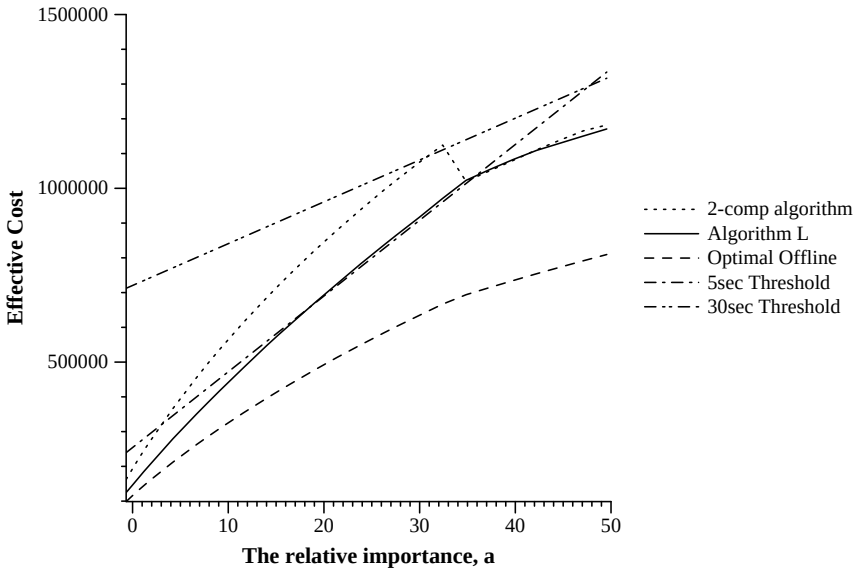
Figures 9.1 and 9.2 show how the effective cost varies with parameter a using the Kittyhawk and Go•Drive disks respectively. Each figure plots the curves for all values of a, and a clearer view for when a is small.

We observe that algorithm L performs best amongst the online algorithms for (almost) all values of a. (It is roughly 1% worse than the 5 second threshold for a lying between 18 and 34 while using the Kittyhawk disk, and for a lying between 14 and 28 while using the Go•Drive.) In particular, the effective cost for algorithm L is 6–25% lesser than the effective cost of the 2-competitive algorithm (except for a small range of values of a between 34 and 60 with the Kittyhawk disk and for a between 28 and 58 for the Go•Drive when the effective costs for the two algorithms are roughly the same).

As should be expected, each fixed threshold algorithm performs well for a very limited range of values for a. Interestingly, the 5 second threshold for certain small values of a and the 5 minute threshold for certain large values of a performs better than the 2-competitive algorithm.

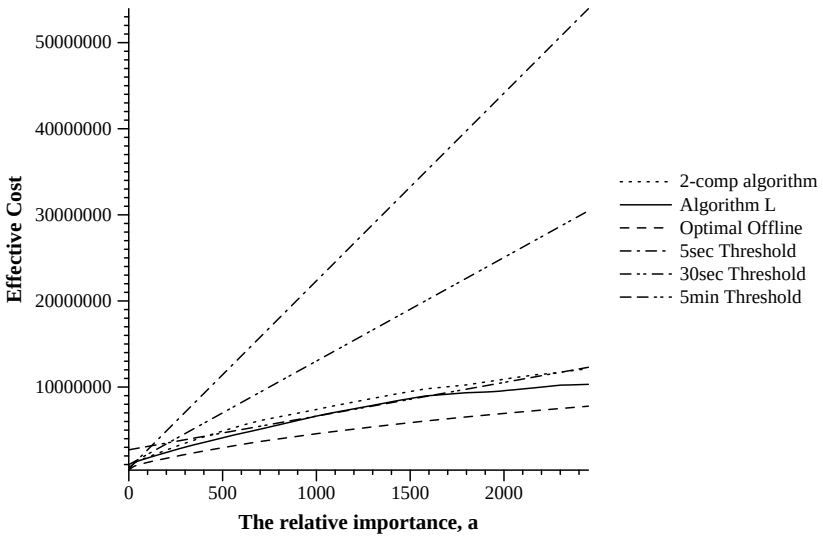


(a) For all values of a.

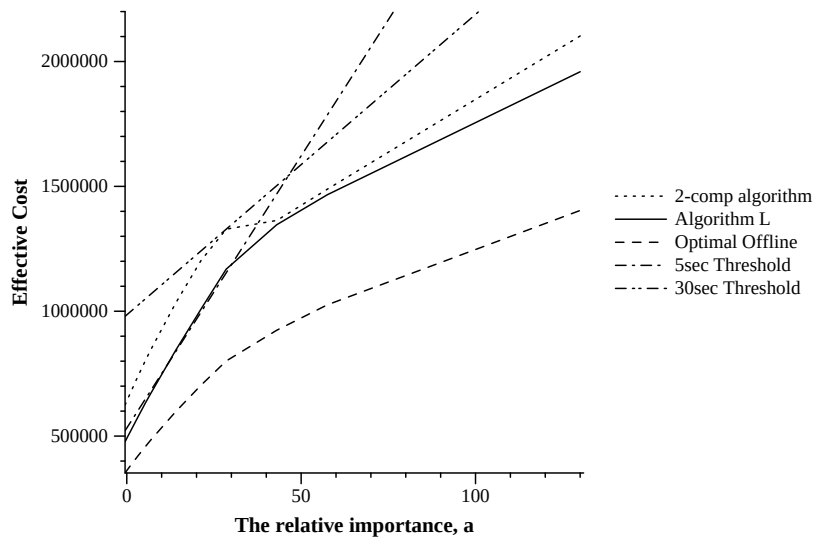


(b) For small values of a.

Figure 9.1: Variation of effective cost with a for the Kittyhawk disk. Figure (b) zooms the portion of the graph for small values of a. The effective cost of the 5 minute threshold is comparatively high (the curve lies above 2240000), and is omitted from Figure (b).



(a) For all values of a.



(b) For small values of a.

Figure 9.2: Variation of effective cost with a for the Go•Drive disk. Figure (b) zooms the portion of the graph for small values of a. The effective cost of the 5 minute threshold is comparatively high (the curve lies above 2700000), and is omitted from Figure (b); similarly, the curves for the 5 second and 30 second policies have been cropped at smaller values of a to show the details of the other three curves.

9.3.2 Excess Energy vs. a

As discussed in Section 9.1, when a is small, conserving energy is more important. Figure 9.3 plots the variation of excess energy with a using the Kittyhawk and Go•Drive disks for the various algorithms.

We observe that for small values of a, algorithm L has the smallest excess energy amongst all online algorithms. In fact, it does better than the 5 second threshold, and its curve is almost parallel to the curve for the optimal offline algorithm. In particular, algorithm L saves 17–60% more excess energy as compared to the 2-competitive algorithm, and 6–42% more excess energy as compared to the 5 second spindown threshold for small values of a (i.e., $a < 25$).

We also observe that for small values of a, the 5 second threshold does better than the 2-competitive algorithm in terms of saving excess energy. (From Figures 9.1 and 9.2, we observe that for most of these values of a, the 5 second threshold is also better than the 2-competitive algorithm in terms of effective cost.)

9.3.3 Operations Delayed vs. a

As discussed in Section 9.1, when a is large, we want to reduce the number of operations delayed. Figure 9.4 plots the variation of number of operations delayed with a using the Kittyhawk and Go•Drive disks for the various algorithms.

We observe two interesting phenomenon: first, the curves for the 2-competitive algorithm and the optimal offline algorithm coincide for a large range of values for a. Second, algorithm L reduces number of operations delayed over both these algorithms for sufficiently large a.

9.3.4 Adaptability and Rent-to-Buy

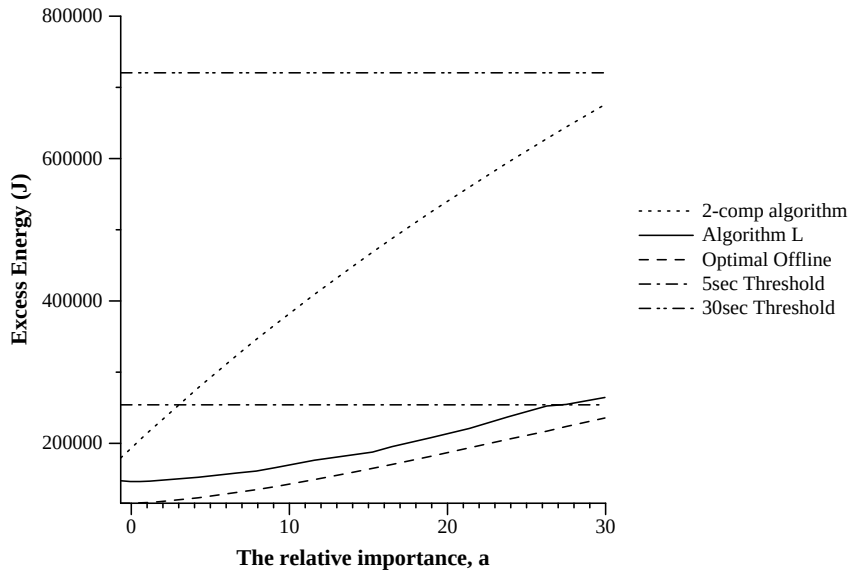
A different way of viewing the tradeoff between excess energy and response time is presented in Figure 9.5. In this figure, excess energy is plotted as a function of number of operations delayed, and the different points on the curve are obtained by varying a; in particular, the value of a (or equivalently, c) decreases from left to right along the curve. (The curve for the Go•Drive is similar in shape and is omitted.)

Figure 9.5 clearly shows the tradeoff between excess energy and response time obtained by varying a. We observe that by increasing the value of one parameter a (equivalent to varying the value of the buy cost c), we can effectively trade power for response time. Concerns on how to effectively trade power for response time have been raised for the disk spindown problem [DKB, DKM], and the rent-to-buy model provides an elegant way of achieving this tradeoff.

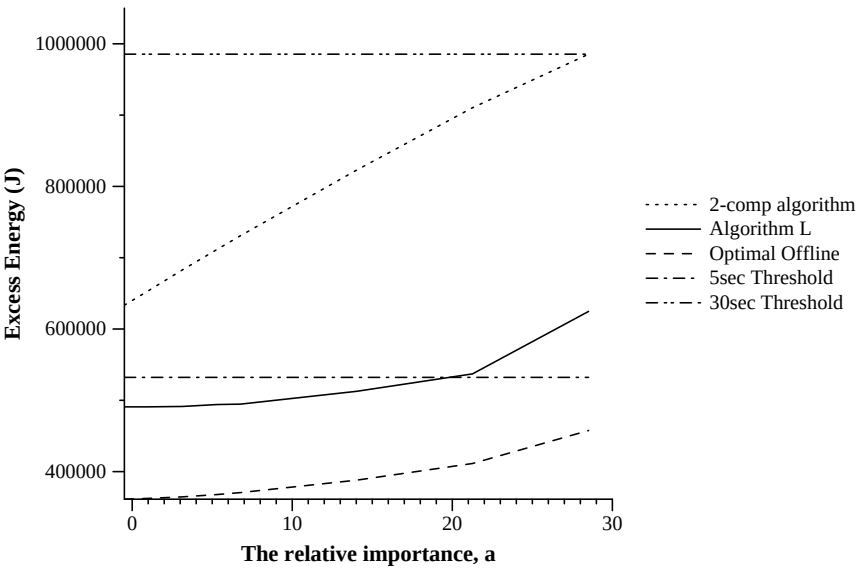
9.3.5 Other Observations

Some other observations from our simulations are as follows:

1. As mentioned in Section 9.3.2, energy conservation is crucial when a is small, and algorithm L is best amongst the online algorithms in terms of excess energy for small a. Interestingly, we observed that the excess energy of algorithm L is less than the excess energy of the 2-competitive algorithm for *all* values of a.

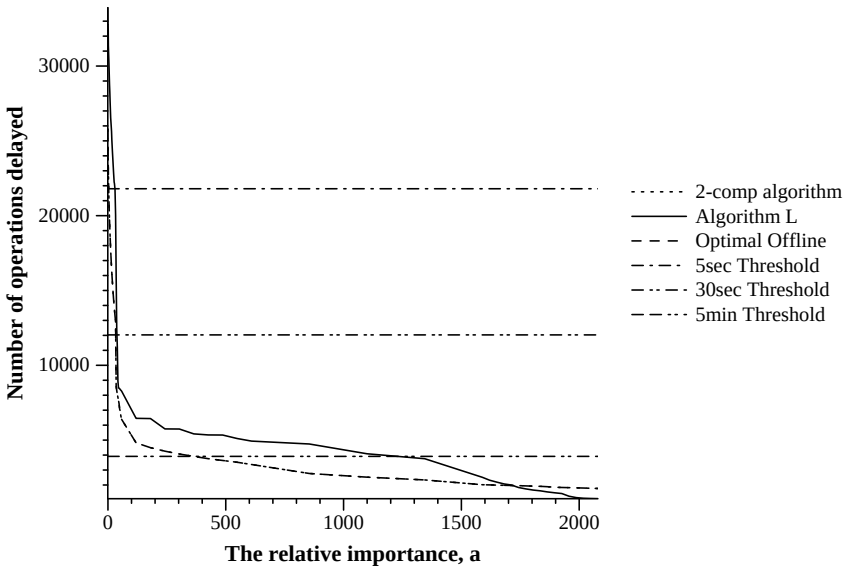


(a) Kittyhawk disk.

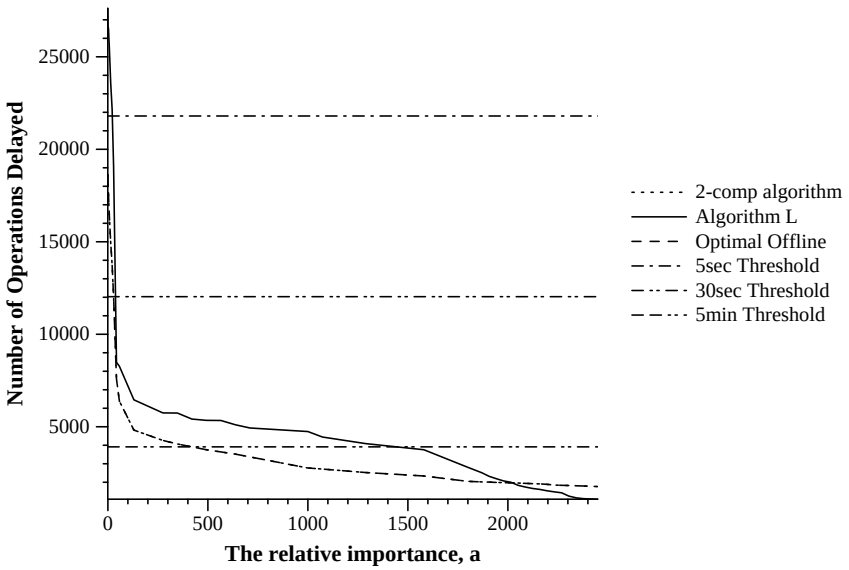


(b) Go•Drive disk.

Figure 9.3: Variation of excess energy with a for the Kittyhawk and Go•Drive disks. The excess energy of the 5 minute threshold using the Kittyhawk disk is 2249 KJ, and using the Go•Drive is 2708 KJ; the curves for the 5 minute threshold are omitted from the graphs.



(a) Kittyhawk disk.



(b) Go•Drive disk.

Figure 9.4: Variation of the number of operations delayed with a for the Kittyhawk and Go•Drive disks. The curves for the 2-competitive algorithm and the optimal offline algorithm coincide for a large range of values of a.

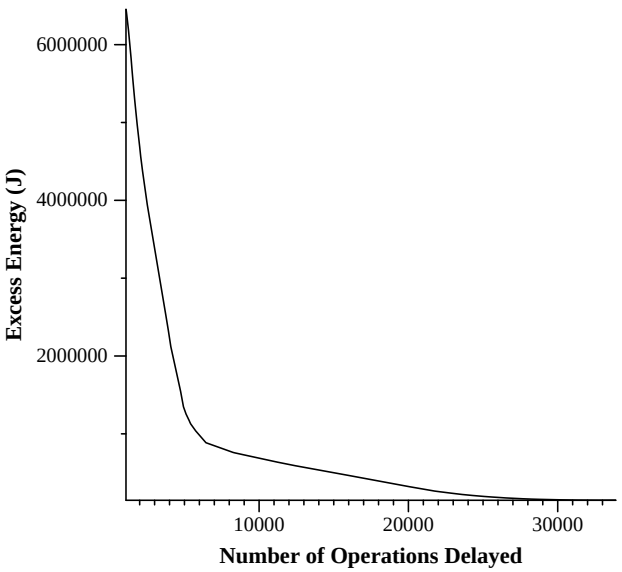


Figure 9.5: Excess Energy, $\mathcal{E}_L$, as a function of the number of operations delayed, O_L, for algorithm L. The graph was obtained by varying a (i.e., c); the value of a increases along the curve from left to right.

2. We also compared algorithm L against L_s allowing at most 25 potential cutoffs for algorithm L_s. Not surprisingly, algorithm L performed better than algorithm L_s; however, preliminary results suggest that algorithm L typically saved only 2–5% more excess energy than algorithm L_s. Allowing more potential cutoffs for algorithm L_s might help in improving its performance.

3. In our simulations, we used at most 300 cutoffs for algorithm L. The computation time for the algorithm was therefore minimal. Interestingly, algorithm L did not change its cutoffs too often in stage 2. (The cutoff changed between 14–56 times when measured over all values of a.)

4. For measuring response time performance, we used the metric of the number of operations delayed. An alternative measure of response time performance is R_X, the number of read operations delayed by a spinup for algorithm X [DKM]. This metric redefines the effective cost from (9.1) as $\mathcal{E} + a \cdot R_X$. The rent-to-buy model can be easily modified to evaluate this measure, by having different costs for a spindown (i.e., different c's) depending on whether the operation is a read or a write. We plan to consider the effect of this modification to the rent-to-buy cost in future work.

   For purely comparison purposes, Figure 9.6 plots the number of reads delayed as a function of a for the different algorithms; the algorithms are still optimizing for effective cost as defined by (9.1). (In other words, the rent-to-buy algorithms think they are optimizing for number of operations delayed, while we measure the number of reads delayed.) Interestingly, the curves from Figure 9.6 are similar to the corresponding curves from Figure 9.4a, suggesting that we should expect to obtain similar results as presented in this chapter by using the number of reads delayed metric instead of the number of operations delayed metric, when we modify the definition for effective cost appropriately.

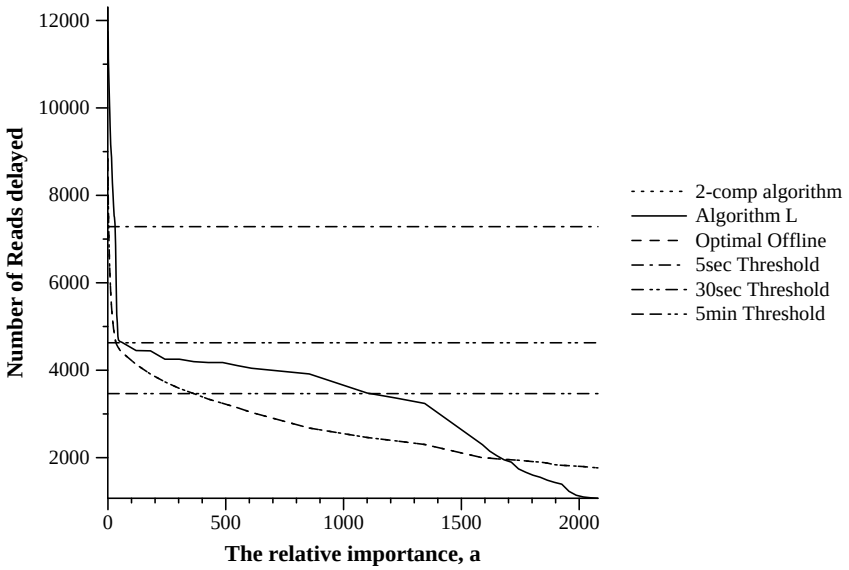


Figure 9.6: Number of reads delayed as a function of a for the various algorithms, while the rent-to-buy algorithms are optimizing using the definition of effective cost from (9.1). This graph is purely for illustration and comparison with Figure 9.4a. See Section 9.3.5, Item 4 .

9.4 Discussion

In this chapter, we have introduced a natural definition of adaptive spindown, and studied the results of simulating our optimal rent-to-buy algorithms for the disk spindown problem using disk access traces obtained from HP. Our results suggest that the rent-to-buy model is a good way to study disk spindown and related systems issues; in particular, a single parameter c effectively models the tradeoff between power and response-time. We also introduced the new metric of "excess energy" that really reflects the relative performance in terms of energy consumed of one disk spindown algorithm against another. We extended this to the natural notion of "effective cost" that incorporates the two metrics of excess energy, and number of operations delayed (weighted by a user specified parameter a), into one cost. This framework addresses many of the concerns posed in Chapter 7.

We observed that our algorithm L out-performed other online algorithms in terms of effective cost for almost all values of a; in particular, it had 6–25% lesser effective cost than the 2-competitive algorithm. In addition, for small values of a (corresponding to when saving energy is critical), we observed that our algorithm L saves 17–60% more of excess energy as compared to the 2-competitive algorithm, and 6–42% more excess energy as compared to the 5 second fixed threshold.

Chapter 10

Alphanumeric Selectivity: Predicting the *How Much*

Wall Street indexes predicted nine out of the last five recessions.
—Paul A. Samuelson (1966)

Query optimization is an integral part of relational databases [Dat, SAC, Ull]. For example, consider the selection predicate

salary between 30K and 60K (10.1)

of a payroll database, which requests records of all employees whose salary lies between 30K and 60K. It is critical to be able to predict *selectivity*, i.e., the fraction of rows in the database that satisfy the selection predicate; in the above example (10.1), the selectivity is the fraction of rows in the payroll database where the *salary* field is within the specified range. The prediction would be used by the query optimizer to determine whether to use a file scan to access all rows of the table, evaluate the predicate against each row, and return qualifying rows to the user, or to use an index on the *salary* field (if present) to access only those rows that qualify. Index access costs and file scans costs are quite different. Query optimizers have cost models that estimate the access cost as a function of the predicted number of qualifying rows and find the cheaper alternative.

Models already exist in current day relational database management systems (RDBMSs) to predict selectivity for numeric fields [ASW, FlM, Iye, SAC, WVT]. With the popularity of textual data being stored in RDBMS, it has become important to predict the selectivity accurately even for alphanumeric fields. A particularly problematic predicate used against alphanumeric fields is the *like* predicate [Iye]. For example, consider the inventory of a department store that has a *parts* table, one of whose columns, *part.color*, is the color of the part. We would like to select all parts where

part.color like "∗`green`∗" (10.2)

where the "∗" represents a wildcard that matches any sequence of symbols. In other words, we want to select all parts whose color entry has the subsequence "`green`" in it; i.e., we want to select all rows such that *part.color* has the substring "`green`" in it. The colors "`light green`", "`dark green`", "`greenish blue`", would all satisfy predicate (10.2). We refer to

selectivity of the *like* predicate as *alphanumeric selectivity in the presence of wildcards*, or simply, *alphanumeric selectivity*.

Notice that the problem of predicting alphanumeric selectivity falls in the same domain as predicting numeric selectivity, but as we will see, techniques currently used for estimating numeric selectivity are unsuitable for predicting alphanumeric selectivity.

Algorithms to predict selectivity can typically preprocess the database (during off-hours, e.g., during the weekend) to build a small *catalog*. (This pass is sometimes referred to as the *preprocessing phase* or *numstats phase*.) During query optimization, or the *online phase*, this catalog is consulted to estimate selectivity; the processing in the online phase must be minimal.

In Chapter 11 we present our techniques for predicting selectivity for the *like* predicate; i.e., techniques for estimating alphanumeric selectivity.

10.1 Background and Related Work

Models already exist in current day RDBMSs to predict selectivity for numeric fields [ASW, FlM, Iye, SAC, WVT]. Typically, in the preprocessing phase, a few numbers that "capture" the distribution of data are accumulated and stored in the catalog. In the earlier example dealing with salaries, the RDBMS would perform an analysis of the data in the *salary* field of the database, and select a small set of salary values to store in the catalog. There are two popular strategies to select these representative salary values: In the first method, the chosen salary values partition the interval between the minimum and maximum salary value into equal length subintervals, and a histogram of the data distribution is maintained. In the second method, the representative salary values are chosen so that the number of items from the database that fall within the range specified by two consecutive representative salary values is the same for each pair of consecutive representatives. The number of salary values chosen depends on the size available to store the catalog, and is a design decision. (Notice that the second scheme for choosing the representative salaries has similarities with our scheme for selecting candidate cutoffs in the rent-to-buy problem described in Section 8.3.) A simple model is then used based on what fraction of the ranges were covered by the range specified by the *between* clause to predict the fraction of qualifying rows.

A well-studied issue related to predicting selectivity is estimating the number of unique values in a column of the table. In [ASW, FlM, WVT], interesting linear time algorithms for finding the number of unique values in a column based on probabilistic and approximate counting methods have been described.

The problem of estimating alphanumeric selectivity is conceptually a natural extension of the problem of estimating numeric selectivity: the *like* predicate, and the wildcards in the patterns are natural extensions of ranges in numeric queries. To the best of our knowledge, the problem of predicting alphanumeric selectivity in the presence of wildcards has not been studied, although it is turning out to be a pressing concern [Iye]. The techniques used for estimating numeric selectivity are unsuitable for estimating alphanumeric selectivity, since the techniques for predicting numeric selectivity intimately use the fact that they are dealing with numbers. In other words, ranges in numeric fields are used to encapsulate information about the distribution of values in a column; such ranges make little sense for general alphanumeric fields.

10.2 Our Approach

In Chapter 11, we present our strategies for estimating selectivity of alphanumeric fields with the *like* predicate. There are stringent limits on the memory and processing allowable to our predictor [Iye]; in other words, the size of the catalog must be small and there must be minimal processing in the online phase.

At first glance, the current prediction scenario seems quite different from the previous two scenarios we have looked at (namely the "what" and "when" scenarios discussed in Chapters 2–9). The rules we operate under in this case are also somewhat different in that although we need to predict online, we are allowed to preprocess the database on which the queries will be made. However, we notice an interesting intuitive link between the current prediction scenarios and the ones we have looked at in the previous chapter: *the model built by a data compressor on an input text encapsulates information about common substrings in the text.* Intuitively, text data compressors (as in [ZiLa, ZiLb]) remove repetition in the input to encapsulate information better. The data structure or model built by the data compressor stores this encapsulated information. (For example, Figure 3.1 stores information about the distinct substrings in the input string "`aaaababaabbbabaa...`".) We could use a strategy similar to a data compressor's to build a structure that captures the information about substrings in a column. Such a structure will aid in the online search.

It is important to note that not *every* data compressor's data structure is good enough for the selectivity estimation problem, since we ultimately have to match patterns that have wildcards. We present a version of the suffix tree [McC] as an appropriate data structure for the selectivity problem, since it allows searches of arbitrary substrings of the original database items; suffix trees are used by Fiala and Greene [FiG] to implement Ziv-Lempel-based algorithms for data compression.

Chapter 11

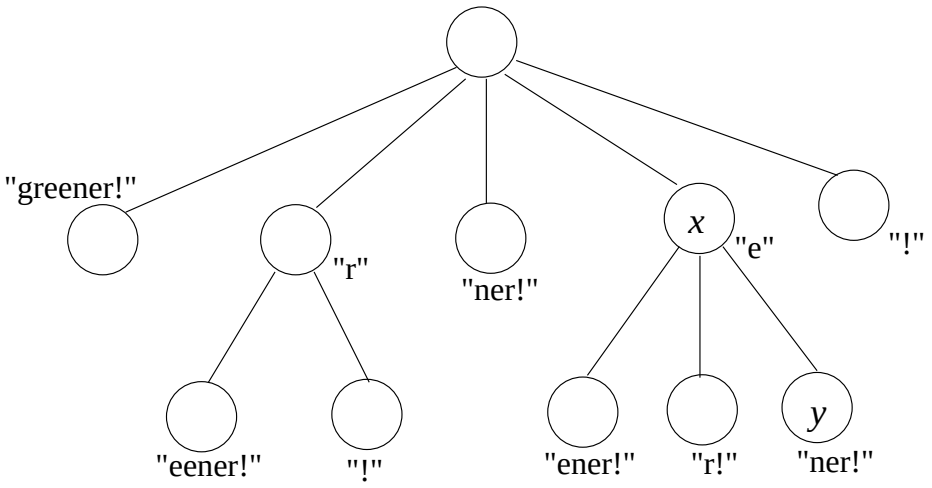
Alphanumeric Selectivity in the Presence of Wildcards

In this chapter, we describe our strategy to predict alphanumeric selectivity. Our algorithms must work under the following constraints required by practical applications [Iye]. We can make one preprocessing pass through the database. Information collected during this pass must be summarized and stored in a small *catalog*. The space allowed for a catalog for any one column of the database depends on the complexity of the column; for our domain of alphanumeric selectivity, the size of the catalog is expected to be on the order of 0.5–1 Kbyte [Iye]. To create the catalog, we can use (reasonably) large amounts of memory and compute power; the preprocessing phase will be performed when the database is not being used much, e.g., during the weekend. In the online phase, when a *like* predicate is presented, the catalog is to be consulted and a prediction of the selectivity of the predicate must be made. The processing time in the online phase must be minimal. The restrictions on the catalog size and time available in the online phase imply that the predicted value of selectivity will be approximate; the goal is to maximize the accuracy of the predictions under the space and time restrictions imposed by the problem.

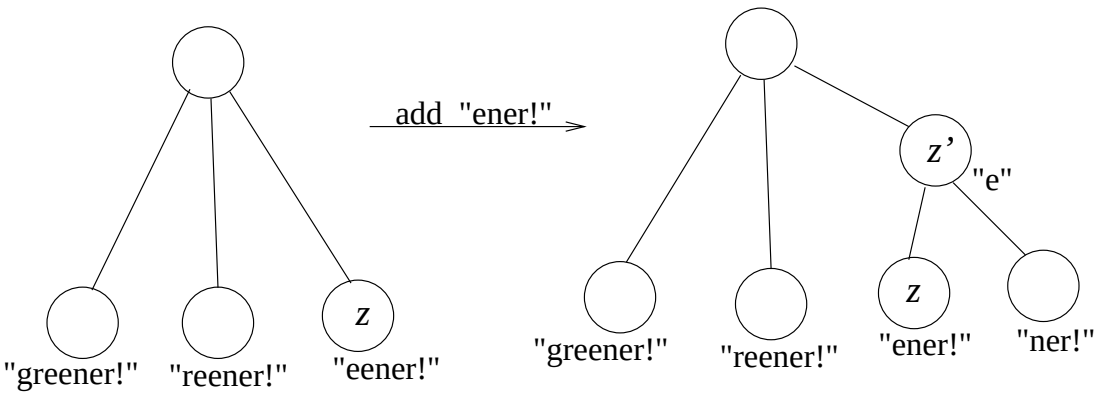
As described in Section 10.2, our strategy is based on our intuition developed in previous chapters of the close relationship between data compression and prediction. There is an emerging industry standard Transaction Processing Council (TPC) benchmark, known as the TPC-D benchmark [TPC], that involves predicates such as the *like* predicate. We study our techniques in the context of this benchmark.

We first concentrate on trying to predict selectivity for *unit patterns*, i.e., patterns where a string is sandwiched between two wildcards (e.g., the pattern "$*$`green`$*$" in the like predicate from (10.2)). Unit patterns are the most common type of patterns used with the *like* predicate [Iye]. We describe strategies to deal with general patterns in Section 11.8.

In Section 11.1, we present the suffix tree data structure, which lies at the heart of our prediction strategy. In Section 11.2, we describe our strategy in the preprocessing phase. We present our strategies for estimating selectivity in the online phase in Section 11.3. We present the methodology for evaluating our strategies in Section 11.4, and describe the results of our experiments in Sections 11.5 and 11.6. To put our results in perspective, in Section 11.7 we compare our method against a simple random sampling technique (where the catalog is built by randomly choosing from the column as many strings as possible, limited by the size of the catalog). We present extensions to our strategies in Section 11.8. Other related issues are discussed in Section 11.9.



(a) Suffix tree for string "`greener!`"



(b) Adding a string to a suffix tree

Figure 11.1: Suffix tree and its creation. In (b), we are adding string "`ener!`" to the tree. The value of α as described in Example 11.1 is "`e`".

11.1 The Suffix Tree

A suffix tree [McC] is a trie-based data structure [Knua, Mor] used for data compression [FiG], pattern matching [Wei], and other applications. We use a suffix tree-based structure for predicting the selectivity of alphanumeric strings. In this section we explain suffix trees with an example; for a formal description, see [McC].

Example 11.1 (See Figure 11.1.) A suffix tree is based on the PATRICIA trie data structure. A trie is a multiway tree with a path from the root to a unique node for each string added to the trie. A suffix tree for string σ is obtained by *inserting* the string σ into the tree; this involves *adding* all suffixes of the string σ into the tree. In other words, if σ_i is

the suffix of string σ starting at position i, inserting string σ into the tree is equivalent to adding strings σ_i, $0 \leq i \leq |\sigma|$ to the tree, where $|\sigma|$ is the length of string σ. It is convenient to assume that the last character of the input string σ is a special end-of-string symbol (that does not appear anywhere else in the string); this implies that there is a path from the root to a unique leaf for each string added to the suffix tree.

Figure 11.1a shows the suffix tree for string "`greener!`", where the character "`!`" is our unique end-of-string symbol. Each node of the tree has an *associated* string, e.g., "`e`" is the string associated with node x, and "`ner!`" is the string associated with node y. The root node has the empty or null string associated with it. The string *corresponding* to a node is the concatenation of the strings associated with the nodes on the path from the root to the node; e.g., the string corresponding to node y is "`ener!`".[1]

The tree in Figure 11.1a is obtained by inserting string "`greener!`" into the tree; this is equivalent to adding the string "`greener!`" and all its proper suffixes "`reener!`", "`eener!`", "`ener!`", "`ner!`", "`er!`", "`r!`", and "`!`" to the tree. When the string σ_i is to be added to the tree (see Figure 11.1b), we determine the *largest* prefix α of σ_i such that there is a node w in the tree with α as the prefix of the string corresponding to node w. (Recall that the root has the null string associated with it, and the null string is a prefix of every string. Hence the node w exists for all strings σ_i.) Amongst the nodes w for which α is a prefix of the string corresponding to w, let z be the node at minimum depth; the trie property of two children of a node not having a common prefix ensures that node z is unique. A simple method to determine α and z is to start at the root of the tree and iteratively move down the tree and ahead in the string σ_i by comparing the unmatched suffix of the string σ_i against the strings associated with the children of the current node in the comparison process until node z is found.

If z is not the root, we create a new node z' with corresponding string α, make z a child of z' retaining the string that z corresponds to, and create a new child of z' with string corresponding to σ_i. (Notice that the string associated with node z changes.) If z is the root, we create a new child of the root corresponding to string σ. This strategy is illustrated for our example in Figure 11.1b by showing how the tree changes when "`ener!`" is added to the tree after "`greener!`", "`reener!`", and "`eener!`" have been added to the tree. In this case, σ is "`ener!`", and α is "`e`".

Having a special end-of-string character implies that no suffix of σ is a proper prefix of another. In other words, while building the suffix tree for string σ, no string we add to the tree can be a prefix of another string added to the tree. □

Various optimizations with respect to time and space to the basic suffix tree construction algorithm we have just described have been studied [FiG, McC]. For example, if the original string σ can be accessed easily, instead of explicitly retaining the strings associated with each node in the tree, we can instead keep two pointers at each node that point into the original string σ such that the substring of σ between these two pointers is the string associated with the node. While adding σ_i to the tree, the process of determining the largest prefix α of σ_i and the node z such that α is a prefix of the string corresponding to node z can be done efficiently using extra "suffix pointers" in the tree [McC].

[1] The difference between the string *associated* with a node and the string *corresponding* to a node is important; these terms are used throughout this chapter in the sense presented here.

11.2 The Preprocessing Phase

The suffix tree as described in Section 11.1 is a natural structure to match patterns in a string. For example, given the suffix tree for string "`greener!`" (Figure 11.1), to find out whether the string "`een`" is a substring of "`greener!`", we *match* the string in the suffix tree. (The pattern to be matched is *not* terminated by the end-of-string character.) Notice that a string α is a substring of σ if and only if α is a prefix of a suffix of σ. The matching process is, therefore, similar to our trying to add "`een`" into the suffix tree; we have a success in the matching process if we find a node z such that α is a prefix of the string corresponding to node z. Finding out if α is a substring of σ is equivalent to the query: Does "$*\alpha*$" match string σ, where "$*$" is the wildcard character. The suffix tree is therefore ideally suited for matching unit patterns.

Our goal is to build a structure that allows us to answer the question: what fraction of the strings of the column R in the database match the unit pattern "$*\alpha*$". Let us for the moment disregard the issue that the catalog has to be really small in size. The observations from the previous paragraph suggest the following strategy to build a suffix tree for a column R, which is a simple generalization of the traditional suffix tree described in Section 11.1: insert all strings in column R into a suffix tree T, keeping a count at each node of the tree of how many strings in column R have as a substring the string corresponding to the node. The tree T will help us answer the selectivity of an alphanumeric query. (The selectivity of a query is the number of strings of the column R of the database that match the pattern specified by the query divided by $|R|$.) In Section 11.2.1 we describe this tree building process in detail; this is the *first stage* of the preprocessing phase. In Section 11.2.2 we study the *second stage* of the preprocessing phase, where we build a catalog from the tree T constructed in the first stage.

11.2.1 Stage 1: Building the Suffix Tree T

Our basic strategy is to insert each string σ in column R of the database into a suffix tree T; this involves adding each suffix σ_i of string σ into the tree. (Each string in column R is assumed to terminate with the same unique end-of-string character.) The method is exactly the same as described in Example 11.1 except for one difference: Unlike the traditional case of inserting a single string σ into the suffix tree, in this case we might try to add a string σ_i to the tree that has already been added to the tree. However, having a unique end-of-string character implies that no string we add to the tree can be a proper prefix of another string added to the tree. Figure 11.2 depicts the suffix tree T obtained by inserting the strings from a column of a hypothetical database, where the column has 200 strings, with 100 of the strings being "`greener!`" and the remaining 100 strings being "`grey!`". In the following discussion in this section, "node z" and "node z'" have a similar connotation as in Example 11.1.

With each node x of the tree, we maintain a variable $count(x)$. The variable $count(x)$ maintains the number of rows of the column R processed until now that have as a substring the string corresponding to node x. This is done by ensuring that all nodes on the path from the root to node z' have their *count* values incremented by one, if not already incremented while inserting string σ.

It is important that we increment the count of a node at most once while inserting a string σ into tree T. This is because for reporting selectivity, we want to determine the

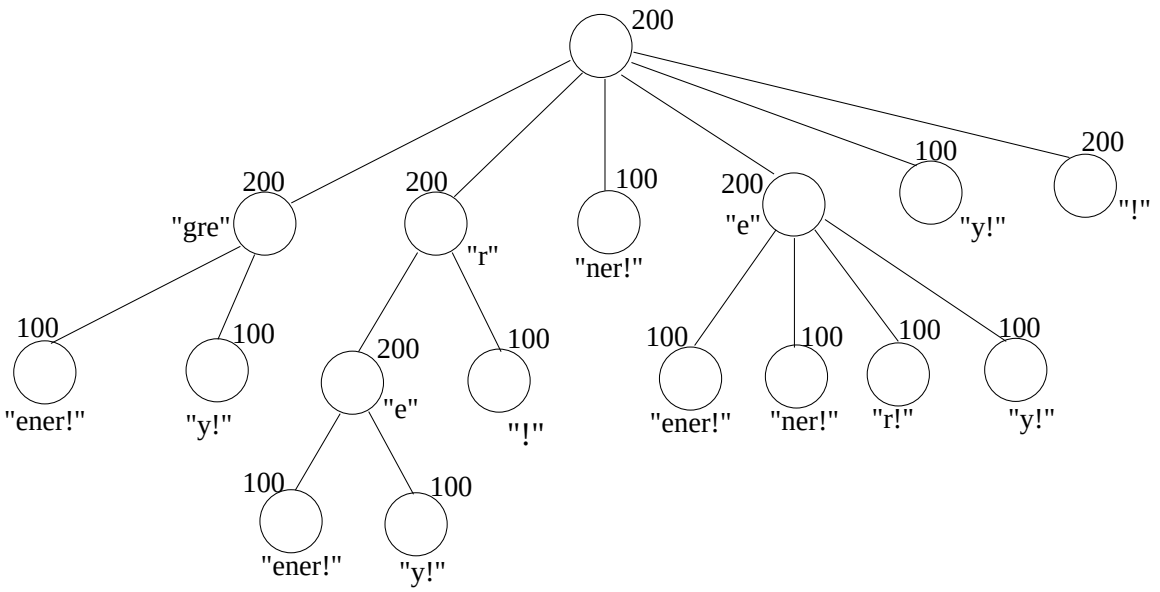


Figure 11.2: The suffix tree obtained by inserting 200 strings into the tree, with 100 of these strings being "`greener!`" and the remaining 100 being "`grey!`".

number of rows of R that match a given unit pattern, not the total number of times the unit pattern appears in the column. This implies that the value of the *count* variable at any node is at most the value of the *count* variable at its parent; the value of the *count* variable at a node is *not* equal to the sum of the values of the *count* variable at the children of the node.

Memory Restrictions while Building the Tree

Although we can expect to have a large amount of memory and processing capacity in the preprocessing phase, we must keep in mind that the column of the database is typically very large, and the tree could grow substantially. Memory is never unlimited in practice; there are various schemes we can use to effectively implement the preprocessing phase. We describe a few possible strategies below.

A simple method is to *limit* the maximum size of the tree. As mentioned at the end of Section 11.1, the string associated with a node can be stored as two fixed-size pointers into the database. Based on the maximum available memory size, and the amount of space occupied by a node of the suffix tree, we can determine the maximum number of nodes M that we can retain in the suffix tree for the preprocessing phase. While the tree is being built we regularly prune out nodes of small count when the number of nodes in the tree threatens to exceed the maximum M. (The pruning process can be done via a depth first search of the tree, for example.) The logic behind pruning out nodes with small count is that the strings corresponding to these nodes seldom occur in the database, and intuitively, we do not lose much information by removing these nodes. (This intuition will be made clearer in Section 11.2.2 when we describe how we build the catalog.)

A related question is how much to prune. The simplest strategy is to choose the smallest i such that after pruning out all nodes with value of *count* at most i, the resulting number of nodes is less than M. (Our implementation used this strategy.) There are certain

enhancements to this strategy suggested by cleaning policies in log structured file systems [OuD] that can be used, especially if we observe during the tree building phase that the pruning process is occurring too often; we could prune out more nodes and ensure that a fraction of total memory is unused.

In the context of data compression, least recently used-based strategies for deciding which nodes to prune [BuB]. We expect these strategies to be less suited to our environment, since we care more about the count at the nodes.

Another strategy to implement the preprocessing phase using more memory than is available physically is to assign nodes of the tree to pages intelligently, and fetch in the nodes of the tree in a manner similar to the technique presented in Section 5.3.1 for paging in the data structure nodes of prefetchers derived from data compressors. The usefulness of this strategy will depend on locality of access in the tree, and whether it provides additional gains over the simpler pruning strategy described earlier.

Issues of Time and "Suffix Pointers"

Let us assume that the string associated with a node is stored as two pointers into the database. When we add a string σ_i to the tree, we need to make string comparisons between string σ_i and the strings associated with some of the nodes of the tree. Since the database is typically large, it will be in secondary memory, and retrieving the strings from the pointers will cause I/Os. "Suffix pointers" as mentioned at the end of Section 11.1 and a hashed implementation [McC] need to be used when the string associated with a node is stored as two pointers; suffix pointers short-circuit the search for node z and ensure that minimal I/O is required per string insertion. We should be careful, however, to maintain valid suffix pointers while pruning the tree to conserve space.

Observe that it is space-efficient to explicitly store the string with the node, if the amount of space required to store the string is less than the space required to store two pointers. A combination of storing the string explicitly at some nodes and pointers at others will achieve a good tradeoff; a bit at each node can specify how the string associated with the node is stored.

11.2.2 Stage 2: Building the Catalog C

At the end of stage 1 of the preprocessing phase, we have a (reasonably large) suffix tree T. As observed in Section 11.2, but for the occasional pruning we do while building the tree, we can exactly match any unit pattern in the tree to get the selectivity of the unit pattern predicate. Recall that we can retain only a small catalog C for use in the online phase.

The main intuition in deriving the catalog is related to the intuition for pruning regularly while building the tree. The predictions made using the catalog will help the query optimizer to decide on the database access strategy. We can think of the query optimizer as choosing between a sequential scan of the database versus an index scan, for example, depending on how large the selectivity is. (In practice, the query optimizer is more complicated since queries are complex, but the basic idea is still the same.) In other words, we are interested in knowing as accurately as possible how much the selectivity is when the value of selectivity is large.

We derive a catalog from the suffix tree based on the size c that we are allowed for storing the catalog. (The size c of the catalog for the specific type of columns and queries we are dealing with is expected to be on the order of 0.5–1 Kbyte [Iye].) We do this by

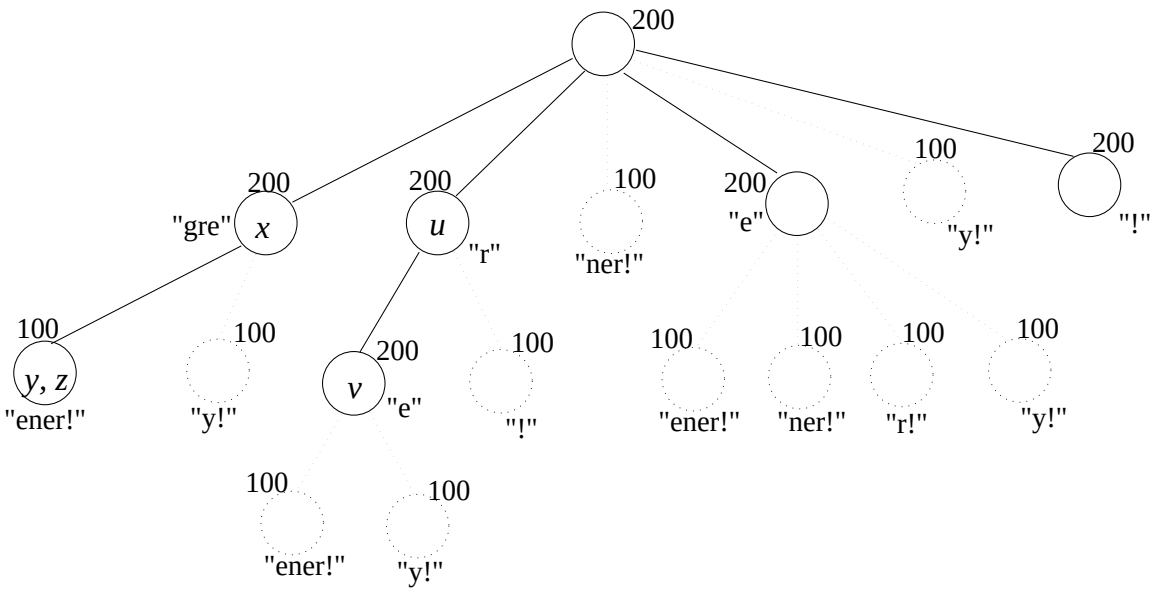


Figure 11.3: Deriving the catalog from the tree in Figure 11.2. We can store seven nodes in the catalog. The portion of the tree depicted by dotted lines is pruned away to get the catalog.

pruning out all nodes of the tree except the n nodes with the largest value of *count* that can be stored using c bytes. We call the maximum count of any node that was pruned from tree T while building the catalog C as the *prune count*. The value of n is determined by the size of a node in the pruned tree and c; we describe how to determine n below. Figure 11.3 depicts the pruned tree stored in the catalog derived from the tree in Figure 11.2, obtained by pruning out all but seven nodes; the six nodes with a value of *count* equal to 200 are retained in the catalog, and the the 11-way tie for the seventh node is broken arbitrarily. The prune count in this case is 100.

There is one important observation in the catalog building process: independent of how the strings associated with a node were stored during the tree building phase, we cannot store them in the catalog as two pointers into the database, since during the online phase, we do not want to perform I/Os and access the database column.

Compressing Information Within a Tree Node

Space in the catalog is a premium resource; time available in the online phase is also limited. We are faced here with the classical space-time tradeoff problem. We describe how some simple encoding strategies can help us effectively utilize the small amount of space available in the catalog, and how to determine n, the number of nodes that we can store in the catalog.

We store the nodes of our tree in a (contiguous) array. The children of a node are stored contiguously in the array. With each node we store the following information: the string associated with the node using a simple encoding scheme described below; the value of *count* at the node with a naive strategy using $\log|R|$ bits, where R is the number of rows in the column of the database, or an approximate value of *count* using fewer bits, e.g., the value of *count* rounded to the closest thousand; a bit to specify whether the node is a

leaf, and if not, a pointer into the array to the first child of the node using $\log c$ bits; and a bit to specify whether the node is the last child in the list of children of its parent. (In Section 11.8.1, we see that keeping a different type of count using approximately 2–4 bits for the count helps in answering the less-frequently seen queries involving non-unit patterns.)

We store the strings associated with the node along with the node using a simple encoding scheme to avoid having to store the null character that signifies the end of the string. The nodes with higher count will be closer to the root and because of the higher branching we expect to see closer to the root, the strings associated with these nodes will have smaller length. In our scheme, we use a small number of bits b to identify the string length. Strings with the most common $2^b - 1$ string lengths are stored without the trailing null character, and the other strings are stored with an explicit trailing null character. For example, we can use 2 bits to identify the length of the string (as 1, 2, 3, or > 3 characters), store strings of length less than 3 without an explicit trailing null character, and store all other strings with an explicit trailing null character.

An alternative strategy is to extract common substrings from the strings associated with the nodes, and store them in a structure separately from the tree. The strings associated with a node can be encoded as two pointers into this structure. It is important to note that this auxiliary structure will be a part of the catalog. This strategy will be useful when the strings associated with different nodes have many common substrings. We expect this to happen only when the catalog size is reasonably large and we can store a large number of nodes of the tree in the catalog.

To pack the maximum number of nodes into the catalog, we can list the nodes in order of decreasing count, and perform a simple calculation to determine the "break point," i.e., the last node that can be squeezed into the catalog. In other words, for a fixed x, we can determine from the tree T built in the first stage how many bytes will be needed to store x nodes by knowing the distribution of string lengths for these x nodes and the (fixed amount of) storage required for other components of the node. (An illustration of this calculation is discussed in Section 11.5.3.) By increasing x we can determine the largest x such that x nodes can be stored in c bytes. We can determine n without performing the required calculations for all x from 1 to $n + 1$. In practice, a rough estimate for n can be obtained from the average (or maximum) string lengths take over the "few" nodes with the largest counts; an upperbound for n is obtained by observing that each string is at least one character long, a lowerbound for n can be obtained by assuming that the trailing null character is explicitly stored with each string.

Our observation, as described in Section 11.5, is that we can store approximately 100–200 nodes of the tree in the catalog, assuming that the catalog size c is between 0.5 and 1 Kbyte; suggesting an average requirement of about 5 bytes of storage per node.

Compressing Strands

Notice that when a tree is pruned, it is possible that we are left with a tree having nodes with exactly one child. It could be advantageous with respect to space to "collapse" these paths into one node. For example, the nodes x and y in Figure 11.3 can be merged into one node, as can nodes u and v.

The semantics of merging nodes needs to be defined carefully. In particular, we need to define the *count* of merged nodes, and identify any information that we could lose by merging nodes. For example, in Figure 11.3, if we merge nodes x and y, so as not to lose

information about the different values of *count* stored at these two nodes we need to keep a special (set of) pointers into the string and associate the appropriate count with each of these pointers. Apart from the count, there is another reason why keeping the pointers is necessary. There is semantic information in a parent-child relationship in the suffix tree. Having node y as a child of x rather than having a merged node implies that the string "`gre`" is not always followed by an "`e`" in column R.

If we are prepared to lose the semantic and/or count information by merging nodes we will gain on space. This is an engineering decision that depends on available space, typical patterns in the query processing phase, and our algorithms (described in Section 11.3) to determine selectivity in the online phase.

11.3 The Online Phase

In the online or query optimization phase, a query using the *like* predicate is presented, and we consult the catalog to estimate the selectivity of the *like* predicate. Let the query with the like predicate involve the unit pattern "$*\sigma*$". We call the string σ the "string in the unit pattern" or simply *the pattern*. With some abuse of notation, we consider the pattern as equivalent to the unit pattern, and call the selectivity of the *like* predicate the selectivity of pattern σ. The pattern is not terminated by the end-of-string character.

We use the term *prune probability* to denote the value of prune count divided by $|R|$, where the prune count is as defined in Section 11.2.2, and $|R|$ is the size of the column in the database. In this section, we will denote the pattern by σ, and the suffix of the pattern σ starting at position i by σ_i; in particular $\sigma = \sigma_0$. We denote the selectivity of pattern σ by $sel(\sigma)$, and the estimated selectivity of pattern σ using strategy X by $est(X, \sigma)$. In most cases X is obvious from context; in these cases, we drop X from our notation, and use $est(\sigma)$ to denote the estimated selectivity of pattern σ. We also think of the catalog as equivalent to the tree stored in the catalog.

The basic operation in the online phase is to *match* a pattern against the catalog to get an estimate $est(\sigma)$ for the selectivity of pattern σ, which is as close to $sel(\sigma)$ as possible. A pattern σ is in the catalog if the pattern is the prefix of the string corresponding to some node z in the catalog. Determining whether the pattern σ is in the catalog is simple:[2] start at the root of the catalog and move down the tree in the catalog and ahead in the pattern based on the labels of the transitions in the tree, where the label of a transition from a parent node to a child node is the string associated with the child node. We have a *successful match* of the pattern as soon as we reach a node z such that the pattern σ is a prefix of the string corresponding to node z; we call node z as the *node of the match*. We have an *unsuccessful match* (equivalent to pattern σ not matching in the catalog) if we do not find such a node z while moving down the tree. Clearly, if we have a successful match, $est(\sigma)$ should be equal to $count(z)/|R|$ for any reasonable estimation strategy, since from the way the catalog was built (see Section 11.2), we know that $est(\sigma)$ will then be equal to $sel(\sigma)$, barring the minimal effects of regular pruning in stage 1 of the preprocessing phase. For example, if we match the pattern σ ="`gree`" against the catalog from Figure 11.3, the node of the match, z, is the node whose corresponding string is "`greener!`", and $est(\sigma) = 100/200 = 0.5 = sel(\sigma)$.

If the entire tree T (i.e., the tree that was built in stage 1 of the preprocessing phase

[2]We do not store any "suffix pointers" in the catalog.

before being pruned to get the catalog) were stored in the catalog, then for an unsuccessful match, the logical value for $est(\sigma)$ should be 0. As described in Section 11.2.2, however, while building the catalog from tree T, we prune away most of the nodes of the tree. It is quite likely, therefore, that a pattern that would have successfully matched in the tree T will not match in the catalog. Since we know that the probability of a pattern that does not match in the catalog is at most the prune probability, setting $est(\sigma)$ equal to the prune probability might be reasonable.

For example, consider the catalog from Figure 11.3, obtained from the tree in Figure 11.2. The query pattern "`ree`" will not match in the catalog. However, we know that the prune probability for the catalog was 0.5. Using the catalog and the prune probability we can be certain that pattern "`ree`" has a selectivity of at most 0.5. This value for $est(\sigma)$ is exactly equal to $sel(\sigma)$ for this specific example. In some cases which we call *a sure zero*, we can be sure that the selectivity is 0. For example, if we try to match pattern $\sigma =$ "`gra`" in the catalog from Figure 11.3, we know that the selectivity should be zero, since if string "`gra`" were in the original column, the node x could not have had an associated string of "`gre`". (Also see the discussion under "Compressing Strands" in Section 11.2.2; we assume in this paper that strands are not compressed.)

If the prune probability is small enough in that the query optimizer's decision can be the same for all values of selectivity below the prune probability, we can safely return the prune probability as our estimation for the selectivity of unmatched patterns. The stringent limits on the size of the catalog do not guarantee that the prune probability will always be this small. We now study strategies to get a better approximation for the selectivity of queries when the pattern does not match in the catalog. In the rest of this section we concentrate on the difficult case when the pattern σ does not match in the catalog.

We have three main types of methods for matching a pattern not explicitly stored in the catalog: the *independence-based*, the *child estimation-based*, and the *depth estimation-based* methods. In each of these methods, we deal with *partial matches* of the pattern in the catalog. The idea behind partial matches is that if a pattern does not match in the catalog we can divide the pattern into sub-patterns that match in the catalog, determine the selectivity of each of these sub-patterns and put these values of selectivity together in some logical way to get the selectivity of the full pattern.

We now describe our strategies for matching grouped under the various methods. (The description of the strategies is followed by an illustrative example.) An estimate for an unmatched pattern should always be less than the prune probability, and each strategy explicitly ensures this is true; i.e., it returns the minimum of its estimation and the prune probability. We omit this obvious clause from the description of our strategies below. Similarly, if we spot a sure zero, we will always be report a selectivity of 0.

11.3.1 Independence-based Strategies, I_*

The independence-based strategies essentially parse the pattern σ into sub-patterns $\sigma(0)$, $\sigma(1), \ldots,\ \sigma(m)$ in a greedy fashion, where $|\sigma(i)| > 0,\ \forall i, 0 \leq i \leq m$, and $\sigma(0)\sigma(1)\cdots\sigma(m)$ equals pattern σ. By "greedy" we mean that $\sigma(i)$ is either the maximal match of σ_j in the catalog, where $j = |\sigma_0| + \cdots + |\sigma_{i-1}|$, or else the single character at position j of pattern σ if no non-null prefix of σ_j successfully matches in the catalog.

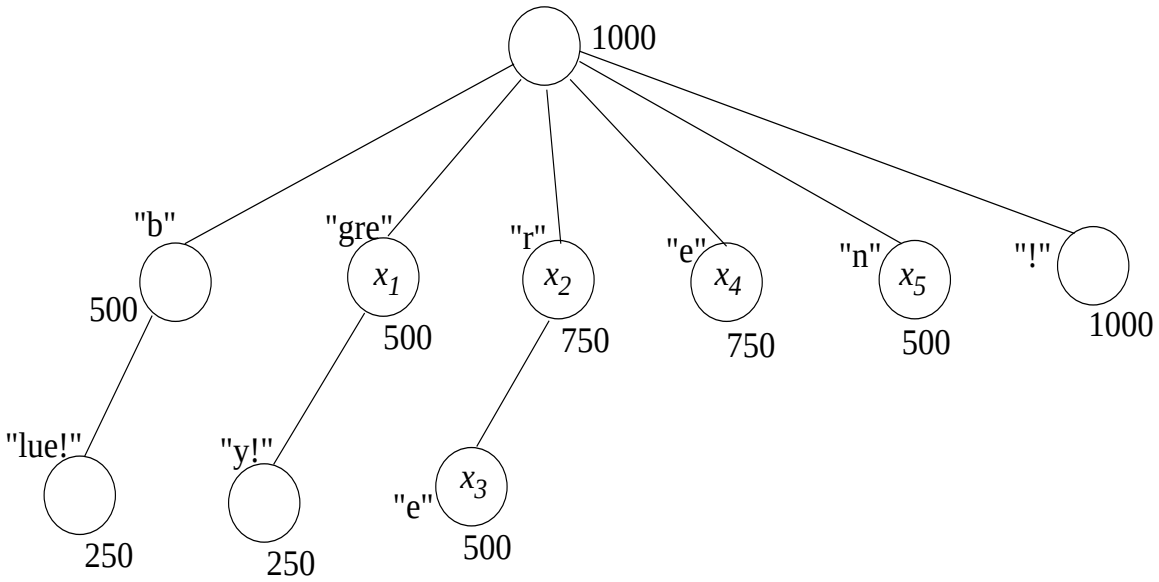


Figure 11.4: The catalog obtained by inserting a column R of a hypothetical database having $|R| = 1000$ strings, with 250 of these strings being "`green`!", 250 being "`grey`!", 250 being "`brown`!", and 250 being "`blue`!", and retaining nodes with the maximum 10 counts in the catalog. The prune count for this catalog is 250, and the prune probability is $250/1000 = 0.25$.

Strategies I_1, I_1': If $\sigma(i)$ matches in the catalog, we define $est(\sigma(i)) = count(z)/|R|$, where z is the node of the match for $\sigma(i)$. Otherwise, $est(\sigma(i))$ is defined to be 0 for strategy I_1, and is defined to be the prune probability for strategy I_1'. The estimated selectivity of the pattern σ is $est(\sigma) = \Pi_0^m est(\sigma(i))$. The advantage of this approach is that the approximation is likely to be large for queries with large output, but will almost always be small when the actual query output is small.

Strategies I_2, I_3, I_2', I_3': This set of approaches is based on the observation that the selectivity of pattern σ is at most the selectivity of pattern σ_i, $\forall i, 0 \leq i \leq m$. Strategies I_2 and I_3 use strategy I_1 as a subroutine to determine $est(I_1, \sigma_i)$. They estimate the selectivity of pattern σ as a weighted average of the estimated selectivity $est(I_1, \sigma_i)$ for the different suffixes σ_i of pattern σ. Intuitively, we would like to weight the estimations of the longer patterns more than the estimations of the shorter patterns. Strategy I_2 uses weights varying linearly with the length of the pattern, weighting $est(I_1, \sigma_i)$ by $|\sigma| - i$, while strategy I_3 uses weights falling exponentially with the length of the patterns, weighting $est(I_1, \sigma_i)$ by $2^{|\sigma|-i}$. Strategies I_2' and I_3' are similar to I_2 and I_3, except that they use strategy I_1' as a subroutine, instead of using I_1 as a subroutine.

Example 11.2 Consider the catalog in Figure 11.4, and let the pattern to be matched be $\sigma =$ "`green`". The selectivity of σ is $sel(\sigma) = 250/1000 = 0.25$. A greedy parsing of σ gives us the sub-patterns $\sigma(0) =$ "`gre`", $\sigma(1) =$ "`e`", and $\sigma(2) =$ "`n`".

Strategy I_1 estimates the selectivity of pattern σ to be $est(\sigma(0)) \cdot est(\sigma(1)) \cdot est(\sigma(2)) = (500/1000) \cdot (750/1000) \cdot (500/1000) = 3/16 = 0.188$. In this case, strategy I_1' will have the same estimate for pattern σ as strategy I_1.

The pattern σ_1 = "`reen`" greedily parses into "`re`", "`e`", and "`n`"; the pattern σ_2 = "`een`" greedily parses into "`e`", "`e`", and "`n`"; the pattern σ_3 = "`en`" greedily parses into "`e`", and "`n`"; and the pattern σ_4 = "`n`" matches in the catalog. Strategy I_2 estimates the selectivity of pattern σ as $5/15 \cdot est(I_1, \text{“}\texttt{green}\text{”}) + 4/15 \cdot est(I_1, \text{“}\texttt{reen}\text{”}) + 3/15 \cdot est(I_1, \text{“}\texttt{een}\text{”}) + 2/15 \cdot est(I_1, \text{“}\texttt{en}\text{”}) + 1/15 \cdot est(I_1, \text{“}\texttt{n}\text{”}) = 5/15 \times 3/16 + 4/15 \times 3/16 + 3/15 \times 9/32 + 2/15 \times 3/8 + 1/15 \times 1/2 = 0.252$. Strategy I_3 estimates the selectivity of pattern σ as $(2^5 \times 3/16 + 2^4 \times 3/16 + 2^3 \times 9/32 + 2^2 \times 3/8 + 2^1 \times 1/2)/(2^5 + 2^4 + 2^3 + 2^2 + 2^1) = 0.222$.

To illustrate the difference between strategies I_1 and I_1', consider the pattern σ = "`red`". In this case, $\sigma(0)$ = "`re`", and $\sigma(1)$ = "`d`". Strategy I_1 has $est(I_1, \text{“}\texttt{red}\text{”}) = 0$, since "`d`" does not match in the catalog; however, $est(I_1', \text{“}\texttt{red}\text{”}) = est(I_1', \sigma(0)) \cdot est(I_1', \sigma(1)) = (500/1000) \cdot (250/1000) = 0.125$, since strategy I_1' estimates the selectivity of "`d`" to be the prune probability, 250/1000. □

11.3.2 Child Estimation-based Strategies CE_*

As in Section 11.3.1, let $\sigma(0)$, $\sigma(1), \ldots,$ $\sigma(m)$ be the sub-patterns obtained by parsing σ via greedy maximal matching in the catalog. Let $z(\sigma(0))$, $z(\sigma(1)), \ldots,$ $z(\sigma(m))$ be the nodes of the match for patterns $\sigma(0)$, $\sigma(1), \ldots,$ $\sigma(m)$, respectively, where $z(\sigma(i))$ is defined to be the root if $\sigma(i)$ does not match in the catalog. We denote the prune count by p. The logic behind the child estimation-based strategies is to approximately estimate the number of children of the nodes that were pruned while building the catalog.

For a node z, let $sum_child_counts(z)$ be the sum of the counts of the children of z that are in the catalog. We denote by $unaccounted_count(z)$, the quantity $\max(p, count(z) - sum_child_counts(z))$; this quantity is an estimate of the sum of the values of *count* of the pruned children of z. We define the value of *unaccounted_count* at the root to be equal to the prune count p. Recall from Section 11.2.1 that by the definition of *count*, in the full tree T from which the catalog was built, the value of *count* at a node is typically smaller than the sum of the values of *count* at the children; hence, $count(z) - sum_child_counts(z)$ could also be negative at some nodes z in the catalog.

Let us denote by $num_pruned_children(z)$ the estimated number of children of node z that were pruned in the catalog building stage of the preprocessing phase. We know that each of the children of node z that were pruned had a value of *count* at most the prune count p. We estimate $num_pruned_children(z)$ to be $\max(1, \lceil unaccounted_count(z)/p \rceil)$. We define the *num_pruned_children* of the root to be 1. The estimate for *num_pruned_children* assumes that the values of *count* for the pruned children were approximately equal. By keeping more information in the catalog, we can get better estimates of $num_pruned_children(z)$.

The CE_* strategies use the $num_pruned_children(z)$ values to estimate the selectivity of pattern σ. Intuitively, we expect that the selectivity of pattern $\sigma(i)\theta$, where θ is some character where pattern $\sigma_i\theta$ does not match in the tree, is closely related to

$$\frac{unaccounted_count(z(\sigma(i)))}{num_pruned_children(z(\sigma(i))) \cdot |R|}$$

Example 11.3 Consider the catalog from Figure 11.4. For node x_1, $sum_child_counts(x_1) = 250$; hence, the quantity $unaccounted_count(x_1) = \max(p, 500 - 250) = \max(250, 250) = 250$. Therefore, $num_pruned_children(x_1) = \max(1, \lceil 250/250 \rceil) = 1$. Similarly, we have for node x_2 that $unaccounted_count(x_2) = 250$, and $num_pruned_children(x_2) = 1$; for node x_3, we have $unaccounted_count(x_3) = 500$, and $num_pruned_children(x_3) = 2$; for node x_4, we

have $unaccounted_count(x_4) = 750$, and $num_pruned_children(x_4) = 3$; and for node x_5, we have $unaccounted_count(x_5) = 500$, and $num_pruned_children(x_5) = 2$.

Intuitively, we expect "`gre`" followed by a string of characters not starting with a "`y`" (e.g., "`gree`") to appear in approximately 250 (the unaccounted count at node x_1) divided by 1 (the number of children we estimate have been pruned at node x_1) rows of column R. □

We now describe our strategies based on the child estimation method. Notice that we do not need to pre-calculate all the $num_pruned_children(z)$ values, but can calculate the required ones on the fly.

Strategy CE_1: Along the lines of strategy I_1, this strategy is based on the premise that the $\sigma(i)$'s might be independent. The estimated selectivity of pattern $\sigma(i)$ is defined to be $est(CE_1, \sigma(i)) = count(z(\sigma(i)))/|R|$. Let θ_i be a character (or string) such that $\sigma(i)\theta_i$ does not match in the catalog. The estimated selectivity of pattern $\sigma(i)\theta_i$ is defined to be $est(CE_1, \sigma(i)\theta_i) = A_i/B_i$, where $A_i = unaccounted_count(z(\sigma(i)))$, and $B_i = num_pruned_children(z(\sigma(i))) \cdot |R|$. Using Bayes rule, strategy CE_1 estimates that θ_i follows $\sigma(i)$ in $est(CE_1, \sigma(i)\theta_i)/est(CE_1, \sigma(i))$ fraction of the rows of R that have the substring $\sigma(i)$. The selectivity of pattern σ is estimated to be $est(CE_1, \sigma) = est(CE_1, \sigma(0)) \times \Pi_{i=0}^{m-1} est(CE_1, \sigma(i)\theta_i)/est(CE_1, \sigma(i))$. (Notice that the upper index in the product term is $m - 1$.)

Strategy CE_2: Like strategies I_2, I_3, this strategy is based on the observation that the selectivity of pattern σ is at most the selectivity of pattern σ_i. The selectivity of pattern σ is estimated as $est(\sigma) = \min_{0 \le j < |\sigma|} A_j/B_j$, where the quantity $A_j = unaccounted_count(z(\sigma_j(0)))$, and $B_j = num_pruned_children(z(\sigma_j(0))) \cdot |R|$. (Pattern $\sigma_j(0)$ is the longest match in the catalog of the suffix of pattern σ starting at the jth position of σ.)

Strategy CE_3: Strategy CE_3 is a mix of strategies CE_1 and CE_2. Instead of multiplying the estimated selectivities of the different sub-patterns (like strategy CE_1 does), strategy CE_3 tries to look "down" the tree, and multiplies the number of estimated children of the different sub-patterns. The logic here is that the unaccounted count gets continually divided amongst the pruned children as we go down the tree. The selectivity of pattern σ is estimated as $est(\sigma) = A/B$, where $A = unaccounted_count(z(\sigma(0)))$, and $B = |R| \cdot \Pi_{i=0}^{m-1} num_pruned_children(z(\sigma(i)))$. (Notice that the upper index in the product term in the expression for B is $m - 1$.)

Example 11.4 Continuing the discussion from Example 11.3, for pattern $\sigma =$ "`green`", we have by greedy parsing that $\sigma(0) =$ "`gre`", $\sigma(1) =$ "`e`", and $\sigma(2) =$ "`n`"; hence, $z(\sigma(0)) = x_1$, $z(\sigma(1)) = x_4$, and $z(\sigma(2)) = x_5$.

Strategy CE_1 estimates the selectivity of pattern "`gre`" (i.e., the pattern $\sigma(0)$) to be $count(x_1)/|R| = 0.5$. It estimates the selectivity of pattern "`gree`" (i.e., pattern $\sigma(0)\sigma(1)$) to be $unaccounted_count(x_1)/(num_pruned_children(x_1) \cdot |R|) = 250/(1 \cdot 1000) = 0.25$. Strategy CE_1 expects that of the rows of R that have the substring "`gre`", an "`e`" follows "`gre`" in 0.25/0.5 or half the rows. Similarly, from node x_4 it estimates that "`e`" has a selectivity of 0.75, "`en`" has a selectivity of 0.25, and of the rows that have the substring "`e`", an "`n`" follows an "`e`" 0.25/0.75 or 1/3rd of the time. Strategy CE_1 estimates the selectivity of σ to be $0.5 \times (0.25/0.5) \times (0.25/0.75) = 0.0833$.

Strategy CE_2 estimates that "`gre`" followed any string of characters not starting with a "`y`" has a selectivity of $unaccounted_count(x_1)/(num_pruned_children(x_1) \cdot |R|) = 0.025$. For σ_1 = "`reen`" it estimates that "`re`" followed by any string of characters has a selectivity of 0.25 (using node x_3). By continuing thus (for strings σ_2, σ_3, and σ_4) it estimates the selectivity of σ as $\min_i\{unaccounted_count(x_i)/(num_pruned_children(x_i) \cdot |R|)\}$, where $i = 1, 3, 4, 5$, giving $est(CE_2, \sigma) = 0.25$.

Strategy CE_3 tries to "look down" into the tree and multiplies the estimated number of children; in other words, it thinks that the *unaccounted_count* at node x_1 will get successively divided as we go down the tree. Since $z(\sigma(0)) = x_1$, and $z(\sigma(1)) = x_4$, strategy CE_3 expects that the *unaccounted_count* at node x_1 will get divided amongst 1×3 children (from the values of *num_pruned_children* at nodes x_1 and x_4 respectively). It estimates the selectivity of pattern σ as $unaccounted_count(x_1)/(3 \cdot 1000) = 250/3000 = 0.083$. □

11.3.3 Depth Estimation-based Strategies, DE_*

The depth estimation strategies combine the intuition of the suffix tree with the strategies for estimating numeric selectivity (see Section 10.1). The idea is to capture the distribution of the depth of a node versus the (average) value of *count* at that depth.

In this case, while building the catalog, we store in the catalog a depth versus count distribution; i.e., we store $av_count(d)$, the average value of the *count* of nodes at depth d, for each (integral) value of depth d. (Although we expect $av_count(d)$ to decrease with increasing depth d, this is not necessary.) Like in the prediction of numeric selectivity, a different strategy can be used, where we store the average depth for nodes grouped into pre-determined fixed size sets, or grouped into sets with pre-determined fixed size total or average counts.

As in Section 11.3.2, let $\sigma(0), \sigma(1), \ldots, \sigma(m)$ be the sub-patterns obtained by parsing σ via greedy maximal matching in the catalog, and let $z(\sigma(i))$ be the node of the match for pattern $\sigma(i)$. We denote the depth of node z by $depth(z)$. If v is the node of match of pattern σ in the full tree T (i.e., the tree built in stage 1 of the preprocessing phase), we call $depth(v(\sigma))$ as the depth of pattern σ. In particular, $depth(z(\sigma(i)))$ equals the depth of pattern $\sigma(i)$ for all $0 \leq i \leq m$.

Strategies DE_1, DE_2: These depth estimation strategies estimate the depth of pattern σ as a weighted average of the $depth(z(\sigma(i)))$'s. We want the weights to decrease with increasing i for reasons given below. We expect that the lengths of strings associated with the nodes increases with the depth of the node. We therefore expect that the depth of pattern $\sigma(0)\sigma(1)$ will be less than the depth of pattern $\sigma(0)$ plus the depth of pattern $\sigma(1)$, since $\sigma(1)$ would have been matched from node $z(\sigma(0))$ in the full tree T, while we match $\sigma(1)$ from the root in the catalog. Intuitively, we want the ith weight to capture the ratio of the quantity depth of pattern $\sigma(0)\sigma(1)\cdots\sigma(i)$ minus the depth of pattern $\sigma(0)\sigma(1)\cdots\sigma(i-1)$ to $depth(z(\sigma(i)))$. We therefore want the weights to decrease with i. It is unclear how fast the weights must decrease; it is intuitive to weight $depth(z(\sigma(0)))$ by 1, though.

Strategy DE_1 weights the depth of the ith sub-pattern, $depth(z(\sigma(i)))$, by $1/(i+1)$, and strategy DE_2 weights $depth(z(\sigma(i)))$ by $1/(2i+1)$. (There is one technical detail: for each of the $\sigma(i)$'s that do not match in the catalog, instead of using $depth(z(\sigma(i)))$ our strategies use an estimate of 1 instead, since these non-matching characters should

Depth d	$av_count(d)$
0	1000
1	500
2	275
3	250

Table 11.1: Table of depth vs. count distribution for the tree T from which the catalog in Figure 11.4 was derived.

increase the depth estimate, while the depth of the root is, by definition, zero.) Let the estimated depth for pattern σ be d, where d is at most the maximum depth of the tree. The estimated selectivity of pattern σ is $(av_count(\lfloor d \rfloor) + (d - \lfloor d \rfloor) \cdot (av_count(\lceil d \rceil) - av_count(\lfloor d \rfloor)))/|R|$.

Example 11.5 Consider the catalog from Figure 11.4, and let the pattern σ = "`green`"; we have, by greedy matching that $\sigma(0)$ = "`gre`", $\sigma(1)$ = "`e`", and $\sigma(3)$ = "`n`". The depth vs. count distribution for the tree T from which the catalog in Figure 11.4 was derived is given in Table 11.1. (This table was calculated before the tree T was pruned to get the catalog. For example, considering the catalog alone, the average count of nodes at depth 1 is $(500 + 500 + 750 + 750 + 500 + 1000)/6 = 666.67$.) The depth of each of $\sigma(0)$, $\sigma(1)$, and $\sigma(2)$ in the catalog is 1.

Strategy DE_1 estimates the depth of pattern σ as $1 \cdot 1 + 1/2 \cdot 1 + 1/3 \cdot 1 = 1.83$. Using Table 11.1, it estimates the selectivity of pattern σ as $(av_count(1) + 0.83 \cdot (av_count(2) - av_count(1)))/|R| = 0.313$. Strategy DE_2 estimates the depth of pattern σ as $1 \cdot 1 + 1/2 \cdot 1 + 1/3 \cdot 1 = 1.53$. Using Table 11.1, it estimates the selectivity of pattern σ as $(av_count(1) + 0.53 \cdot (av_count(2) - av_count(1)))/|R| = 0.38$. □

Strategies for estimating selectivity with the other types of suggested count versus depth distributions can be obtained similarly.

11.4 Methodology for the Experiments

We experimentally evaluated our strategies from Sections 11.2 and 11.3 for the preprocessing and online phases. There is an emerging industry standard Transaction Processing Council (TPC) benchmark, known as the TPC-D benchmark [TPC], that involves predicates such as the *like* predicate. We studied our techniques in the context of this benchmark.

In particular, we used the *dbgen* program, which is part of the TPC-D benchmark, with a scale factor of one to generate a database. (A scale factor of one corresponds to a database of size 1 Gbyte.) The specifications for the benchmark suggest a few columns on which queries with the *like* predicate are used. From amongst these columns, we chose the P_NAME column of the PARTS table to test our techniques; the P_NAME column is "harder" that the other columns on which the *like* predicate is expected to be used in that the P_NAME column is expected to have a large number of distinct entries; we expect this column to be "typical" in that the strings in this column are not random alphanumeric

strings. For a scale factor of one, the P_NAME column is expected to have 200K rows of strings.

The benchmark specifies that the P_NAME column is populated as follows: there are 92 different *base patterns* specified in the benchmark, where each of the patterns is a color, e.g., "`green`". Each row of the P_NAME column is obtained by choosing five distinct base color patterns at random, and concatenating these five patterns separating any two patterns with a blank character, to get an entry. It is important to note that we *have not* used specific information about how the column is built to optimize our techniques; we discuss more about this in Section 11.8.2.

In Section 11.5, we present our experimental observations and validations of our techniques from Section 11.2 used in the preprocessing phase.

In the online phase, we need to match query patterns against our catalog. The benchmark does not provide strict guidelines on how to generate query patterns; it suggests that unit patterns formed from the base patterns be matched against the catalog. We do a more extensive study. We first matched each of the 92 base patterns in our catalog and determined the error distribution. (Matching a base pattern σ is equivalent to a *like* predicate with the query pattern "$*\sigma*$".) We call this the *positive single pattern* study, since each pattern exists in the column and the pattern is a base, or single, pattern. We also considered the $92 \times 91 = 8372$ *double patterns* obtained by concatenating each of the unit patterns with another, separating these two patterns by a blank. These patterns also exist in the column, and we call this the *positive double pattern* study. Most patterns used with the *like* predicate are *positive patterns*, i.e., patterns that are present in the database [Iye]; hence, the above two studies are most important.

Although *negative patterns*, i.e., patterns not found in the database, are seldom seen in query patterns [Iye], it is important to understand the effect of our matching strategies via-a-vis negative patterns. We generated three sets of negative patterns by taking the 92 base patterns and introducing one, two, and three errors in each basic pattern, respectively. These are particularly harsh negative patterns since they differ very little from a corresponding positive pattern. The intuition behind generating these negative patterns is that, even if a user is not malicious, people sometimes make typing mistakes.

Notice that from the way the column is built, we expect each positive single pattern to have a selectivity of $5/92 = 5.4\%$. In other words, we expect each base color to be present in roughly $200\text{K} \times 5/92$ rows of the P_NAME column, i.e., in 10869 rows. Similarly, the selectivity of a positive double pattern is expected to be 0.048%, i.e., a positive double pattern in expected to be present in 96 rows of the P_NAME column.

We present the results of using our matching strategies from Section 11.3 for matching the positive and negative patterns against our catalog in Section 11.6. It is important to keep in mind the distinction between positive and negative patterns: positive patterns are present in the P_NAME column, and therefore have a positive selectivity, while negative patterns are not present in the P_NAME column, and therefore have a selectivity of zero.

11.5 Results for the Preprocessing Phase

In the preprocessing phase, we took each of the 200K patterns in the P_NAME column of the database and inserted them into a suffix tree T. We then pruned the tree T to retain 100 nodes; these 100 nodes formed the catalog for our experiments of the online phase. We now analyze the different statistics measured in the preprocessing phase and their implications

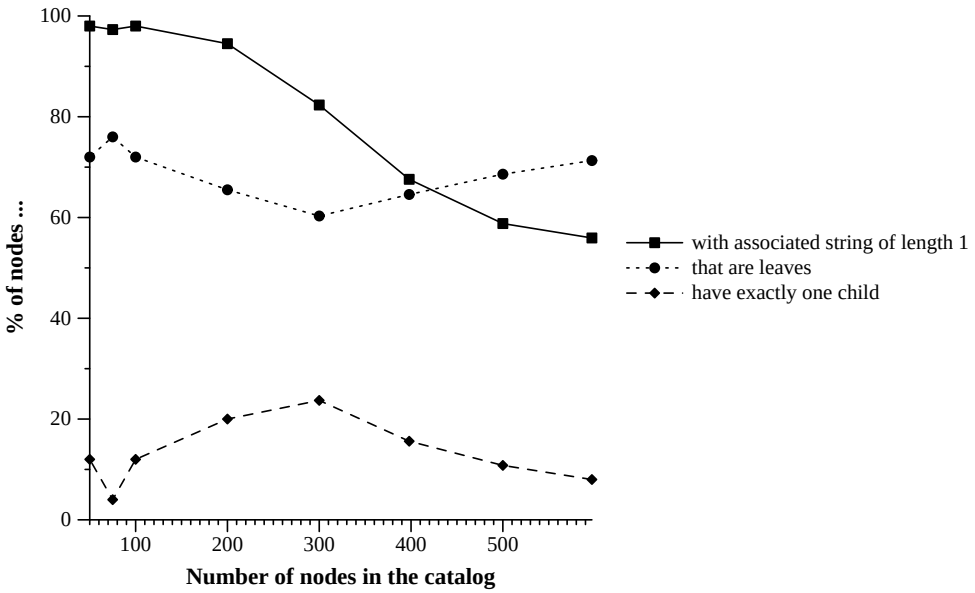


Figure 11.5: Various statistics related to the preprocessing phase as a function of the number of nodes that can be stored in the catalog.

on building the catalog. In this section, by "top nodes" we will mean nodes with the highest value of *count*.

11.5.1 Size of the Tree in Stage 1

In stage 1, i.e., the tree building phase, we varied the maximum number of nodes M retained in the tree from between 300K to 450K. (Recall from Section 11.2 that we prune the tree whenever the the number of nodes in the tree threatens to exceed M.) With $M = 450$K, we only needed to prune out nodes with value of *count* equal to 1 from the tree regularly; with $M = 300$K, we needed to prune out nodes with value of *count* up to 3. The final catalog stored a 100 nodes, and the differences in the catalogs were negligible between using $M = 300$K and using $M = 450$K; the value of *count* at a few nodes differed by 1.

11.5.2 Statistics Dealing With Catalog Building

In Figure 11.5, we plot some interesting characteristics about the tree built in stage 1 of the preprocessing phase; in particular, we present the statistics along the y-axis as a function of the number of nodes that can be stored in a catalog.

The solid line in Figure 11.5 shows that most of the strings associated with the top nodes are of length 1. This verifies our intuition from Section 11.2.2 that nodes with high count (i.e., the ones that find their way into the catalog) have strings of small length associated with them. Recall that this intuition was used in our strategy to encode the strings associated with the nodes. The average associated string length taken over the nodes whose associated strings have length greater than 1 is small; in particular, it is 2.1 characters/node for the top 200 nodes, and 3.55 characters/node for the top 500 nodes.

The dotted line in Figure 11.5 shows what percent of the nodes in the catalog are leaves, as a function of the amount of nodes we store in the catalog. (Recall that we do not need

to store a child pointer with leaf nodes.) The percentages are high as we would expect; surprisingly, it does not fall by much in the range considered. The dashed line shows the percent of nodes that have only one child. These nodes will be present as strands. (Note that if n nodes have only one child, we could have anywhere from 1 strand of n nodes to $n/2$ strands of 2 nodes each.) In our experiments, we did not collapse strands into single nodes. (See the discussion in Section 11.2.2.) Interestingly, the dotted and dashed curves seem to be mirror images of each other about $y = 40$; in other words, the sum of the percent of the number of leaves and the percent of nodes with exactly one child is approximately constant (80%).

With $M = 450$K in stage 1 of the preprocessing phase, the maximum depth of the tree is just 15. This information is required to determine the amount of memory we need to store the depth versus count distribution, which is used by strategies DE_* from Section 11.3.3 for estimating selectivity.

11.5.3 Catalog Size

Based on the above observations, and the discussion in Section 11.2.2, let us determine the amount of space required to store the top 100 nodes in the catalog. If we prune the tree obtained at the end of stage 1 of the preprocessing phase by retaining the top 100 nodes, 72 of them are leaves, 98 out of the 100 nodes have an associated string of length 1, and the root has an empty string associated with it. We will use the simple encoding scheme suggested in Section 11.2.2 to encode the strings associated with the nodes; i.e., we will use a bit to specify whether the associated string is longer than one character, and store the string with a trailing null character only if it is.

We need 18 bits with each node to store the value of *count* (since $\lceil \log_2(200,000) \rceil = 18$), one bit to inform us whether the node is the last child in its parent's list of children, and one bit to inform us whether the node is a leaf. To store these 20 bits with each node would require a total of 250 bytes. For the 28 non-leaf nodes, we need an additional 10 bits per node to store the pointer to the first child of the node (assuming we will use at most 1 Kbyte of storage); this information requires a net storage of 35 bytes. The 98 strings of length 1 require a net storage space of 98 bytes, the one string of length 3 required three bytes of storage. The net storage required is therefore 386 bytes. If we use four bits per node to store another type of count (as mentioned in Section 11.2.2, and described in Section 11.8.1), we need an additional 50 bytes of storage.

Performing similar calculations, we determine that the top 200 nodes can be stored in the catalog using 1000 bytes.

11.5.4 Size of the Catalog versus Prune Count

As we have seen in Section 11.3, the error of our matching strategies in the online phase is closely related to the prune count. Figure 11.6 depicts how the prune count varies with the number of nodes retained in the catalog. We can see the steep jump in the prune count as the number of nodes decreases. Notice that the prune count is roughly 30K when the number of nodes in the catalog is 100, and is 16K when the number of nodes in the catalog is 200.

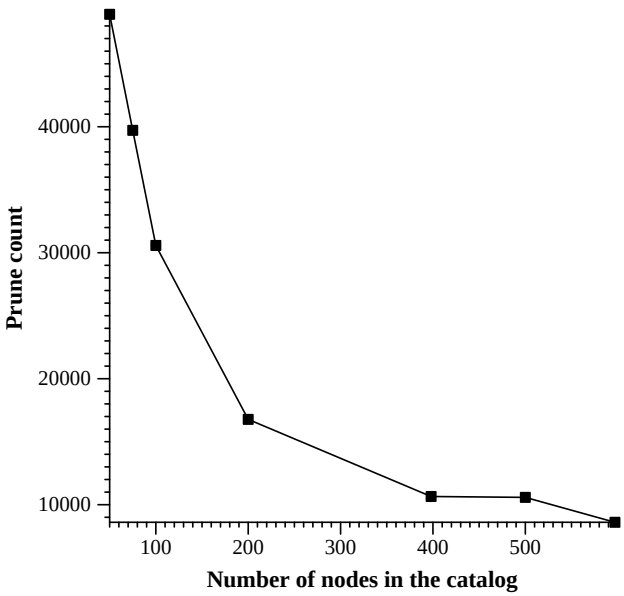


Figure 11.6: Variation of prune count with the number of nodes retained in the catalog.

11.6 Results for the Online Phase

In this section, we analyze the results of using our strategies from Section 11.3 for estimating selectivity. From Section 11.5.3, we observe that if we are allowed a catalog of size 0.5–1 Kbytes, we can store from between 100–200 nodes in the catalog. Of course, our errors in estimation will decrease with increasing number of nodes in the catalog. We also observed in Section 11.5.4 that the prune count for a catalog with 100 nodes is significantly greater than the prune count for a catalog with 200 nodes. When we compare the respective prune counts with the observation at the end of Section 11.4 on how many rows a positive single pattern appears in, it is obvious that the errors would be minimal when we use a catalog with 200 nodes. In particular, with a 200 node catalog, for positive single patterns, a strategy that returns the prune probability when a pattern does not match will estimate the selectivity of a non-matching single pattern to be 16K/200K = 0.08, while the true selectivity is expected to be approximately 0.05; the error is therefore small.

To really test our estimation strategies, and for us to be confident about their validity in environments outside the TPC-D benchmark, it is important that we test them against a catalog with a prune probability sufficiently larger than the expected selectivity of the base patterns. We therefore report our results in this section for when the catalog holds 100 nodes. We would like to emphasize that for the complexity of the problem, having a larger catalog size would help significantly; in particular, the errors for the experiments we report in this section will be minimal if we had used a 200-node catalog.

Evaluation Metric. We first present our results for matching positive single patterns in Section 11.6.1 and our results for matching positive double patterns in Section 11.6.2. For a positive pattern σ we measure the *relative error*, i.e., $(est(\sigma) - sel(\sigma))/sel(\sigma)$. Notice that the relative error can never be less than -100%. We plot the cumulative number of patterns along the y axis against the relative error on the x axis; a point (x, y) on the graph implies that y positive single patterns have relative error $\leq x\%$. To find out if a strategy is

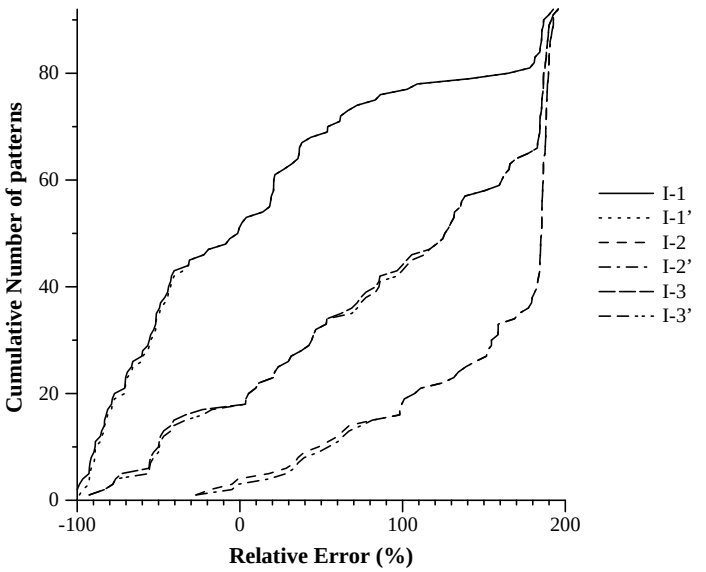


Figure 11.7: Performance of the Independence-based strategies for positive single patterns. The number of different query patterns is 92.

acceptable, we can draw vertical lines at the maximum and minimum acceptable error; the difference of the y values of a curve where these lines intersect it will give us the number of patterns with acceptable error. In particular, a "good" strategy has a *steeper slope* (not a larger value, necessarily) around $x = 0$.

We then look at the case of negative patterns in Section 11.6.3. In this case $sel(\sigma) = 0$. We therefore measure the *absolute error*; i.e., $est(\sigma)$, which is proportional to the number of rows of the column R that match the pattern. In this case, a "good" strategy has a large y value close to $x = 0$. In Section 11.6.4, we put together all our observations to evaluate the estimation strategies by defining a precise comparison metric.

11.6.1 Positive Single Patterns

Our results obtained by matching positive single patterns against the catalog are shown in Figures 11.7–11.10. Recall from Section 11.4 that positive single patterns have a selectivity of approximately 0.05.

We observe from Figure 11.7 that amongst the independence-based strategies, strategies I_1, I_1' perform best. Further, there is little difference in performance between the "primed" and "non-primed" strategies; in particular, strategies I_1 and I_1' are essentially identical. Strategies I_2 and I_3 tend to over-estimate selectivity, with strategy I_3 being a little more accurate than strategy I_2.

From Figure 11.8, we observe that the child estimation-based strategies perform well; strategy CE_2 has a particularly sharp slope around $x = 0$. From Figure 11.9, we observe that strategies DE_1 and DE_2 perform well, with strategy DE_1 having a slightly larger slope around $x = 0$.

Figure 11.10 plots the best strategies identified above. Strategy CE_2 stands out as being particularly good; strategies CE_3 and DE_1 are slightly better than strategy I_1. (Strategies I_3, I_3' and CE_1 are not much away from I_1 in performance; their curves are not plotted in Figure 11.10.)

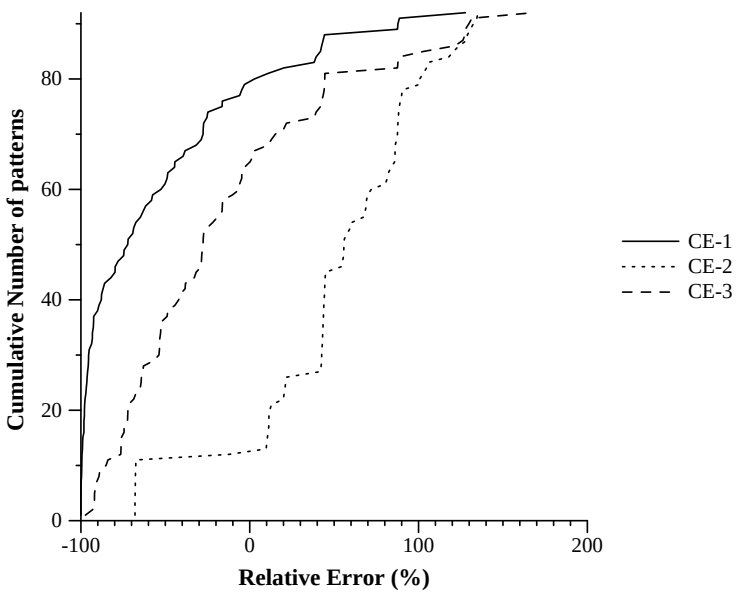


Figure 11.8: Performance of the Child Estimation-based strategies for positive single patterns. The number of different query patterns is 92.

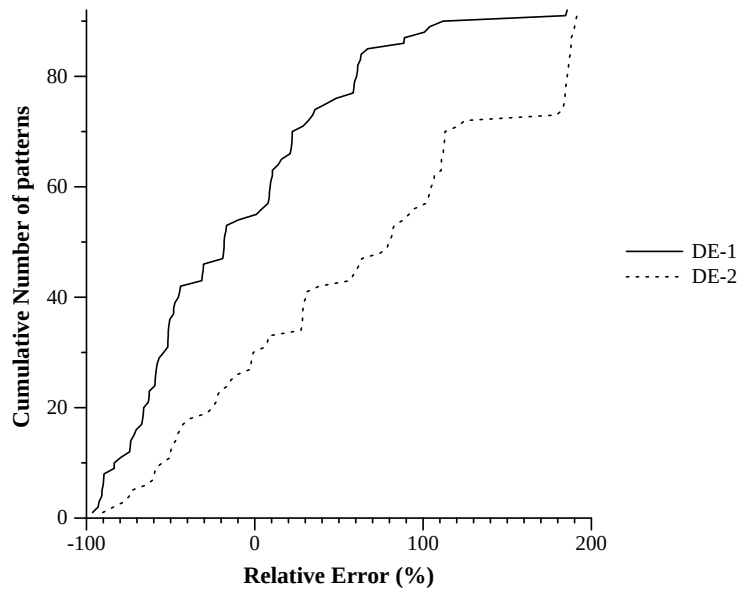


Figure 11.9: Performance of the Depth Estimation-based strategies for positive single patterns. The number of different query patterns is 92.

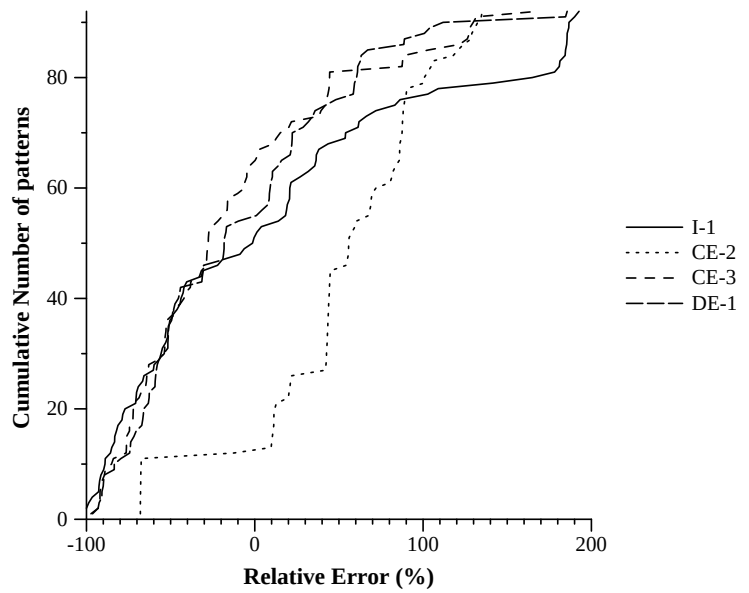


Figure 11.10: Graph of the best performing strategies for positive single patterns. The number of different query patterns is 92.

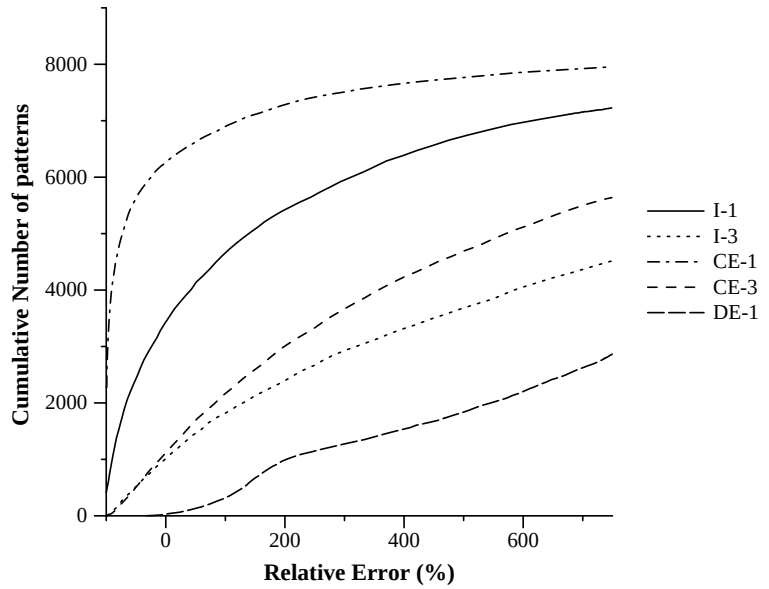


Figure 11.11: Graph of the best performing strategies for positive double patterns. The number of different query patterns is $92 \times 91 = 8372$.

11.6.2 Positive Double Patterns

Figure 11.11 plots for positive double patterns the performance of strategies that were identified as best for the positive single patterns. We crop the graphs at a relative error of 750%. Recall from Section 11.4 that there were a total of 8372 patterns The selectivity of the double patterns are very small, approximately 0.0005; i.e., a double pattern appeared in roughly 100 rows of P_NAME. A larger relative error in this case corresponds to a much

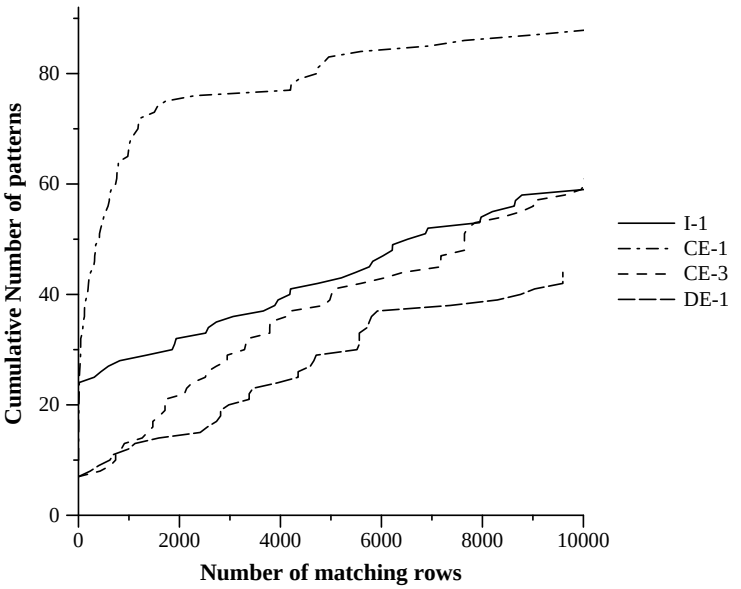


Figure 11.12: Graph of the best performing strategies for negative patterns. The x axis plots the number of matching rows (i.e., 200K, the number of rows in P_NAMES, times the selectivity). The patterns were obtained by introducing two errors into each of the positive single patterns. In this case, we want as many patterns as possible to have a selectivity close to 0.

smaller absolute error. For example, with positive double patterns, a relative error of 750% corresponds to an estimate that 850 rows out of 250K rows will match the specific positive double pattern, while in reality a 100 rows will match the pattern.

We observe from Figure 11.11 that strategies that perform well for the positive single patterns, i.e., strategies I_1 and CE_3, also perform well for the positive double patterns. (In addition, the trends shown by the strategies for the positive single patterns also hold for positive double patterns.) The notable exception is strategy CE_2; all the double patterns have a relative error greater than 2000% with strategy CE_2. This is not surprising since strategy CE_2 does not use the parsing of the pattern effectively. Interestingly, strategies I_3 and CE_1 perform reasonably well and strategy I_1' performs slightly better than strategy I_1 for positive double patterns. Strategy DE_1 seems to be on the border-line with respect to the errors it incurs being acceptable.

We observe that the relative performance of the strategies for positive double patterns is $I_1 > CE_3 > I_3 > CE_1 > DE_1$, where ">" implies "is better than."

11.6.3 Negative Patterns

As described in Section 11.4, we studied our strategies against three sets of negative patterns. For negative patterns, the selectivity is zero and we would like our estimation to be as close to zero as possible. In this section, we present our results for the negative patterns, where each negative pattern was obtained by introducing two errors into each positive single pattern. The relative performance of the strategies are qualitatively similar to the situation when one or three errors are introduced in the single patterns; the selectivity is closer to zero (i.e., better) when the pattern is formed by introducing more errors.

Figure 11.12 graphs the cumulative number of patterns against the estimation of selectivity. For the sake of simplicity, instead of using selectivity, we use a proportional measure, the number of rows that match the query pattern. Notice that unlike positive patterns, relative error does not make sense here.

In the case of negative patterns, we want the the curve to lie as close to the y-axis as possible. We plot curves for the three strategies that perform well for the single and double positive patterns, i.e., strategies I_1, CE_3, and DE_1, and one other strategy, CE_1, that performs particularly well for the negative patterns. We observe that these strategies perform well for negative patterns also, with strategy I_1 doing better than strategies CE_3 and DE_1. For negative patterns, strategy I_1 performs a little better than strategy I_1'.

11.6.4 Comparing the Strategies Globally

We have observed that some strategies perform particularly well for specific types of patterns (e.g., strategy CE_1 for negative patterns, and strategy CE_2 for positive single patterns). However, we want strategies that are consistently good over all types of patterns: positive single patterns, positive double patterns, and negative patterns. We have observed three strategies that consistently perform well for all types of patterns, namely strategies I_1 (or I_1'), CE_3, and DE_1. In this section, we develop a quantitative measure to evaluate the strategies.

Our goal is to assign a *grade* to each strategy based on certain intuitive notions we have of how much error is acceptable. Given a measure of acceptable error, we define the grade of the strategy as the percentage of patterns that fall within the acceptable error range. The acceptable error range would depend on the type of pattern. Other measures that take into account the entire distribution (i.e., more continuous measures that study degrees of acceptance) can also be considered.

For our study, we define a relative error between -75% and $+150\%$ as acceptable for positive single patterns, a relative error between -75% and 500% as acceptable for positive double patterns, and an absolute error, where the estimated number of rows is less than 5000, as acceptable for negative patterns. The logic behind these definitions is based on what we expect will be the action of the query optimizer with respect to the estimations, and is based on the real value of selectivity. For example, we allow a higher relative error as acceptable for positive double patterns, since this corresponds to a small absolute error, as explained in Section 11.6.2. An error of 500% for positive double patterns corresponds to an estimate that 600 rows of the P_NAME column match the pattern, while in fact only about 100 rows match the pattern. (Recall that there are 200K rows in P_NAME.) We are more lenient with negative patterns, since we expect them to occur less frequently. We are also less critical about under-estimating as opposed to over-estimating.

Table 11.2 presents the grades of the different algorithms for positive single patterns, positive double patterns, and negative patterns, where a negative pattern is obtained by introducing two errors into a single positive pattern. We conclude from Table 11.2 and our discussion from Sections 11.6.1–11.6.3 that strategies I_1, I_1', CE_1, CE_3, and DE_1 hold promise for good prediction of alphanumeric selectivity.

Pattern	I_1	I'_1	I_2	I'_2	I_3	I'_3	CE_1	CE_2	CE_3	DE_1	DE_2
positive single	64	65	28	28	57	57	49	100	82	85	75
positive double	60	63	0	0	41	41	34	0	53	22	11
negative	46	39	21	0	21	14	90	31	42	31	16

Table 11.2: Table of grades for the different algorithms. A grade of x means that the algorithm did "well" (as defined in Section 11.6.4) for x% of the queries.

	Catalog size	
Pattern	389 bytes	438 bytes
positive single	34	42
positive double	0	0
negative	100	100

Table 11.3: Grades for the "strawman" random sampling scheme. The 389 byte catalog stored 12 strings and the 438 byte catalog stored 14 strings. The catalog for the results presented in Table 11.2 used 386 bytes (or, as described in Section 11.5.3, the catalog used 436 bytes if an extra count was stored with each node).

11.7 A "Strawman" Random Sampling Scheme

It is instructive to compare the results presented in Section 11.6 against a simple random sampling strategy motivated by work in learning theory [Vapa]. The comparisons are especially interesting given that we know of no prior work in predicting alphanumeric selectivity.

A simple solution to the problem of estimating alphanumeric selectivity is to take a random sample of the strings in the column of the database and store as many as possible in the catalog, limited by the size of the catalog. When a query pattern is presented, the pattern is matched against the catalog, and the selectivity is estimated in the obvious way: If there are ℓ strings in the catalog and m of them match the query pattern, the selectivity is estimated to be m/ℓ. This approach will always correctly estimate the selectivity to be 0 for negative patterns.

In Table 11.3, we present the grades for the random sampling scheme using catalogs of size 389 and 438 respectively. (As described in Section 11.5.3, the catalog used for the results mentioned in Section 11.6 had a size of 386 bytes, or 436 bytes if an extra count was stored with each node.)

Comparing Tables 11.2 and 11.3 we observe that our recommended matching strategies from Section 11.6.4 outperform the random sampling strategy for positive single patterns and especially for positive double patterns. Strategy CE_1 does almost as well as the random sampling strategy for negative patterns, and does better for both positive single and positive double patterns.

The strategy presented above of using the randomly sampled strings is in a basic or "strawman" form. Modifications to this basic strategy, along the lines of the approaches in Section 11.3 could be used to improve the performance of the random sampling method.

11.8 Extensions

In this section we describe certain extensions to our method for estimating selectivity.

11.8.1 Non-Unit Patterns

We have observed techniques for predicting alphanumeric selectivity for unit patterns. Given a general pattern that begins with a wildcard, we can break it up into sub-patterns that are each unit patterns. For example, the pattern "$*\alpha*\beta*$" can be broken up into two unit sub-patterns "$*\alpha*$" and "$*\beta*$". We can use our strategies for estimating selectivity for each of these unit sub-patterns and put them together using ideas similar to the ones presented in Section 11.3; in particular, we can estimate the selectivity as the product of the selectivity of two unit sub-patterns by making an independence assumption.

If a pattern does not begin with a "$*$", the situation is somewhat different. We can, however, keep another count $count'$ with each node of the tree during stage 1 of the pre-processing phase which specifies the number of rows of the column that will match a query that does not start with a wildcard. Since such predicates are less likely in practice, it will be wasteful to store these new counts explicitly in the catalog. We can, however, use a small number of bits, say 4 bits, to store the approximate value of the fraction $count'/count$ at each node, and use this fraction and the value of $count$ in the online phase to get an approximation to $count'$.

11.8.2 Optimization With Extra Knowledge

It is important to note that the methods we have presented for creating the catalog, the strategies for matching, and our experimental results do not assume any knowledge of the distribution or type of data in the column of the database. As mentioned in Section 3.5 for the prefetching problem, and as is true with most systems scenarios, knowledge of the environment (in our case, information about the column) can be used effectively to further optimize our predictors.

For example, with the P_NAME column of the TPC-D benchmark that we used in our experimental studies, we could have performed many optimizations. We could have, for example, reduced the number of nodes in the catalog, by avoiding duplication using the following observation: The blank character appears in every string, and we expect that in the catalog, the tree below the node z whose corresponding string is the blank character can be eliminated totally, since the estimations there can be derived from the rest of the tree. (We expect the tree below node z to be a duplication of the tree obtained by pruning out the subtree rooted at node z.) We did not do this optimization in our experiments, since this is very data-specific.

Another optimization that could have been done during the online phase was to break up the query pattern into sub-patterns based on the length of the pattern. For example, the positive double patterns are on the average about twice the size of a single pattern. We could break a pattern that is "long" into sub-patterns and use strategy CE_2, for example, against each sub-pattern. Taking a product of the selectivities returned by strategy CE_2 will give a very good selectivity estimator for double patterns. Again, this was not done in our experiments because it is unclear whether such information will be available in the real world.

11.9 Discussion

In this chapter, we have performed the first known study of predicting alphanumeric selectivity. This scenario corresponds to predicting "how much" of an event is going to happen in the future. Predictors of selectivity have to operate under severe time and space constraints.

We have developed strategies for predicting selectivity based on suffix trees by drawing on our intuition of the close relationship between prediction and data compression. We have shown how to efficiently build a small catalog in the preprocessing phase, and how to match query patterns against such a catalog. Experimental evidence based on data from the TPC-D benchmark suggest that our techniques hold great promise for effective use in real database systems. It will be interesting to see if better strategies for matching against our catalog can be developed.

The database used for the study was derived from the TPC-D benchmark. All our experiments were performed with a slightly pessimistic view of the world; in other words, we strained our techniques beyond what might happen in real-life query processing of alphanumeric selectivity, and we tried not to tune the methods to the specifics of the TPC-D benchmark. Hence the results can be interpreted to be somewhat conservative.

Chapter 12

Conclusions

It is a test of true theories not only to account for but to predict phenomena.

—William Whewell (1840)

Performance with low response time is undoubtedly a very important desired feature of computing systems. Most response time issues reduce to the system having to make online decisions. Such decisions inherently involve an estimation of the future. Efficient prediction schemes to aid in this decision process significantly help in improving the performance of the system.

In this thesis, we have studied three important prediction scenarios that arise in online decision making in computer systems. Specifically, we have looked at the scenarios of *what* is going to happen, or *when* a specific event will take place, or *how much* of something is going to happen. We have developed algorithms for optimal online decision making by predicting or estimating the future. Our analyses are a blend of theoretically rigorous arguments and experimental evaluations. Our goal has been not just to predict but to do so efficiently with respect to computational resources.

We have developed efficient techniques for prefetching, the decision scenario requiring a prediction of "what" is going to happen, by investigating the novel idea of using data compression techniques for prediction (Chapters 2–5). We have shown the theoretical optimality of our techniques under general analytic models, and we have studied the practical aspects via simulation studies on sequences of page requests derived from CAD applications and the OO1 and OO7 benchmarks. Our simulations demonstrate significant improvements in the page fault rate achieved by our prefetchers over currently used demand-fetching algorithms and other proposed prefetchers.

We have studied the disk spindown problem, the decision scenario requiring a prediction of "when" an event is going to happen, in the context of mobile computing (Chapters 6–9). We have developed efficient adaptive disk spindown algorithms by modeling the problem using the rent-to-buy-framework, and developed provably optimal rent-to-buy strategies in probabilistic environments. We have studied fixed-threshold online algorithms, predictive algorithms, and adaptive strategies for disk spindown by simulation studies using disk access traces from a Macintosh Powerbook Duo (a typical portable machine) and the Hewlett-Packard 9000/845 personal workstation running HP-UX. We observed that in the disk spindown scenario predictive techniques did not help as much as they did in the prefetching problem. Threshold strategies were helpful in reducing the energy consumed by the disk.

Adaptive strategies were even better; in addition they helped in effectively balancing energy consumption and response time performance.

We have also performed the first known study of the problem of estimating non-numeric selectivity in relational databases, the decision scenario requiring a prediction of "how much" is going to happen (Chapters 10–11). We exploited our intuitive understanding of the close relationship between data compression and prediction to use a suffix-tree based structure for the prediction of alphanumeric selectivity. Our techniques cope with stringent memory and time limitations to perform effective estimation of selectivity. Experiments using the TPC-D benchmark have given very encouraging results.

Two important themes emerge from our work. The first relates to the theoretical analysis of online systems algorithms. Traditional competitive analysis compares online strategies against optimal offline algorithms. Comparison against offline algorithms is unduly harsh on the online method, since perfect knowledge of the future provides an offline algorithm with tremendous power. From the practical view-point, theoretical studies of online algorithms using competitive analysis lead to pessimistic bounds on their performance. We have analyzed our algorithms for prefetching and rent-to-buy against optimal *online* algorithms. Specifically, we have performed analysis by modeling input behavior using powerful probabilistic models, or in the worst-case by restricting the power of the adversary. This is essential for prediction problems, since offline algorithms are prefect predictors. The optimality results of our analysis have translated into practical algorithms for prediction. It is appropriate that comparison of online algorithms be done against the best online algorithm, rather than the best offline algorithm, to better estimate their practical merit.

An important unifying contribution of our work is the common theme we have observed across the different prediction scenarios of *using data compression techniques for prediction.* We have had very good results from using this approach for the prefetching problem, mixed results from using an interpretation of this approach for the disk spindown problem, and encouraging results from using this idea for the problem of predicting alphanumeric selectivity. Specific problems arising from the systems context in which the predictors got used created interesting issues that needed to be tackled individually. We conclude that techniques based on data compression provide effective methods for prediction in computer systems.

References

[AbW] N. Abe and M. Warmuth, "On the Computational Complexity of Approximating Distributions by Probabilistic Automata," UCSC, UCSC-CRL-90-63, December 1990.

[AlV] D. Aldous and U. Vazirani, "A Markovian Extension of Valiant's Learning Model," *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science* (October 1990), 392–396.

[Ale] T. Alexander, personal communication.

[AlK] T. Alexander and G. Kedem, "Design and Evaluation of a Distributed Cache Architecture with Prediction," Duke University Technical Report, DUKE–CS–1994–05.

[AmM] Y. Amit and M. Miller, "Large Deviations for Coding Markov Chains and Gibbs Random Fields," Washington University, Technical Report, 1990.

[App] Applelink, "Developer Support-Developer Talk," AppleLink bulletin board, October 1993.

[ASW] M. M. Astrahan, M. Schkolnick, and K.-Y. Whang, "Approximating the Number of Unique Values of an Attribute Without Sorting ," *Information Systems* 12 (1987), 11–15.

[BAD] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, "Non-volatile Memory for Fast, Reliable File Systems," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1992), 10–22, ACM.

[Bel] L. A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Systems Journal* 5 (1966), 78–101.

[BCW] T. C. Bell, J. C. Cleary, and I. H. Witten, *Text Compression*, Prentice Hall Advanced Reference Series, 1990.

[Bla] D. Blackwell, "An Analog to the Minimax Theorem for Vector Payoffs," *Pacific Journal of Mathematics* 6 (1956), 1–8.

[BEHa] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, "Occam's Razor," *Information Processing Letters* 24 (1987).

[BEHb] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, "Learnability and the Vapnik Chervonenkis Dimension," *Journal of the ACM* (October 1989).

[BoP] R. Board and L. Pitt, "On the Necessity of Occam Algorithms," *Proceedings of the 22nd Annual ACM Symposium on Theory of Computation* (May 1990), 54–63.

[BIR] A. Borodin, S. Irani, P. Raghavan, and B. Schieber, "Competitive Paging with Locality of Reference," *Proceedings of the 23rd Annual ACM Symposium on Theory of Computation* (May 1991).

[Bra] J. T. Brady, "A theory of productivity in the creative process," *IEEE CG&A* (May 1986), 25–34.

[BuB] S. Bunton and G. Borriello, "Practical Dictionary Management for Hardware Data Compression," Department of Computer Science, University of Washington, FR-35, 1991.

[CFK] P. Cao, E. W. Felten, A. Karlin, and K. Li, "A Study of Integrated Prefetching and Caching Strategies," *Proceedings of the 1995 SIGMETRICS Conference*, to appear.

[CDN] M. J. Carey, D. J. DeWitt, and J. F. Naughton, "The DEC OO7 Benchmark," *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (May 1993).

[CaS] R. G. G. Cattell and J. Skeen, "Object Operations Benchmark," *ACM Transactions on Database Systems* 17 (March 1992), 1–31.

[ChB] T. F. Chen and J. L. Baer, "Reducing Memory Latency via Non-blocking and Prefetching Caches," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1992).

[Con] Connectix Corporation, "CPU Connectix PowerBook Utilities Version 2.0 Addendum," San Mateo, CA, April 1993.

[Cor] Maxtor Corporation, "The MXL–105–III," 1993.

[Cov] T. M. Cover, "Behavior of Predictors of Binary Sequences," *Proceedings of the 4th Prague Conference on Information Theory, Statistical Decision Functions, Random Processes* (1967), 263–272.

[CoS] T. M. Cover and A. Shenhar, "Compound Bayes Predictors with Apparent Markov Structure," *IEEE Transactions on System Man and Cybernetics* SMC-7 (June 1977), 421–424.

[CoT] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, Wiley, 1991.

[CKV] K. Curewitz, P. Krishnan, and J. S. Vitter, "Practical Prefetching via Data Compression," *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (May 1993), 257–266.

[Dat] C. Date, *An Introduction to Database Systems*, Addison-Wesley, 1981.

[Del] Dell Computer Corporation, "Dell System 320SLi User's Guide," June 1992.

[Den] P. J. Denning, "Working Sets Past and Present," *IEEE Transactions on Software Engg.,* SE-6 (1980), 64–84.

[DKB] F. Douglis, P. Krishnan, and B. Bershad, "Adaptive Disk Spindown Policies for Mobile Computers," *Proceedings of the Second USENIX Symposium on Mobile and Location Independent Computing.*

[DKM] F. Douglis, P. Krishnan, and B. Marsh, "Thwarting the Power Hungry Disk," *Proceedings of the 1994 Winter USENIX Conference* (January 1994).

[FMG] M. Feder, N. Merhav, and M. Gutman, "Universal Prediction of Individual Sequences," *IEEE Transactions on Information Theory* IT-38 (July 1992), 1258–1270.

[FiG] E. R. Fiala and D. H. Greene, "Data Compression with Finite Windows," *Communications of the ACM* 32 (April 1989), 490–505.

[FKL] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young, "On Competitive Algorithms for Paging Problems," *Journal of Algorithms* 12 (1991), 685–699.

[FlM] P. Flajolet and G. N. Martin, "Probabilistic Counting Algorithms for Database Applications," *Journal of Computer and Systems Sciences* 31 (1985), 182–209.

[Gal] R. G. Gallager, *Information Theory and Reliable Communication*, Wiley, 1968.

[GBS] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes, "Idleness is not Sloth," *Proceedings of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems* (January, 1995), 201–212.

[GKP] R. L. Graham, D. E. Knuth, and O. Patashnik, *Concrete Mathematics*, Addison-Wesley Publishing Company, 1989.

[GrR] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, Inc., 1993.

[Gre] P. Greenawalt, "Modeling Power Management for Hard Disks," *Proceedings of the Symposium on Modeling and Simulation of Computer Telecommunication Systems* (1994).

[HMM] T. Hagerup, K. Mehlhorn, and I. Munro, "Optimal Algorithms for Generating Discrete Random Variables with Changing Distributions," Max Planck Institute, Technical Report MPI–I–92–145, October 15, 1992.

[Han] J. F. Hannan, "Approximation to Bayes Risk in Repeated Plays," *Contributions to the Theory of Games, Vol 3, Annals of Mathematical Studies* (1957), 97–139.

[Hewa] Hewlett Packard, "Omnibook 300 Operating Guide, 1st edition," Corvallis, OR, April 1993.

[Hewb] Hewlett Packard, "Kittyhawk Power Management Modes," April 1993, Internal Document.

[HoV] P. G. Howard and J. S. Vitter, "Analysis of Arithmetic Coding for Data Compression," *Information Processing and Management* 28 (1992), 749–763, invited paper in Special Issue on Data Compression for Images and Texts.

[IKP] S. Irani, A. R. Karlin, and S. Phillips, "Strongly Competitive Algorithms for Paging with Locality of Reference," *Proceedings of the 3rd Annual ACM-SIAM Symposium of Discrete Algorithms* (January 1992).

[Iye] B. Iyer, IBM, Santa Teresa, personal communication.

[KLM] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki, "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor ," *Proceedings of the 1991 ACM Symposium on Operating System Principles* (1991), 41–55.

[KMM] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki, "Competitive Randomized Algorithms for Non-Uniform Problems," *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms* (1990), 301–309.

[KPR] A. R. Karlin, S. J. Phillips, and P. Raghavan, "Markov Paging," *Proceedings of the 33rd Annual IEEE Conference on Foundations of Computer Science* (October 1992), 208–217.

[KeS] M. J. Kearns and R. E. Schapire, "Efficient Distribution-Free Learning of Probabilistic Concepts," *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science* (October 1990), 382–391.

[KLP] S. Keshav, C. Lund, S. J. Phillips, N. Reingold, and H. Saran, "An Empirical Evaluation of Virtual Circuit Holding Time Policies in IP-over-ATM Networks," *Proceedings of INFOCOM '95*, to appear.

[Knua] D. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[Knub] D. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Addison-Wesley, Reading, MA, second edition 1981.

[KoE] D. F. Kotz and C. S. Ellis, "Prefetching in File Systems for MIMD Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems* 1 (April 1990), 218–230.

[KLV] P. Krishnan, P. M. Long, and J. S. Vitter, "Adaptive Disk Spindown via Optimal Rent-to-Buy in Probabilistic Environments," Duke University Technical Report, CS–1995–08, March, 1995, to appear in *Machine Learning: Proceedings of the Twelfth International Conference*, Armand Prieditis and Stuart Russell, eds., Morgan Kaufmann Publishers, San Fransisco, CA, 1995.

[KrV] P. Krishnan and J. S. Vitter, "Optimal Prediction for Prefetching in the Worst Case," *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms* (January 1994), Also appears as Duke University Technical Report CS-1993-26..

[KPR] G. H. Kuenning, G. J. Popek, and P. L. Reiher, "An Analysis of Trace Data for Predictive File Caching in Mobile Computing," *Proceedings of the 1994 Summer USENIX* .

[Lai] P. Laird, "Discrete Sequence Prediction and its Applications," AI Research Branch, NASA Ames Research Center, manuscript, 1992.

[Lana] G. G. Langdon, "A note on the Ziv-Lempel model for compressing individual sequences," *IEEE Transactions on Information Theory* 29 (March 1983), 284–287.

[Lanb] G. G. Langdon, "An Introduction to Arithmetic Coding," *IBM J. Res. Develop.* 28 (March 1984), 135–149.

[LeZ] A. Lempel and J. Ziv, "On the Complexity of Finite Sequences," *IEEE Transactions on Information Theory* 22 (January 1976).

[LKH] K. Li, R. Kumpf, P. Horton, and T. Anderson, "A Quantitative Analysis of Disk Drive Power Management in Portable Computers," *Proceedings of the 1994 Winter USENIX* (January 1994).

[LPR] C. Lund, S. Phillips, and N. Reingold, "IP over Connection-Oriented Networks and Distributed Paging," *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science* (November 1994), 424–434.

[MDK] B. Marsh, F. Douglis, and P. Krishnan, "Flash Memory File Caching for Mobile Computers," *Proceedings of the 27th IEEE Hawaii Conference on System Sciences* (January 1994).

[MVN] Y. Matias, J. S. Vitter, and W. C. Ni, "Dynamic Generation of Discrete Random Variates," *Proceedings of the 4th Annual SIAM/ACM Symposium on Discrete Algorithms* (January 1993).

[McC] E. M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," *Journal of the ACM* 23 (1976), 262–272.

[McS] L. A. McGeoch and D. D. Sleator, "A Strongly Competitive Randomized Paging Algorithm," Carnegie-Mellon University, CS–89–122, March 1989.

[MeF] N. Merhav and M. Feder, "Universal Sequential Learning and Decision from Individual Data Sequences," *Proceedings of the 5th ACM Workshop on Computational Learning Theory* (July 1992), .

[Mor] D. R. Morrison, "PATRICIA–A Practical Algorithm to Retrieve Information Coded in Alphanumeric," *Journal of the ACM* 15 (1968), 514–534.

[MLG] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1992).

[Nat] S. Natarajan, "Large Deviations, Hypothesis Testing, and Source Coding for Finite Markov Sources ," *IEEE Transactions on Information Theory* 31 (1985), 360–365.

[OuD] J. Ousterhout and F. Douglis, "Beating the I/O Bottleneck: A Case for Log-Structured File Systems," U. C. Berkeley, UCB/CSD 88/467, October 1988.

[Pac] Hewlett Packard, "Kittyhawk HP C3013A/C3014A Personal Storage Modules Technical Reference Manual," March 1993, HP Part No. 5961-4343.

[PaZa] M. Palmer and S. Zdonik, "Predictive Caching," Brown University, CS–90–29, November 1990.

[PaZb] M. Palmer and S. Zdonik, "Fido: A Cache that Learns to Fetch," *Proceedings of the 1991 International Conference on Very Large Databases* (September 1991).

[PGS] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan, "A Status Report on Research in Transparent Informed Prefetching," *ACM Operating Systems Review* 27 (April 1993), 21–34.

[Pol] D. Pollard, *Convergence of Stochastic Processes*, Springer-Verlag, 1984.

[Qua] Quantum, "Go•Drive 60/120S Product Manual," May 1992.

[RoL] A. Rogers and K. Li, "Software Support for Speculative Loads," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1992).

[RuWa] C. Ruemmler and J. Wilkes, "UNIX Disk Access Patterns," *Proceedings of the 1993 Winter USENIX Conference* (January 1993), 405–420.

[RuWb] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *IEEE Computer* 27 (March 1994), 17–28.

[Sal] K. Salem, "Adaptive Prefetching for Disk Buffers," CESDIS, Goddard Space Flight Center, TR–91–64, January 1991.

[SaK] H. Saran and S. Keshav, "An Empirical Evaluation of Virtual Circuit Holding Times in IP-over-ATM networks," *Proceedings of INFOCOM '94* (June 1994).

[SAC] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of the 1979 ACM SIGMOD Conference*, 23–34.

[ShT] G. S. Shedler and C. Tung, "Locality in Page Reference Strings," *SIAM Journal on Computing* 1 (1972), 218–241.

[SlT] D. D. Sleator and R. E. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Communications of the ACM* 28 (February 1985), 202–208.

[SoC] I. Song and Y. Cho, "Page Prefetching Based on Fault History," *Proceedings of the USENIX Mach III Symposium*, 203–213.

[Stoa] M. Stonebraker, "Operating System Support for Database Management," *Communications of the ACM* 24 (July 1981), 412–418.

[Stob] J. A. Storer, *Data Compression Methods and Theory*, Computer Science Press, 1988.

[TPC] Transaction Processing Performance Council, TPC, "TPC Benchmark^{TM} D (Decision Support), Working Draft 6.0," 1993, F. Raab (editor) .

[Tri] K. S. Trivedi, "An Analysis of Prepaging," *Computing* 22 (1979), 191–210.

[TsN] M. M. Tsangaris and J. F. Naughton, "On the Performance of Object Clustering Techniques," *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (June 1992), 144–153, Also appears as University of Wisconsin Madison Technical Report number 1090-1992.

[Ull] J. D. Ullman, *Principles of Database Systems*, Computer Science Press, 1988.

[Vapa] V. N. Vapnik, *Estimation of Dependencies based on Empirical Data*, Springer Verlag, 1982.

[Vapb] V. N. Vapnik, "Inductive principles of the search for empirical dependences (methods based on weak convergence of probability measures)," *Proceedings of the 1989 Workshop on Computational Learning Theory* (1989).

[VaC] V. N. Vapnik and A. Y. Chervonenkis, "On the Uniform Convergence of Relative Frequencies of Events to their Probabilities," *Theoretical Probability and Its Applications* 16 (1971), 264–280.

[ViK] J. S. Vitter and P. Krishnan, "Optimal Prefetching via Data Compression," *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science* (October 1991), 121–130, Also appears as Brown University Technical Report No. CS–91–46.

[Wei] P. Weiner, "Linear Pattern Matching Algorithms," *Proceedings of the IEEE 14th Annual Symposium on Switching and Automata Theory* (October, 1973), 1–11.

[WVT] K.Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A Linear-Time Probabilistic Counting Algorithm for Database Applications," *ACM Transactions on Database Systems* 15 (June 1990), 208–229.

[Wil] J. Wilkes, "Predictive Power Conservation," Hewlett Packard Laboratories Technical Report HPL-CSP-92-5, , February, 1994.

[WNC] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic Coding for Data Compression," *Communications of the ACM* 30 (June 1987), 520–540.

[Zen] Zenith Data Systems, Groupe Bull, "MastersPort 386L/386Le Owners Manual," 1991.

[ZiLa] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory* 23 (May 1977).

[ZiLb] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Transactions on Information Theory* 24 (September 1978), 530–536.