DETECTING DIGITAL COPYRIGHT VIOLATIONS ON THE INTERNET

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Narayanan Shivakumar
August 1999

© Copyright 1999 by Narayanan Shivakumar
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Professor Hector Garcia-Molina
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Professor Jeffrey D. Ullman

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Professor Jennifer Widom

Approved for the University Committee on Graduate Studies:

Abstract

Cyber-pirates are offering music CDs, video clips and books on the web in digital format to a large audience at virtually no cost. Content publishers such as Disney and Sony Records therefore expect to lose several billions of dollars in copyright revenues over the next few years. To address this problem, we propose building a copy detection system (CDS), where content publishers will register their valuable digital content. The CDS then crawls the web, compares the web content to the registered content, and notifies the content owners of illegal copies. In this dissertation, we discuss how to build such a system so it is accurate, scalable (e.g., to hundreds of gigabytes of data, or millions of web pages) and resilient to "attacks" (e.g., copying a resampled audio clip) from cyber-pirates. We also discuss two prototype CDS systems we have built as "proofs of concept:" (1) SCAM (Stanford Copy Analysis Mechanism) for text documents, and (2) DECEIVE (Detecting Copies of Internet Video) for video sequences.

Acknowledgements

Hector was the coolest advisor I could have hoped for. I would like to thank him for showing me how to have fun while doing research, how to identify interesting problems, and also for shielding me from the many non-technical problems that I kept stumbling into on a regular basis.

In addition to Hector, I had the privilege to work with, and learn from many researchers during the course of my dissertation work. Jennifer Widom was a second advisor to me in my first few years when I was dabbling in everything except my dissertation. I also learned a lot from the coauthors of my papers (a few of which are part of this dissertation) including Chandra Chekuri, Junghoo Cho, Min Fang, Luis Gravano, Piotr Indyk, Giridharan Iyengar, Rajeev Motwani and Jeff Ullman.

Thanks to Professor Jason Cong (my undergraduate advisor at UCLA) for laying my initial foundations in research before I came to Stanford. Thanks to Professors Jeff Ullman, Jennifer Widom and Gio Wiederhold for creating the best database group this side of the galaxy, and also for being part of my defense and reading committees. Without Andy Kacsmar and Marianne Siroker secretly running the DBgroup, nothing would have been possible.

My UCLA friends are the greatest. In particular, I have to mention Sheng Kong. When she moved to the Bay Area in '96, I knew she would delay my thesis by at least a factor of two. Though she succeeded, I ain't complaining (except for those ridiculous Beanie Baby hunts). Wilburt Labio and I have worked on way too many projects together at UCLA and subsequently at Stanford. Hope our future ventures are even more successful! Hui Cheng, Ali Niknejad, Giao Nguyen, Thomas Edward Schirmer and Marlene Wan helped me retain some semblance of sanity.

The list of cool Stanford folks is too long. Suresh Venkat, Piotr Indyk, Vasilis Vassalos and Ashish Goel specialized in generating interesting discussion topics after 2 AM, ranging from "parameterized Egyptian mummies" to "aliens with very strong arguments in their

favor." Also many thanks to Piotr and Suresh for making me a more "enlightened" systems person by showing me that theory, when used judiciously, could be incredibly useful for building systems. Arturo Crespo, Jan Jannink, Svetlozar Nestorov, Orkut Buyukokkten, Junghoo Cho, Roy Goldman, Jason McHugh, and Ramana Yerneni kept things interesting. I am grateful to Sergey Brin and Larry Page for (1) feeding me with obscene amounts of web data on a regular basis from Google to test out my algorithms, and (2) for fixing many bugs after 2 AM when they were at Stanford. Kathirgamar Aingaran, Chandra Chekuri, Naren Gokul, Arvind Parthasarathy, Shankar Govindaraju, Aris Gionis and Sudipto Guha added another year to graduation by being ever enthusiastic about playing tennis, squash, chess, cricket and foozball.

My parents gave me incredible support throughout the years. My brother (KK) and my sister-in-law (Aruna) were always happy to have long telephonic conversations with me on topics from algorithms and databases, to research and life. Amrita, my cute and precocious six-month old niece, has already taught me a lot – nothing matters more than drinking lots of milk, smiling a lot, and sleeping whenever you want to.

Finally, I would like to thank the Academy, Beavis and Austin Powers. During the makings of this dissertation, no animals were intentionally harmed.

This dissertation is dedicated to
my parents Radha and P.K. Narayanan, and
my brother Dr. Narayanan Krishnakumar (KK).

Contents

Abstract **v**

Acknowledgements **vii**

1 Introduction **1**

1.1 Recent approaches to combat cyber-piracy 2

1.1.1 Copy prevention 2

1.1.2 Copy detection 3

1.2 Protecting copyright in the future 6

1.3 Architecture of a Copy Detection System 6

1.4 Challenges in building a Copy Detection System 8

1.5 Contributions and structure of dissertation 9

1.6 Brief survey of related work 12

1.7 Summary 14

2 Features for a Text CDS **15**

2.1 Introduction 15

2.2 Attacks from a cyber-pirate 15

2.3 Possible features and similarity functions 16

2.3.1 Chunking based 16

2.3.2 Information Retrieval based 16

2.4 Experiments 19

2.4.1 Human classification 20

2.4.2 Data sets 21

2.4.3 Training similarity measures 22

2.4.4 Evaluating similarity measures 23
2.5 Conclusion 25

3 Executing Text CDS Operations **27**
3.1 Introduction 27
3.2 Implementing the FIND operation 28
3.3 Implementing the FIND-ALL operation 30
3.3.1 Generalizing FIND-ALL operation to *Icebergs* 32
3.4 Techniques for thresholding 33
3.4.1 A Sampling-Based Algorithm (SCALED-SAMPLING) 35
3.4.2 Coarse counting by bucketizing elements (COARSE-COUNT) 35
3.5 HYBRID techniques 36
3.5.1 DEFER-COUNT Algorithm 36
3.5.2 MULTI-LEVEL Algorithm 37
3.5.3 MULTI-STAGE Algorithm 38
3.6 Optimizing HYBRID with MULTIBUCKET algorithms 39
3.6.1 Single scan DEFER-COUNT with multiple hash functions (UNISCAN) 40
3.6.2 Multiple scan DEFER-COUNT with multiple hash functions (MULTISCAN) 40
3.6.3 MULTISCAN with shared bitmaps (MULTISCAN-SHARED) 41
3.6.4 Variant of MULTISCAN-SHARED (MULTISCAN-SHARED2) . . . 42
3.7 Extending iceberg algorithms 42
3.8 Case studies 43
3.8.1 Summary 55
3.9 Conclusion 57

4 Running Text CDS on the Web **59**
4.1 Introduction 59
4.2 Automatic detection of mirrored collections 60
4.3 Similarity of web pages 64
4.4 Similarity of collections 65
4.5 Defining similar clusters 69
4.6 Computing similar clusters 69
4.6.1 Growth strategy 72

4.6.2 Implementing the cluster growing algorithm . . . 73
4.7 Quality of similarity measures . . . 75
4.8 Performance experiments . . . 78
4.8.1 Resources wasted while crawling . . . 78
4.8.2 Improving search engine result presentation . . . 83
4.9 Conclusion . . . 84

5 Discovery-based Plagiarism Detection 89
5.1 Introduction . . . 89
5.1.1 History of plagiarism case . . . 89
5.1.2 Automatic plagiarism detection . . . 91
5.1.3 Summary of our methodology . . . 92
5.2 RFM measure . . . 94
5.3 The PROBE-CDS information about the databases . . . 95
5.4 The conservative approach . . . 97
5.5 The liberal approach . . . 100
5.5.1 Counting only rare words . . . 100
5.5.2 Using probabilities . . . 102
5.6 Searching the databases with potential copies . . . 103
5.7 Experiments . . . 106
5.7.1 Results for extraction techniques . . . 113
5.8 Conclusion . . . 116

6 Building a Video Comparator 117
6.1 Introduction . . . 117
6.2 Attacks from a cyber-pirate . . . 117
6.3 Feature extraction . . . 118
6.3.1 Using spatial signatures . . . 118
6.3.2 Using temporal signatures . . . 119
6.4 Similarity measure based on shot transitions . . . 121
6.4.1 Distance measure between video clips . . . 123
6.4.2 Fuzzy distance measure . . . 125
6.5 Implementing the FIND operation . . . 127
6.6 Putting it all together . . . 131

6.6.1 Handling video clipping . . . 131
6.6.2 Executing FIND and FIND-ALL operations . . . 132
6.7 Experiments . . . 132
6.7.1 Data set . . . 132
6.7.2 Computing shot transitions . . . 133
6.7.3 Quality of similarity measure . . . 134
6.7.4 Timing results . . . 136
6.8 Conclusion . . . 137

7 Approximate Query processing **139**
7.1 Introduction . . . 139
7.1.1 Other motivating applications . . . 140
7.2 Characterizing predicates . . . 141
7.3 Space of query plans . . . 143
7.3.1 Minimization measures . . . 147
7.4 General query optimization . . . 147
7.5 Optimizing `Select` queries for *MIN-EXP* measure . . . 151
7.5.1 Modeling predicate dependencies . . . 151
7.5.2 Propagating characteristics in a simple `Select` query . . . 152
7.5.3 Conjunctive filters . . . 154
7.5.4 Conjunctive filters computable in polynomial time . . . 156
7.6 Optimizing SPJ queries for *MIN-EXP* measure . . . 159
7.7 Minimizing (p, n)-*MIN-EXP* . . . 160
7.8 Experiments . . . 162
7.9 Conclusion . . . 165

8 Indexing Temporal Repositories **167**
8.1 Introduction . . . 167
8.2 Preliminaries . . . 172
8.2.1 Update techniques . . . 174
8.2.2 Operations on a Wave Index . . . 175
8.3 Building simple wave indices . . . 177
8.3.1 Deletion *(DEL)* . . . 177
8.3.2 Reindexing (*REINDEX*) . . . 178

8.3.3 Wait and Throw Away (WATA) 178
8.4 Enhancing simple wave indices with temporary indices 181
8.4.1 Improved reindexing ($REINDEX^{+}$) 181
8.4.2 Further improved reindexing ($REINDEX^{++}$) 182
8.4.3 Reindex and Throw Away (RATA) 185
8.5 Analytic comparison of wave indexing schemes 186
8.6 Case studies 192
8.7 Conclusion 203

9 Conclusions and Future Work **205**
9.1 Future Work 206

10 Appendix A: Wave Indexing Algorithms **209**

11 Appendix B: Efficacy of WATA* **217**

Bibliography **221**

List of Tables

2.1 Accuracy of similarity measures. 23

4.1 Storage and time costs for computing trivial similar clusters. 79
4.2 Popular web collections. 82

5.1 Summary of the PROBE-CDS techniques, where $k < 100$. (When $k = 100$, $Upper_{Range}$ and $Upper_{Ratio}$ coincide with Sum_{Range} and Sum_{Ratio}, respectively.) 103

6.1 Timing results. 137

7.1 Characteristics of example predicates. 143
7.2 Characteristics of filters constructed with LI approximate predicates. 152
7.3 Characteristics of filters constructed with LCI approximate predicates. . . . 152
7.4 Minimization functions for *MIN-EXP* measure. 157
7.5 Query plans for given n (LI Tests). 163
7.6 Query plans for given n (LCI Tests). 163

8.1 Deletion based index maintenance ($W = 10$). 169
8.2 Reindexing based index maintenance ($W = 10, n = 2$). 169
8.3 WATA based index transitions ($W = 10, n = 4$). 170
8.4 Another example of index transitions based on WATA (W = 10, n = 4). . . 180
8.5 Example of index transitions in $REINDEX^{+}$ (W = 10, n = 2). 183
8.6 Example of index transitions in $REINDEX^{++}$ (W = 10, n = 2). 184
8.7 Example of index transitions in RATA 185
8.8 Space utilization of wave indices that use simple shadow updating ($X = \frac{W}{n}, Y = \frac{W-1}{n-1}$). 190

8.9 Query performance of wave indices that use simple shadow updating. 190
8.10 Maintenance performance of wave indices that use simple shadow updating. 191
8.11 Maintenance performance of wave indexing techniques that use packed shadow updating. 192
8.12 Parameter values chosen in case study. 193

List of Figures

1.1 A Copy Detection System (CDS). 4

3.1 Data structures for FIND operation. 28
3.2 A graphical view of terminology. 33
3.3 Alternate HYBRID techniques to combine sampling and coarse-counting. . 37
3.4 Comparing MULTISCAN versus MULTISCAN-SHARED. 41
3.5 $|F|$ as memory varies ($T = 500$). 46
3.6 Total time as memory varies ($T = 500$). 47
3.7 $|F|$ as threshold varies ($M = 20$ MB). 48
3.8 Total time as threshold varies ($M = 20$ MB). 49
3.9 Frequency-rank curves for different chunkings. 51
3.10 Result sizes for different thresholds. 52
3.11 Performance of MULTISCAN-SHARED2/D with k ($T = 1000, m = 1\%$ of n). 54
3.12 Performance of MULTISCAN-SHARED2/D with m ($T = 1000, k = 2$). . . . 54
3.13 Performance of algorithms with M for DocumentOverlap query for $C = 1$. . 56

4.1 Mirrored document collections. 61
4.2 Example pairs of collections. 66
4.3 One possible similar cluster. 70
4.4 Another possible similar cluster. 70
4.5 Computing similar clusters from a set of web pages. 71
4.6 Growing similar clusters. 74
4.7 Join-based construction of *LinkSummary* table 75
4.8 Cluster growing algorithm. 76
4.9 Example of "break point." 77
4.10 Example of partial mirrors. 78

4.11 Document replication on 25 million web pages. 80
4.12 Distribution of maximal clusters. 81
4.13 Search results for "XML databases" from Google search engine. 85
4.14 Rolled up search results for "XML databases". 86

5.1 Example abstract "written" by Mr. X. 91
5.2 Closest matching abstract from INSPEC according to SCAM. 92
5.3 The percentage of the 50 databases that are searched as a function of the adjusted similarity threshold T^k (*Registered* suspicious documents; Sum_{Ratio} strategy; $T = 1$). 106
5.4 The average recall as a function of the adjusted similarity threshold T^k (*Registered* suspicious documents; Sum_{Ratio} strategy; $T = 1$). 107
5.5 The average precision as a function of the adjusted similarity threshold T^k (*Registered* suspicious documents; Sum_{Ratio} strategy; $T = 1$). 107
5.6 The percentage of the 50 databases that are searched as a function of the adjusted similarity threshold T^k (*Disjoint* suspicious documents; Sum_{Ratio} strategy; $T = 1$). 108
5.7 The percentage of the 50 databases that are searched as a function of the CDS threshold T (*Registered* suspicious documents; $k = 10$; $T^k = 0.05 \cdot T$). 111
5.8 The average recall as a function of the CDS threshold T (*Registered* suspicious documents; $k = 10$; $T^k = 0.05 \cdot T$). 112
5.9 The average precision as a function of the CDS threshold T (*Registered* suspicious documents; $k = 10$; $T^k = 0.05 \cdot T$). 112
5.10 The percentage of the 50 databases that are searched as a function of the CDS threshold T (*Disjoint* suspicious documents; $k = 10$; $T^k = 0.05 \cdot T$). . . 113
5.11 The average number of times that PROBE-CDS (incorrectly) chooses to search the (growing) database, as a function of the size of the database (*Disjoint* suspicious documents; Sum_{Ratio} strategy). 114
5.12 The percentage of words of the suspicious documents that are included in the query to extract the potential copies from the databases (*Registered* suspicious documents; *Ratio* strategy). 115

5.13 Average selectivity of the queries used to extract the potential copies from the databases, as a function of the CDS threshold T (*Registered* suspicious documents; *Ratio* strategy). 115

6.1 Similar video sequences. 122
6.2 Contribution per matching shot transition. 126
6.3 Matching shot transitions. 126
6.4 Fuzzified timing sequence. 128
6.5 Hamming embedding of timing sequence. 129
6.6 Overlapping fuzzified timing sequences. 130
6.7 Histogram of distances ($k = 45$ secs). 134
6.8 Separation as k varies. 136

7.1 Some query plans for Example 7.3.1. 144
7.2 Some query plans for Example 7.3.2. 146
7.3 Naively wrapping any query optimizer for (p, n)-*MIN-EXP* measure. 148
7.4 ConjunctiveWrapper for `Select` query for *MIN-EXP* measure. 156
7.5 LinearWrapper for simple `Select` query for *MIN-EXP* measure. 159
7.6 Example of a sawtooth filter. 161
7.7 Quality of solutions with number of approximate predicates ($p = 0.0$). . . . 164
7.8 Number of plans with number of approximate predicates ($p = 0.0$). 165

8.1 Basic index structures. 173
8.2 Number of Usenet postings per day in September 1997. 180
8.3 Average space required during CDS' operation and transition (W = 7). . . 196
8.4 Average transition time for CDS (W = 7). 196
8.5 Average work done by CDS during day (W = 7). 198
8.6 Average work done by WSE during day (W = 35). 198
8.7 Work done in TPC-D (packed shadowing) during day (W = 100). 200
8.8 Work done in TPC-D (simple shadowing) during day (W = 100). 201
8.9 Work done during day by CDS with W (n = 4). 202
8.10 Work done during day by CDS with SF (W = 14, n = 4). 202
8.11 Index size ratio with n (W = 7). 203

10.1 Algorithm for *DEL* 210

10.2 Algorithm for *REINDEX* 211
10.3 Algorithm for REINDEX^{+} 212
10.4 Algorithm for *REINDEX*$^{++}$ 213
10.5 Algorithm for *WATA** 214
10.6 Algorithm for *RATA** 215

Chapter 1

Introduction

The World Wide Web has created new business opportunities for content publishers such as Hollywood studios, music labels, software manufacturers and book publishers. These publishers can now exploit the web as a digital distribution channel rather than sell content through traditional mediums and stores. For instance, a music label may sell digital music on the web rather than print a physical CD and sell it through a record store. It is predicted that several billions of dollars worth of digital goods will soon be bought and sold on the web.

The web also creates new problems for digital content publishers. Once a customer has bought some goods and paid for them, the merchant has to deliver the content. In the case of digital content, the merchant may use some information exchange protocol such as HTTP or e-mail to deliver the goods to the customer. However once the provider delivers the digital content, the customer may offer this content on his web site, post it to a Usenet newsgroup or mail it to friends for a reduced price or perhaps even for free. The publisher will lose revenues due to reduced sales, if other potential customers start accessing the content from this *cyber-pirate*. Cyber-piracy is a major problem even if the publisher does not make the content available digitally, but if the content can be digitized. For instance, a cyber-pirate can purchase an audio CD of his favorite band from a record store. He can then use "CD ripping" software to digitize the content into a standard audio format (e.g., MP3) and offer it to other customers through a web site, netnews or email. Similarly, the cyber-pirate can scan in digital images and books and offer them on the web. In general, any digitizable content can be pirated; content publishers, authors, artists and performers are therefore projected to lose significant revenues. For instance, the music industry alone

is projected to lose two billion dollars every year in revenues by the year 2002 [RIAA99]. This loss corresponds to nearly 15% of their total revenues [For98].

Content publishers currently face a dilemma. On the one hand, they would like to use the web to (1) tap into the "impulse buyer" market that relishes immediate access to content, (2) make higher profits due to lower distribution and delivery costs, and (3) increase sales by attracting a larger customer base. On the other hand, they will lose revenues due to cyber-piracy and the proliferation of web sites with illegal copies. Sony Music, a record label, is one such publisher facing this dilemma. In 1997, Sony Music announced plans for an online jukebox where web surfers could listen to, buy and download digital music. When Sony Music realized that cyber-piracy on the web will soon cut into their sale of audio tapes and CDs, they retracted their announcement [Bul98]. Some book publishers when faced with this dilemma are adopting a less drastic solution. They offer a few pages of popular novels as a "teaser" on the web, while selling and shipping physical copies of the book to the reader. However, the true benefits of a digital delivery system (i.e., lower manufacturing and delivery costs) are lost because the publisher still has to print, bind and deliver the physical book.

We believe that content publishers will offer valuable digital content on the web only when effective techniques to combat cyber-piracy are in place. In this dissertation, we focus on techniques to protect the intellectual property of digital content authors and publishers.

1.1 Recent approaches to combat cyber-piracy

Protecting intellectual property has received a lot of attention recently, both in terms of revised intellectual property laws [Gol96], as well as new technology-based solutions [Ros96, Goo97]. We now outline a few technology-based solutions in research prototypes, or that are already available in commercial products.

1.1.1 Copy prevention

With copy prevention, content providers make it difficult to copy the original content. One common approach is to place information on stand-alone CD-ROM systems and allow users to access the data only through a restricted interface. A related approach is to use special-purpose hardware to build "fully trusted systems" using "fully trusted components" and secure interconnections [PK79, SL97]. For example, Bloomberg Financial Services deliver

real-time financial content to market brokers who lease special-purpose hardware units from Bloomberg Financials. Other examples include the recently announced MagicGate and OpenMG protection technologies [Osa99] from Sony Corporation. MagicGate will employ a microchip embedded in both the player/recorders and media so that protected content is transmitted only between compliant devices and media. OpenMG will employ a hardware module and special software to encrypt digital music content stored on a hard drive or similar storage device.

We believe that protecting content using special-purpose hardware works well in practice for relatively small-scale deployments (as in Bloomberg Financial Services). However this approach is unlikely to work for a global digital distribution channel because of the following "chicken and egg" problem. End-users will buy special-purpose hardware units only if (1) there is significant, valuable content available through these units, and (2) this content is not available elsewhere (e.g., as physical CDs). Publishers on the other hand are unlikely to switch away from existing distribution media (e.g., physical CDs), until they are convinced that customers will quickly adopt the new means (recall failure of Sony's Betamax video players and Digital Video Express' DivX players).

InterTrust Corporation and IBM offer software-based "trusted systems" to protect digital content [KLK97, SBW98]. For instance, Intertrust's *DigiBox* product allows a content publisher to encrypt content, define rules for content usage, and publish this encrypted content on a web site. When a customer downloads the content and wants to access it, he first obtains a decryption key from a DigiBox clearing center by paying for the content. After decrypting the content with the key, the user views the content using a "trusted viewer" that disallows copying. The approach adopted by DigiBoxes however has several drawbacks. Firstly, their approach requires a user's favorite viewer (such as emacs, vi, Microsoft Word) to be "DigiBox-compatible." Secondly, hackers with software emulators or screen capture software can copy the data when they view or play it. Also, the approach is designed for a publisher who is exclusively selling content online. The approach does not address the case when a a cyber-pirate digitizes content from an alternative source, e.g., by using CD ripping software for CDs.

1.1.2 Copy detection

With copy detection, the content provider (author or publisher) imposes few restrictions on content distribution unlike the copy protection schemes we discussed earlier. However

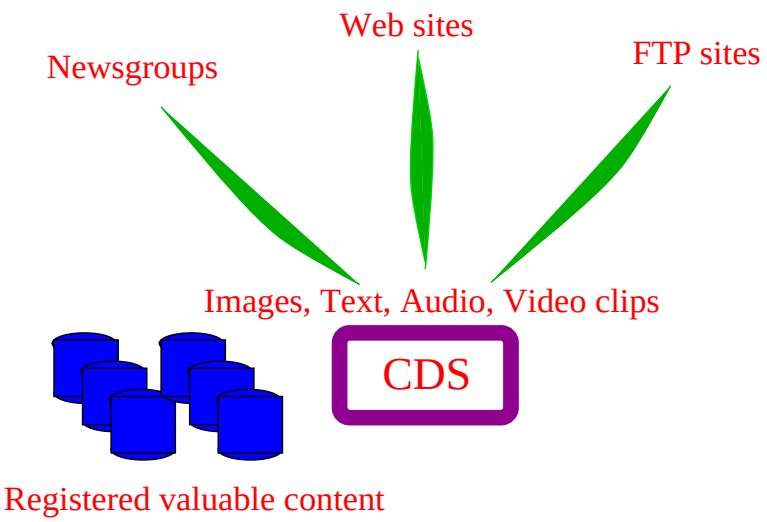


Figure 1.1: A Copy Detection System (CDS).

the provider *registers* his valuable digital content into a Copy Detection System (CDS) (Figure 1.1). The CDS then accesses public web sites, newsgroups, and FTP sites, and notifies the publisher if it find copies of registered digital content using one of the following techniques:

1. **Watermarking:** In the watermarking approach, the content provider uses steganographic techniques to hide information into digital content before selling the content. For instance when the content provider sells an image to a customer, the image is imperceptibly marked with the user's credit card number as well as the content publisher's name. When the CDS finds digital content at a public site, it automatically extracts the watermarks from the content. The CDS then notifies the corresponding publisher where the copy is located as well as the extracted credit card number (i.e., who was the initial buyer of the content.) Currently, there is a significant body of research into watermarking for multimedia content [Har99]. Commercial products are also available from Dice [Dic99] and DigiMarc [Dig99] to watermark audio and images respectively.

   The watermarking scheme relies on adding imperceptible, "hard-to-remove" watermarks to digital content, so that the cyber-pirate cannot identify and remove the watermark. However, hackers already know how to defeat existing watermarking techniques [CL99]. For example, if two customers *collude* to compare the different

versions of the content they bought, they can automatically remove differences (the watermarks) between the two versions [BS98]. Also the watermarking approach does not address the problem of a person digitizing content available through other means (e.g., ripping a CD), and making it publicly available.

2. **Content-based:** In this approach, the CDS compares the content on public sources (e.g., web sites) against the registered content and identifies potential copies of the registered content. For example, if Walt Disney Corporation wants to find all images of Mickey Mouse available on the web, they would register some sample images of Mickey Mouse with the CDS. The CDS will then try to identify images on public sources that are "similar" to Mickey Mouse and notify Disney of these copies.

   Building a content-based CDS is easy if the CDS only has to find exact copies of registered content. For example, we can use a simple checksum-based comparison of the content to identify potentially exact copies and then manually compare the content. However, a cyber-pirate can defeat such a CDS by modifying his copy of the content so it has a different checksum than the original registered content. For instance, the cyber-pirate may change the sampling rate of an audio clip before offering it on a web site. In general, the cyber-pirate will modify digital content so that the "quality" of the modified content is retained and there is little loss in the content's commercial value. Therefore, the goal of the CDS is to compare digital objects efficiently and identify good-quality renderings of registered content.

One problem with the copy detection approach is that the CDS can only access digital content in public web sites and newsgroups. The CDS cannot find copied content that is delivered through private email or content that is available behind fire-walls and password-protected sites. However, we believe this weakness is not significant for the following reason. Content publishers expect to lose significant revenues primarily from piracy on publicly accessible sites and not from "below the radar" password-protected sites and firewalls [RIAA99]. If a password-protected site tries to attract more traffic by publishing its password, the CDS can also access content at the site. So we expect CDS systems to play a crucial role in protecting intellectual property as we discuss in the next section.

1.2 Protecting copyright in the future

We believe that copy protection and detection techniques should be used in conjunction with each other to protect intellectual property. That is, the publisher should watermark digital content and use Cryptolope-like solutions [KLK97] while delivering content to customers. In addition, the publisher should register his content with a CDS that notifies him of publicly available, potential copies of the content.

We propose the above hybrid approach for the following reason. Copy protection schemes as well as watermarking schemes act as "cyber-locks" on content. However, these schemes are ineffective in combatting piracy once the "cyber-lock" is broken. For example, once a technically savvy cyber-pirate extracts an MP3 song from a Cryptolope, he can offer it on a web site[1]. The content-based CDS complements the above *passive* techniques by acting as a "cyber-cop." It automatically identifies publicly available, potential copies of digital content and notifies the appropriate content publisher. The publisher can then close down the web site with legal assistance [Gol96], if the copy is indeed illegal.

Cyber-piracy can never be completely stopped in the real world. For example, the software industry has been battling piracy for many years now without much success. However, our goal is to "raise the bar" for cyber-pirates so it is harder to make content illegally available to a wide-spread customer base. In this dissertation, we focus on building an active policing system for the web, by designing, building and evaluating the performance of a content-based copy detection system to combat cyber-piracy.

1.3 Architecture of a Copy Detection System

Content publishers *register* their digital content into the CDS *registry*. The CDS then *polices* publicly accessible, popular web sites, identifies potential copies and notifies the corresponding content publishers. The two principal components of such a CDS architecture are a (1) *crawler*, and (2) a *comparator* as we discuss below.

1. **Crawler:** The CDS employs a variety of *crawlers* such as the following to retrieve digital objects from the web.

[1] Henceforth we use the term "web" to refer to web sites, FTP sites, and newsgroups.

- **Traditional:** The CDS uses traditional crawlers like those servicing search engines (e.g., AltaVista and Excite) to retrieve text documents from the web. The CDS also extracts hyperlinks from these pages and retrieves multimedia content based on file extensions (e.g., `*.mov, *.mpg, *.mp3, *.ram`) for the Comparator (see below).

- **Probe:** The CDS "intelligently" queries search engines to retrieve potential copies of registered objects. For instance, the CDS can sample a few phrases from a registered text document and query a search engine to find web documents with these phrases. The CDS then downloads these documents to be evaluated by the Comparator. Similarly, the CDS can identify candidate audio copies of a "Beatles" song by querying a search engine for "Beatles and mp3." This search will find pages that mention both "Beatles" and "mp3." The CDS can then download these pages, extract links to ".mp3" files (if any exist), and download these files for the Comparator.

- **Streamed:** Many web sites push content across to clients using streaming technologies, such as PointCast, RealAudio and NetVideo. The crawler processes these streams and buffers them locally for the Comparator.

2. **Comparator:** Assume we have a boolean predicate $Similar(p, q)$ that checks if two digital objects p and q are similar. (We will soon discuss how to define such a predicate for specific media.) The comparator module supports the following operations:

   - FIND **operation:** Given a *query object* p, the FIND operation finds each registered object r such that $Similar(p, r)$ is true. For example, a Professor can check if a student report is plagiarized from some registered document, using this operation.

   - FIND-ALL **operation:** Given a set of query objects P, the FIND-ALL operation computes object pairs $\langle\ p, r\ \rangle$ such that p is an object in P, r is a registered object and $Similar(p, r)$ is true. For example, ClariNews may wish to identify unauthorized postings of their copyrighted articles, onto public newsgroups. Here R corresponds to the ClariNews articles and P corresponds to the hundreds of thousands of news postings on Usenet groups.

The CDS can execute FIND-ALL by performing a FIND operation for each object p in P, one at a time. However, we consider the FIND-ALL operation separately because (as we will see later) we can achieve better performance by executing several object comparisons simultaneously rather than performing them one at a time.

1.4 Challenges in building a Copy Detection System

1. **Scalability:** The CDS should identify web objects that are similar to registered objects efficiently, even when the number of web objects and registered objects is large. For example, the web is currently estimated to have about 800 million text web pages and this number is growing rapidly each month [BP99]. Therefore, a text CDS has to efficiently compare hundreds of millions of documents to registered documents using available disk, network bandwidth and CPU resources. Similarly, a CDS for other media such as audio and video should scale to the number of audio and video clips available on the web, especially as the number of Internet radio and TV stations increases.

2. **Resiliency:** As we discussed earlier, a cyber-pirate can modify a digital object in a variety of ways. For instance, a cyber-pirate can resample an audio signal or crop an image. The CDS should be resilient to such tamperings as much as possible and should be able to detect the similarity between the modified and the original object.

3. **Accuracy:** The CDS notifies a content publisher about potential copies of his registered objects. However the CDS may occasionally have *false positive* or *false negative* errors. If a web object is falsely reported to be similar to a registered object, the CDS has made a *false positive* error. Similarly, if a web object is similar to some registered object but is not reported to the publisher, the CDS has made a *false negative* error. Ideally, the CDS makes neither error.

4. **Different user requirements:** In general, a CDS should be flexible enough to handle varied requirements for different users. For instance, a photographer may want to find potential copies of his photographs on the web, even if the CDS takes a few weeks. However Disney Corporation, which has a large portfolio of images to protect, may prefer to use the CDS in a less accurate but faster mode. The underlying challenge here is to build a flexible CDS that can service different user requirements.

1.5 Contributions and structure of dissertation

In this dissertation, we address the above challenges by designing, implementing and evaluating content-based CDS systems. Conceptually, we adopt the following approach to build the Comparator for specific media.

- With the help of a domain expert, we design procedures to automatically extract media-specific *features* from objects. We then design *comparison functions* that identify an object and a high-quality rendering of the object, so that the functions mimic an objective human. For example, we can extract sentences from a text document to be the document features. We can then define two documents to be similar if they share more than (say) ten sentences in common.

- We automatically extract the above media-specific features from our registered content. We then build *index structures* to service the FIND and FIND-ALL queries efficiently. In the text document example, a useful index structure may be a hash table on all the extracted sentences.

- We extract the corresponding features from the crawled web content. We then probe the indices and identify all registered objects that are "similar" to the crawled object. In the text document example, we can use the hash table on registered sentences to speed up counting shared sentences.

The main challenge in building a CDS is to (1) choose the "best" media-specific features to extract from content so we identify different renderings of the content, and (2) build appropriate index structures to execute comparator operations efficiently.

As part of this dissertation, we built two experimental CDS systems: (1) SCAM (Stanford Copy Analysis Mechanism) for text documents, and (2) DECEIVE (Detecting Copies of Internet Video). We present the algorithms, implementation and experimental results behind these systems in the following chapters. We discuss our techniques first in the context of a text CDS (Chapters 2 through 5). Then we consider how our techniques are applicable to other media, such as video (Chapter 6). In Chapters 7 and 8, we discuss some media-independent issues in building CDS systems. Specifically, the chapters are organized as follows.

(Chapter 2) Textual features

We first discuss the attacks possible by a cyber-pirate to modify text content. Then we discuss different ways to define $Similar(a, b)$ for any pair of documents a and b. Specifically, we propose a variety of signatures (features) to extract from a text document, and document comparison functions. We then experimentally evaluate these similarity functions.

(Chapter 3) Executing text CDS operations efficiently

We discuss how to preprocess the registered text content and build index structures, so we can execute the FIND operation efficiently. Then we consider how to efficiently execute the FIND-ALL operation when the number of registered and web documents is of the order of millions of documents. Specifically, we introduce the *iceberg* class of techniques that efficiently execute the FIND-ALL operation using compact in-memory structures. We also note that the iceberg class of techniques can be employed in a wide class of applications including data mining, information retrieval and online analytical processing. Using a case-study approach, we show experimentally that our iceberg techniques offer two orders of magnitude speed up over previously employed techniques for some of the above applications.

(Chapter 4) Running our text CDS on the web

In this chapter, we show that our techniques and algorithms indeed scale to a very large data set, the web. We also note that in addition to our original motivation for protecting intellectual property, our techniques for building a text CDS are useful for a variety of other applications. We discuss two such applications in this chapter. In the first application, we show how a web crawler can use our techniques to avoid crawling redundant content. In addition, we also discuss a clustering tool we built to remove near copies of web pages before presenting web search results to a user. We report experimental results showing that our techniques and algorithms scale for over four hundred gigabytes of text data, about 60 million web pages.

(Chapter 5) Discovery-based plagiarism detection

In 1995, we successfully used an initial prototype of SCAM (our text CDS) in the following real-life plagiarism case. A researcher had plagiarized several technical papers written by other researchers and published them in conferences and journals. We discuss the history of this case and the role SCAM played in helping the Association of Computing Machinery

(ACM) in compiling a case against this researcher.

Based on the above case, we were motivated to study the following related problem. Many text documents are available in autonomous databases (e.g., Knight-Ridder's Dialog) and are accessible only through a search interface. If a publisher wants to check if a text document occurs in one of these databases, he has to query each database to identify similar documents. For instance, given an article discussing the Mars Pathfinder project, he may extract a few keywords from the document, and issue a Boolean query such as "Mars OR Pathfinder OR robot" to each database. This process is expensive if the number of databases is large (e.g., Knight-Ridder has several hundred databases) or if the disjunct query extracts many documents from each database. In this chapter, we propose schemes to automatically prune away databases that are unlikely to have documents similar to the query document. Also we show how to automatically compute the "best" Boolean query to search each unpruned database. We empirically evaluate our pruning and searching schemes, using a collection of 50 databases and report the results.

(Chapter 6) Building a video CDS

We first discuss the attacks possible on video content. Then we propose and experimentally evaluate a variety of features to extract, as well as functions to compare videos. We then propose novel index structures to speed up execution of the CDS operations.

(Chapter 7) Approximate query processing

For given media, a domain expert can design many different tests (predicates) to compute similarity of digital objects. Each such predicate will cost some time units in computational resources, and has a certain accuracy in estimated number of false positives and false negatives. At the same time, as we discussed earlier, each CDS user will have different requirements, in terms of tolerable accuracy and maximum cost. Then for each CDS user, we can choose the "best" available predicate that matches the user's requirement. However, since each user's requirement is likely to be very different from other users, it is unlikely we will always find a "good" matching predicate.

To address the above problem, we adopt the following approach. As before, the domain expert designs a variety of predicates, each with different cost and accuracy characteristics. However the CDS automatically *composes* some of the given predicates in different ways to create new predicates when a user specifies his requirement. For example, if an artist

requires the CDS to execute a FIND operation within two hours, the CDS can automatically create a new predicate as follows. The CDS *filters* the input to an expensive predicate with a cheaper but less accurate predicate so that the overall computation takes two hours to execute. Similarly, the CDS can create new predicates with smaller overall errors. In this chapter, we study the problem of automatically selecting and structuring predicates in order to reduce the cost of computing similar objects, while keeping errors within user-specified bounds. We then propose and evaluate some optimization strategies.

(Chapter 8) Indexing temporal data
In a CDS system, new data is being added every day by publishers. For instance, a publisher such as ClariNews may add a few thousand articles each day to the CDS registry. However, the publisher may also choose to *expire* articles more than a week old if he is not interested in protecting copyright of older articles. Index structures that we build to support our comparator operations should then be constantly reorganized when data is continually being added and expired. In this chapter, we consider the case when publishers require the CDS to maintain their registered objects for some window of days, e.g., a week or month. We propose a variety of *wave indices* where the data from new days can be efficiently added, and old data can be quickly expired, to maintain the required window.

1.6 Brief survey of related work

We now briefly summarize some related work to compare the overall contributions of this dissertation with other recent work. In each chapter, we also discuss closely related work when we contrast our techniques with such work.

In the past few years, a few research prototypes (e.g., SIF [Man94], COPS [BDGM95], Koala [Hei96], DEC's prototype [BGM97, BB99]) have been developed for applications similar to our text CDS (e.g., file compression and document versioning). As we will discuss in Chapter 2, our text CDS differs in the similarity measures it uses due to the accuracy, and resiliency requirements of our application. Also our text CDS uses specific data structures, and algorithms we have developed (Chapter 3) so that our text CDS scales efficiently to large data sets.

Our *iceberg* techniques in Chapter 3 are related to algorithms that derive *association*

rules in data mining [PCY95], and query processing algorithms in online analytical processing (OLAP) and data warehousing. Our primary contributions are (1) to observe that queries in such varied applications have similar structure (i.e., the "iceberg" structure), and (2) to propose new techniques that compute such queries more efficiently than existing techniques.

Our techniques that statistically summarize text databases in Chapter 5 are related to work in the Information Retrieval community in building metasearchers [GGMT94, GGM95]. However, as we discuss in Chapter 5, the summaries we extract are fundamentally different because our target application is different.

We discuss feature extraction from video sequences in Chapter 6. This work is related to feature extraction in the content-based video retrieval community [FSN95, Gev95, OS95, Sri95, ACM96, DM97]. However, as we discuss in Chapter 6, we propose features that are more efficient to extract, and similarity functions that work well for our target application (i.e., satisfy the accuracy and resiliency requirements). In addition, we propose novel index structures to satisfy our scalability requirements.

Our algorithms (in Chapter 7) to automatically compose approximate predicates are related to query optimization strategies in extensible database systems with expensive, user-defined predicates [CGK89, CS93b, HS93, CS96, CS97]. The LDL project at MCC [CGK89] and the Papyrus project at HP Laboratories [CS93b] proposed viewing expensive predicates as relations and using the System R dynamic programming algorithm for join and expensive predicate enumeration. Hellerstein and Stonebraker [HS93] proposed *predicate migration* as a way of interleaving join and selection predicates. Chaudhuri and Shim [CS96, CS97] recently proposed an algorithm to optimize the placement of expensive predicates in a query plan, in a System-R style query optimizer [SAC+79]. These optimizers do not consider approximate predicates and errors. Our focus is on approximate predicates, and how to construct the "right" composite predicate to *filter* input to the more expensive user-defined predicates, for a given user level requirement.

In Chapter 8, we consider how to index efficiently temporal data. Brown et al. [BCC94], Cutting and Pedersen [CP90], Faloutsos and Jagadish [FJ92] and Tomasic et al. [TGMS94] all consider how to incrementally index a growing corpus for fast information retrieval. They do not consider the case where a sliding window of documents is indexed. However their work is orthogonal to us. There is also a significant body of work in extending well-known index structures such as B+Trees and R-Trees for indexing time-series data [ST99]. Our

work differs from the above temporal indexing schemes in that we index sliding windows of data rather than data with arbitrary insertion and expiry times. This assumption helps us carefully organize our indices and make several performance optimizations. Also our schemes are independent of the underlying index structures used, and hence can be used on top of widely available index structures (such as B+Trees, ISAMs, etc.).

1.7 Summary

In summary, in this dissertation we discuss many aspects of building a content-based CDS system. Our contributions span issues such as what media-specific features to extract, how to compare objects, and what indices to build, to how we can support different user level requirements. In some cases, we also discuss how some of our techniques improve the "state-of-the-art" in other problem domains. In all cases, we evaluate our techniques and show how to build a scalable, accurate and resilient CDS system for specific media.

Chapter 2

Features for a Text CDS

2.1 Introduction

In this chapter we first discuss domain-specific attacks from a cyber-pirate that a text CDS should be resilient to. Keeping these attacks in mind, we discuss how to extract features and how to compute similarity between two documents. We discuss experimental results that highlight some of the drawbacks and benefits of these measures, for a variety of data sets.

2.2 Attacks from a cyber-pirate

A text CDS can easily find documents that are exact copies of each other. For instance, the CDS can compute the MD-5 checksum [KR95] of the given documents. Since documents with the same checksum are exact copies with a high probability, a human can then manually examine such documents. However, the text CDS has to identify a document and a modified version of the document to be similar. For instance, the cyber-pirate may copy only a section, chapter or paragraph out of a document. The cyber-pirate may also modify the document by inserting (or deleting) sentences and words not in the original document. In addition, the cyber-pirate may convert a document from its native format into another format. For instance, he may convert a PostScript document into an Adobe PDF document. The reader should keep these attacks in mind when we discuss our similarity measures in the next section. We will later consider how resilient our measures are to such attacks.

2.3 Possible features and similarity functions

In the past few years, a few research prototypes (e.g., SIF [Man94], COPS [BDGM95], Koala [Hei96], DEC's prototype [BGM97]) have been developed for applications similar to our own (e.g., file compression and document versioning). We roughly classify the underlying approaches of these systems into (1) *chunking* based and (2) *information retrieval (IR)* based techniques as discussed below.

2.3.1 Chunking based

Each text document is broken into a set of *chunks*, i.e., smaller text units. For example, assume we have a document ABCDEFGH where each letter corresponds to some text unit such as a sentence. Then the text document can be broken into one of several chunks, such as AB, CD, EF, GH, or ABC, CD, EFGH, or ABCD, BCDE, DEFGH. These chunks are usually then *compressed* (e.g., by hashing) and *sampled* for performance reasons (e.g., lower storage requirements). Two documents are then compared by the number of compressed chunks they share. For instance, if two documents share more than some minimum threshold of chunks, they are defined to be similar.

Systems such as SIF and DEC's prototype choose different chunking strategies and thresholds since they were developed for different applications. For instance, SIF was developed at the University of Arizona to identify different versions of files in a personal file system. On the other hand, DEC's recent prototype syntactically clusters millions of web pages for the AltaVista search engine [BGM97, Bro97].

The COPS [BDGM95] and Koala systems [Hei96] are related to our work since they target identifying copyright violations as well. However these systems were designed to support the FIND operation for small document sets. Our system has been designed to handle both the FIND and the FIND-ALL operations and to deal with much larger document sets, as we discuss in Chapter 4. We will discuss specific differences in chunking strategies later.

2.3.2 Information Retrieval based

First, we define some terminology. For our discussion, we denote the list of all distinct *words* (alpha-numeric strings delimited by punctuations and white spaces) in a document corpus to be V, the *vocabulary* of the corpus. Let N be the number of words in the vocabulary. Let w_i refer to the i^{th} word in V. Let D refer to a generic document (string of words).

We define the *occurrence vector* $O(D)$ to be the list of words appearing in D. Let $F(D)$ be the *frequency vector*, where $F_i(D)$ is the number of occurrences of word w_i in D. Let $sim(D_1, D_2)$ denote the similarity measure between documents D_1 and D_2, as computed below.

Information Retrieval (IR) engines are very popular since they allow keyword-based search interfaces to large corpora, e.g., the web. When a user issues a query set of keywords to an IR engine, the engine *ranks* documents in its corpus using some ranking function [SB88]. One popular ranking function is the *cosine similarity* measure (CSM) [Sal88] which defines relevance of document R to query Q (of keywords) to be

$$relevance(R,Q) = \frac{\sum_{i=1}^{N} F_i(R) * F_i(Q)}{\sqrt{\sum_{i=1}^{N} F_i^2(R) * \sum_{i=1}^{N} F_i^2(Q)}}$$

We can define document similarity for a text CDS as follows based on the above IR approach. Conceptually view each *query* document as a set of keywords. We can then rank registered documents using the CSM measure. To evaluate how this approach works in practice, consider the following example.

EXAMPLE 2.3.1 Assume that the corpus vocabulary is $V = \{a, b, c, d, e, f, g, h\}$. Consider query document S with $O(S) = \langle a, b, c\rangle$ and registered documents R_1, R_2 and R_3 with $O(R_1) = \langle a, b\rangle$, $O(R_2) = \langle a, b, c\rangle$ and $O(R_3) = \langle a, b, c, d, e, f, g, h\rangle$. Also consider additional registered documents $\{T_k\}$ $(k \geq 1)$ such that $O(T_k) = \langle a^k\rangle$, where k denotes the number of occurrences of word a in document T_k. For example, document $T_2 = \langle a, a\rangle$ and $T_4 = \langle a, a, a, a\rangle$. We would expect a good similarity measure to report S and R_1 to be "quite" similar, S and R_2 to be exact replicas, and S and R_3 to have significant overlap. Also document T_k should be similar to S for small k and dissimilar for high k.

We then have $sim(S, R_1) = (1*1+1*1)/\sqrt{3*2} = 0.82$ and $sim(S, R_2) = 1$. In this case, the measure appears to work well. But $sim(S, R_3) = 0.61$ is a fairly low value considering that R_3 contains S. Also $sim(S, T_k) = k/(k*\sqrt{3}) = 0.58$, independent of k. □

From the above example, we see some pitfalls in using the CSM measure. We now propose the Relative Frequency Measure, a modified version of CSM that detects *subset* overlaps (e.g., S and R_3) and is more sensitive to word occurrence frequency (e.g., S and T_k).

Relative Frequency Measure (RFM)

First, we define the *closeness set* for two documents, which is the set of words in the two documents with "similar" occurrence frequencies.

Definition 2.3.1 (**Closeness set**) For documents R and S, we define the *closeness set* $c(R,S)$ to contain only those words w_i such that

$$\epsilon > (\frac{F_i(R)}{F_i(S)} + \frac{F_i(S)}{F_i(R)})$$

where $\epsilon = (2^+, \infty)$ is a user-tunable parameter. If either $F_i(R)$ or $F_i(S)$ is zero, we define the above closeness condition to be false. □

Notice that if a word occurs the same number of times in two documents, it occurs in the closeness set irrespective of the value of ϵ. Now consider the case $F_i(R) = 3$ and $F_i(S) = 2$. If $\epsilon > (3/2 + 2/3)$ then w_i will be considered to be "close enough" to be in the closeness set. Intuitively, the closeness set determines the set of words used to the same extent in two documents: ϵ is a tolerance factor while computing this set.

Definition 2.3.2 (**Document subset measure**) We estimate the portion of document R that is a subset of document S as

$$subset(R,S) = \frac{\sum_{w_i \in c(R,S)} F_i(R) * F_i(S)}{\sum_{i=1}^{N} F_i^2(R)}$$

□

The above expression computes the *asymmetric* subset measure for document pair R and S, while only considering the close words. Note that this measure is very similar to the CSM measure. The main motivation for this measure is that the values reported in CSM are low even though a document may be a subset of another document (e.g., S and R_3). The subset measure avoids this problem by normalizing the numerator of the CSM measure only with respect to the first document.

Definition 2.3.3 (**Relative frequency measure**) We define the relative frequency measure for documents R and S to be

$$sim(R,S) = \max\{subset(R,S), subset(S,R)\}$$

□

If $sim(R,S) \geq 1$, we set $sim(R,S)$ to be 1. This is because no extra information is gathered when $sim(R,S)$ is computed to be greater than 1: the two documents are denoted to be very related anyway. We set the maximum value to be 1 merely to be able to express our similarity value as a range between 0 and 100%.

Intuitively, the components of the new similarity measure fit together as follows. The value of ϵ is the leeway in determining words to be used in computing the subset measure of a document pair. The similarity between the two documents is determined to be higher of the pair-wise subset measure. Hence RFM will help identify documents that are either a subset or a superset of another document. We continue with Example 2.3.1 below.

EXAMPLE 2.3.2 With $\epsilon = 2^+$, we get $c(S,R_1) = \{a,b\}$ and $sim(S,R_1) = \max\{2/3, 2/2\} = 1$. Also we get $sim(S,R_2) = 1$ and $sim(S,R_3) = 1$. Also $sim(S,T_k) = 1$ when $k = 1$, $sim(S,T_k) = 0$ when $k > 1$. Notice that the $sim(S,T_k)$ values decrease as k increases, as desired, something not done by the original CSM measure. Also $sim(S,R_3)$ is a high value indicating an overlap between S and R_3.

With an $\epsilon = 3$, $sim(S,R_1)$, $sim(S,R_2)$ and $sim(S,R_3)$ are unchanged. However, $sim(S,T_l)$ is now 1 for $k = 1$, 2/3 for $k = 2$ and 0 for $k \geq 3$. In general, the similarity value drops as k increases, but the difference in the "a" count needs to be larger before the similarity drops. □

In general, a high value of ϵ increases the tolerance level for flagging common words in partially overlapping documents, but increases the chances of matching unrelated documents (false positives). A low value of ϵ (2^+) will decrease false positives but will also decrease the ability to detect minor overlaps. One important question is how to choose a "good" value of ϵ that avoids missing partial overlaps while reporting few false positives. In the next section we address this question empirically by studying different collections.

2.4 Experiments

Evaluating the quality of similarity measures is tricky, since these measures approximate a human's decision process. Also, even two humans may not agree if two documents are similar. Ideally, if a benchmark database of copyright violations were available, we could evaluate our similarity measures against this data. Unfortunately, no such benchmark exists. Thus, for our experiments, we start with a set of documents that are known to have "substantial" overlap, and then see if our measures can correctly identify these cases. In

addition, in Chapter 5 we will discuss a real-life plagiarism case where we successfully tested our measures.

2.4.1 Human classification

For our experiments, we first classify documents in our corpus (we will discuss our data sets soon) based on the following predicates. We stress that this classification was very subjective, but was our best attempt to identify documents that most humans would agree are closely related.

1. **Some overlap:** If a document includes some clearly recognizable portions of another document, the document pair satisfies the predicate.

2. **High overlap:** This is similar to the previous category, except that a document includes large portions of another document, possibly interspersed with other text.

Recall that the objective of our similarity measures is to closely mimic the above human classification. Based on some exploratory experiments we discussed in [SGM95], we choose the following measures to evaluate further.

1. **RFM:** Traditional IR systems often throw away "stop words" before indexing since these words are too frequent. We also pre-process our documents in a similar fashion. Specifically, we use the stop word list (391 words) from WAIS [KM91], a popular IR system. We refer to this measure as the RFM-0 measure. In addition, we also evaluate RFM-n measures, where we also throw away the most frequent n words in our data set. For instance, if in addition to removing the WAIS stop words, we also throw away the 100 most frequent words in our data set, we refer to that as the RFM-100 measure.

2. **Chunk based schemes:** Here we evaluate the following chunking strategies. First, we evaluate a simple n-gram chunking, where the document is broken into a series of overlapping n-grams. For instance, for document ABCDEFG, a 3-gram chunking would be ABC, BCD, CDE, DEF, EFG. In addition, we evaluate the *hashed breakpoints* measure proposed by Brin et al. [BDGM95] and Manber [Man94] that operates as follows. Start by hashing the first word in the document. If the hash value modulo k is equal to zero (for some chosen k), the first chunk is merely the first word. If not, consider the second word. If its hash value modulo k is zero, the first two words constitute the chunk. If not, continue to consider the subsequent words until some

word has a hash modulo k equal to zero, and the sequence of words from the previous chunk break until this word constitutes the chunk. For such chunk based schemes, we can estimate the overlap between the two documents to be the number of such shared chunks.

2.4.2 Data sets

For our experiments we used the following four data sets.

1. **Netnews articles (N1):** We used 1233 netnews articles (with Usenet headers removed) from the `rec.sports.cricket` news hierarchy from January 1995. We chose netnews articles since these types of articles "stress-test" copy detection mechanisms for several reasons. Netnews articles are relatively small. Hence substantial overlap between articles may be a few sentences, and may be difficult to detect. Also, the articles are informal, so when people copy text into an article, they may copy incomplete sentences. Furthermore, many articles have long "signatures" with character pictures and unstructured text. We then chose 50 articles with many partial and exact duplicates. We identified 129 articles that overlapped with these articles according to our human classification.

2. **Netnews articles (N2):** This data set constitutes $50,000$ netnews articles from September 29th 1995. We carefully examined 1000 documents from the `clari.news` and `rec.sports.cricket` hierarchies in this data set. We chose 100 articles that had many partial and exact duplicates. We then identified and classified the 220 articles that overlapped with these 100 articles.

3. **Technical papers:** This data set constitutes ninety-two technical papers written by members of our research group. These papers averaged 7300 words and 450 sentences in length. Some of these papers were related since they were revisions of some original document, i.e., the technical report version, conference version or journal versions of some paper. In these cases, the authors of these papers classified their papers based on how much overlap the papers had.

4. **Web pages:** We populated this data set with four copies of a Perl manual (about 200 pages each) in HTML that we downloaded from the Internet. Each copy was a slightly different version of the original Perl manual, since they were mirrored at web sites at

different times. (i.e., some bug fixes and updates in the most recent version were not yet propagated to the earlier mirrored versions). We also added four different versions of an XML manual that had 25 HTML pages. Finally, we added 10,000 pages that we randomly crawled from the web.

We assume that these 10,000 pages are not related to each other and are not related to the pages in the Perl and XML manuals. We then classified the Perl and XML pages based on overlap as follows. We assume that matching pages in the different versions of a manual with the same modification time-stamp are identical. We tested this assumption on a few sample pages and found this assumption to be valid. We examined the pages with different time-stamps and classified them manually based on overlap.

2.4.3 Training similarity measures

We first "trained" our similarity measures with our N1 netnews data to choose good values for tunable parameters (e.g., ϵ) as follows.

1. For a given measure and a set of parameter values, we computed the similarity for each of the 50 query articles against all articles in N1. Then we computed the false positive and negative errors for the measure as motivated by the following example.

   EXAMPLE 2.4.1 Consider the case where the human had classified ten documents to have high overlap with a query document. Say the measure identifies 8 documents to have high overlap with the same query document, 6 (out of the 8) of which are documents that have high overlap according to the human.

   The false negative error for that query document is computed as $(10 - 6)/10 = 40\%$ since four documents with high overlap according to the human were not identified by the measure. The corresponding false positive error is 2/1233 since the measure falsely identified two out of the 1233 documents in N1. □

   The accuracy for a specific measure and parameter values is then computed to be the average false positive and negative error for the 50 query documents.

2. For each measure, we automatically varied the relevant parameter values and repeated the above procedure. We also recorded the corresponding average false positive and negative errors.

Chunking	Overlap	N2 Netnews		Papers		Web	
		False -	False +	False -	False +	False -	False +
RFM-0	Some	13.88%	0.13%	12.3%	1.2%	6.2%	1.3%
	High	9.26%	$\sim 0\%$	11.3%	0.4%	5.3%	0.8%
RFM-100	Some	8.81%	0.22%	8.1%	1.9%	7.9%	1.1%
	High	6.07%	$\sim 0\%$	7.2%	$\sim 0\%$	9.3%	$\sim 0\%$
RFM-300	Some	6.06%	0.11%	4.1%	0.2%	5.2%	0.1%
	High	6.0%	$\sim 0\%$	3.2%	$\sim 0\%$	5.6%	$\sim 0\%$
5-modulo	Some	11.76%	$\sim 0\%$	4.6%	0.1%	10.2%	$\sim 0\%$
	High	8.38%	$\sim 0\%$	3.2%	0.1%	8.1%	$\sim 0\%$
7-modulo	Some	13.88%	$\sim 0\%$	12.1%	$\sim 0\%$	13.3%	$\sim 0\%$
	High	13.04%	$\sim 0\%$	6.1%	$\sim 0\%$	8.1%	$\sim 0\%$
3-gram	Some	8.3%	0.2%	6.5%	$\sim 0.5\%$	9.1%	$\sim 0\%$
	High	6.9%	0.15%	5.9%	$\sim 0\%$	6.1%	$\sim 0\%$
5-gram	Some	6.1%	$\sim 0\%$	4.7%	$\sim 0\%$	5.9%	$\sim 0\%$
	High	6.5%	$\sim 0\%$	3.2%	$\sim 0\%$	5.1%	$\sim 0\%$
7-gram	Some	8.2%	$\sim 0\%$	9.2%	$\sim 0\%$	7.1%	$\sim 0\%$
	High	8.8%	$\sim 0\%$	9.9%	$\sim 0\%$	7.3%	$\sim 0\%$

Table 2.1: Accuracy of similarity measures.

3. With the values computed from the first two steps, we chose parameters values for each similarity measure as follows. We chose sets of parameter values for which the false positive error was less than 0.1% and broke ties by choosing the values with the lowest false negative error. The similarity measure with these chosen parameter values was used as the "trained" measure for subsequent experiments we discuss in the next section.

   We considered other similar approaches of choosing parameter values (e.g., choosing specific false negative errors and then breaking ties with lower false positive errors). However, we do not report on these approaches since we observed similar trends for the results that we discuss below. In Chapter 7, we will revisit this issue when we consider how to design predicates for specific user requirements.

2.4.4 Evaluating similarity measures

Given the trained similarity measures from the previous section, we evaluated how they performed for the N2 netnews, technical papers and web page data sets we discussed earlier.

We present the corresponding average false positive and negative errors in Table 2.1. From Table 2.1 we observe that the 5−gram and the RFM-300 measures perform well in general. That is, their false negative errors are typically lower than the other measures, and also they perform well across different data sets. We explored some of the classified results to understand why these measures performed better than others and observed the following trends.

- The main factor that impacts accuracy is the average length of chunk. As the length of the chunk increases, it is harder to detect partial overlap since overlapping sequences in two documents may start anywhere within the chunk. Hence longer chunks tend to have larger false negative errors. For example, the 7-gram measure has high false negative errors especially when trying to identify document pairs with Some overlap. On the other hand, when chunk lengths are small, the false positive errors increase due to loss in sequencing information. For instance, two documents that share the words "copyright" and "protection" may really not have any overlap. Therefore, we see that the 3-gram and the RFM measures have higher false positive errors.

- The accuracy of the RFM measures increases when we increase the number of stop words. This is because the less common words become a unique signature for the documents, when the common words are removed. However, we have observed that this trend does not continue beyond RFM-300 since no overlap is detected if we consider too many words as stop words in these documents.

- The measures in general perform poorly for netnews articles compared to technical papers and web pages primarily because netnews articles are very unstructured. For instance, netnews articles often have long "signatures" at the bottom of news messages. Also users sometimes include portions of netnews articles when responding. Also there were several articles that had limited punctuation and odd tabbing structures. For instance, many articles in `rec.sport.*` include lists of "favorite" players/ teams, and `clari.finance.*` groups list current gold prices and currency conversion rates as tables. The RFM schemes are more resilient to these cases since they are based on individual words, unlike the chunking schemes that rely on sequences of words. On the other hand, all measures perform relatively well for the technical papers and for the web data since they are more structured.

Based on the above observations, in the rest of the thesis we primarily focus on the 5-gram similarity measure due to its accuracy and inherent simplicity.

2.5 Conclusion

In this chapter we discussed a variety of attacks on a text CDS. We then designed similarity measures for the text CDS to compare documents. We trained these measures and evaluated how these measures perform for a variety of data sets. Based on our experiments, we recommended some similarity measures that are simple to evaluate and are also accurate.

Chapter 3

Executing Text CDS Operations

3.1 Introduction

In this chapter, we first discuss how to execute the FIND operation, given a query document and a set of registered documents. We then focus on how to efficiently execute the FIND-ALL operation given a set of registered documents and a set of query documents. As we discussed earlier, we can implement the FIND-ALL operation by executing the FIND operation for each query document. However in this chapter, we show how to efficiently execute the FIND-ALL operation by simultaneously comparing all documents in both sets. In summary, the primary contributions of this chapter are three-fold:

1. We first discuss how we execute the FIND operation efficiently using some basic data structures such as inverted indices.

2. We identify *iceberg* queries as a fundamental class of queries, of which FIND-ALL is a special case. We then present a wide-ranging set of applications where icebergs appear either directly, or as sub-queries in more complex queries. We discuss techniques that these applications currently employ and show why they do not scale well to large data sets.

3. We propose a variety of novel algorithms for iceberg query processing. Our algorithms use as building blocks well-known techniques such as *sampling* and *multiple hash functions*, but combine them and extend them for fast execution of iceberg queries. Finally, we show experimentally that our techniques scale for several gigabytes of data.

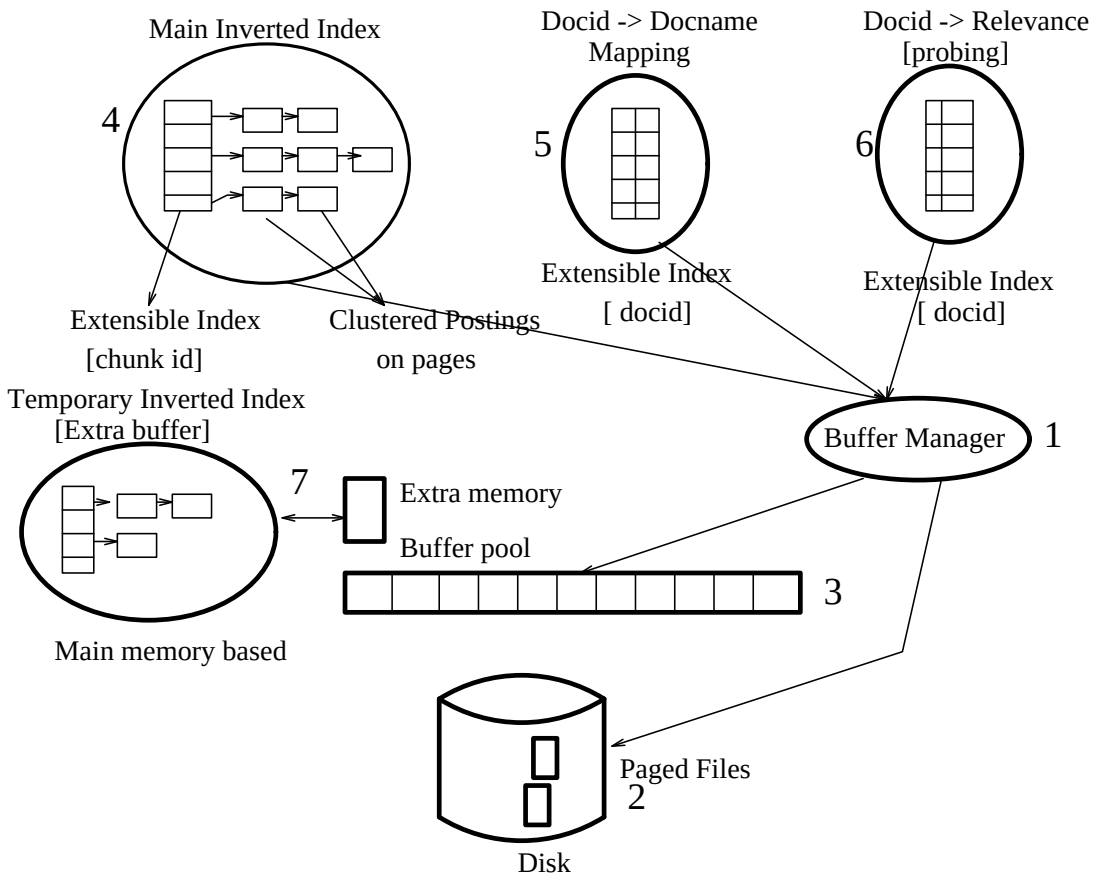


Figure 3.1: Data structures for FIND operation.

The rest of the chapter is structured as follows. In Section 3.2 we discuss some data structures we use to efficiently execute the FIND operation. In Section 3.3.1 we discuss how to implement the FIND-ALL operation, and notice that it is a special kind of an iceberg query. In Section 3.4 we present two simple algorithms to execute iceberg queries. In Section 3.5 we propose hybrid algorithms that combine the advantages of the two simple algorithms. In Section 3.6 we propose some additional optimizations for our hybrid strategies. In Section 3.8 we evaluate our techniques experimentally. In Section 3.7 we propose some extensions to our algorithms.

3.2 Implementing the FIND operation

For a given query document, we now discuss how to implement the FIND operation.

Recall that the FIND operation identifies all registered documents similar to the query document. That is, it identifies all registered documents that share more than T (threshold) 5-gram chunks with the query document. Figure 3.1 illustrates the main data structures we use for this operation. Keep in mind that these structures should scale to large numbers of documents. For the reader's convenience we have numbered the components shown in the figure, and use these numbers as superscripts in the text that follows. We use a Buffer Manager (BM)[1] that interacts with the disk[2] through paged files (blocks of data). The BM caches the following persistent data structures (that reside on disk) into the buffer pool[3](main memory) for fast access:

- **Inverted index on chunks:**[4] In this structure (commonly used in IR systems [Sal88, Sal92]), we maintain *postings* indicating the (nonzero) occurrence frequency of chunks in registered documents. For instance, a posting may be of the form *("picture", 11, 5)* to indicate that chunk "picture" occurs 5 times in a document with unique identifier 11. We use an *extensible hashing* structure indexed on chunks for efficient lookup [Ull88]. Each chunk points to a set of pages that stores its postings. All the postings for a chunk are clustered together (in the same disk pages) for efficient access, with overflow pages added if necessary.

  One simple way of building such an inverted index is as follows. For each document with identifier (ID) d_i in registry, compute its chunks. For each chunk c_j in d_i, append entry $\langle d_i, c_j \rangle$ to a file named $DocChunk$. After processing all registered documents, sort $DocChunk$ by the chunks, so that the IDs of all documents that share chunk c_j are clustered together. After the sort operation, build an index (e.g., Indexed Sequential Access Mechanism [Ull88]) on the chunks. This index plus the all clustered postings together constitute the inverted index.

- **Name mapper:**[5] In this structure, we store the mapping from the unique document ID used in postings to the actual file name or URL of the web page. For instance, an entry may be (11, `http://www-db.stanford.edu`) indicating that the document with ID 11 is a web page at the given URL. This structure is used so that the postings in the inverted index are smaller, since we store integer IDs rather than a string of bytes for each document. We implement the name mapper as an extensible hashing structure that is indexed on document ID. It is easy to build this structure while building the inverted index as well.

- **Relevance array:**[6] Recall that we compute the similarity between the query document and a registered document by counting the number of common chunks. We use a relevance array for maintaining the count (as we discuss below). Typically such structures are implemented as arrays in main memory. However since the set of registered documents could be large, it may not be possible to allocate an array at run time. We implement the relevance array as an extensible hashing structure that is indexed on document ID to provide fast access to similarity scores. Since this structure is also implemented as paged files, portions can be paged to disk as the structure grows.

Given the above data structures, we execute the FIND operation for a query document as follows. For each chunk in the query document, we retrieve the postings in the persistent inverted index[4] to identify all registered documents containing that chunk. For each such posting, we increment by one the corresponding registered document's entry in the relevance array[6]. Finally, for each entry in the relevance array with count above threshold T, we probe the name mapper[5] for the name (filename or URL) of the corresponding document and report this to the user. The above process is very similar to how IR engines compute relevance of user queries to documents in a corpus [Sal88].

3.3 Implementing the FIND-ALL operation

Given a set of query documents, we now discuss how to find all registered documents similar to each of the query documents. As we mentioned earlier, the easiest approach is as follows. For each query document, execute the FIND operation across the registry as discussed in the last section. However, we now show how to achieve significant speedups by simultaneously comparing all query and registered documents. For simplicity, we assume that the set of registered documents and the set of query documents is the same. That is, we show how to efficiently identify all pairs of documents in a corpus that are similar to each other. We will later discuss the case when these sets are different.

In the last section, we discussed how to compute the *DocChunk* relation. Recall that this relation has tuples $\langle d_i, c_j \rangle$ for each chunk c_j in document d_i. Also recall that we sorted this relation on chunks (i.e, 5-grams), so that the IDs of all documents that share chunk c_j are clustered together. Given this relation, the FIND-ALL operation should produce $\langle d_i, d_j, f_k \rangle$ for each d_i, d_j document pair with f_k chunks in common, where $f_k \geq T$ for threshold T.

The simplest way to implement the above operation is to maintain an array of counters

in main memory for each document pair. We can then scan $DocChunk$ and for each pair of tuples $\langle d_i, c_k \rangle$ and $\langle d_j, c_k \rangle$, we can increment the counter for d_i, d_k. Finally, we scan all the counters and output only those document pairs with a counter value greater than T. While this approach is conceptually simple, we need to allocate n^2 counters for n registered documents. While this memory requirement is acceptable when n is small (e.g., few thousands), this approach requires several hundreds of gigabytes of main memory when n is on the order of millions of documents. One way to reduce the main-memory requirement of the above scheme is to allocate counters only for document pairs that share at least one chunk. However, we will see later that this optimization does not offer much reduction since many document pairs often share one or more chunks.

We can modify the above main-memory approach as follows by using some temporary storage on disk.

1. Scan the sorted $DocChunk$ relation. For each pair of tuples $\langle d_i, c_k \rangle$ and $\langle d_j, c_k \rangle$, produce $\langle d_i, d_j \rangle$ and store into a file called $DocDoc$, on disk.

2. Sort $DocDoc$ so that entries for each given document pair $\langle d_i, d_j \rangle$ are contiguous.

3. Scan the sorted $DocDoc$ relation. Count the number of occurrences of each document pair and only output document pairs that occur more than T times.

While the above algorithm does not have the high main-memory requirements of the earlier solutions, it unfortunately is not *output sensitive* for the following reason. In Step (1), when we produce all document pairs that share a chunk, often the number of tuples produced is very large. This is because many document pairs may share one or more chunks, even if few document pairs share more than T chunks. However, the above procedure explicitly counts the number of shared chunks for all document pairs before thresholding with T. For example, as we will see later in one of our experiments, an input $DocChunk$ produces $DocDoc$ that has 80 times as many tuples as $DocChunk$. However, after thresholding, the final answer set is about one hundredth as many tuples as $DocChunk$. That is, the intermediate $DocDoc$ has about 8000 times as many tuples as the final output. In the next few sections, we discuss algorithms that identify the final answer set without explicitly counting the number of shared chunks for all document pairs.

3.3.1 Generalizing FIND-ALL operation to *Icebergs*

Consider relation *R(target1, target2,..., targetk, rest)*. The following operation (in SQL) aggregates (i.e., counts) over a set of attributes and then eliminates aggregate values that are below threshold T.

```
SELECT    target1, target2, ..., targetk, count(rest)
FROM      R
GROUP BY  target1, target2, ..., targetk
HAVING    count(rest) >= T
```

When the number of unique *target* values is huge (the "iceberg"), but the answer, i.e., the number of frequently occurring targets, is very small (the "tip of the iceberg"), we refer to the above query as an *iceberg* query. Notice that the FIND-ALL operation is an iceberg query. The FIND-ALL operation finds all document pairs (targets) with more than T common chunks. The number of unique document pairs (targets) is huge, but the number of document pairs with more than T common chunks is typically small. For example, as we mentioned earlier, we will soon discuss a case where the number of targets is 80 times the cardinality of *DocChunk*, while the final output is one-hundredth the cardinality of *DocChunk*.

Many *data mining* queries are iceberg queries. For instance, market analysts execute *market basket* queries on large data warehouses that store customer sales transactions. These queries identify user buying patterns, by finding item pairs (and triples) that are bought together by many customers [AS94, BMS97, BMUT97]. (Target sets are item-pairs, and T is the minimum number of transactions required to *support* the item pair.) Since these queries operate on very large datasets, solving such iceberg queries efficiently is an important problem. In fact, Park et al. claim that the time to execute the above query dominates the cost of producing interesting *association rules* [PCY95].

Iceberg queries also arise in many information retrieval (IR) problems. For instance, IR systems often compute *stop words*, the set of frequently occurring words in a given corpus, to optimize query processing and build inverted indices – computing this set is another iceberg query. IR systems also sometimes compute sets of frequent co-occurring words, and use these to help users construct queries. For instance, the pairs "stock market," "stock price," and "chicken stock" may occur often in a collection of documents. If the user enters the word "stock" in a query, the system may suggest "market, "price," and "chicken" as useful words to add to the query to distinguish the way in which "stock" is

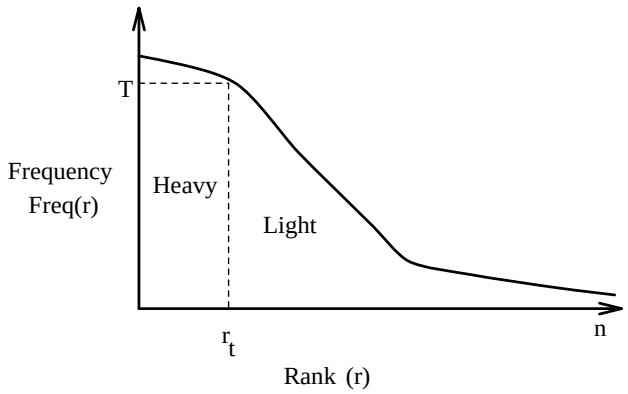


Figure 3.2: A graphical view of terminology.

used. Computing co-occurring words again involves an iceberg query, where target-sets are pairs of words [BMUT97]. We will study this application again in more detail in our experimental case-study.

From the above illustrative examples, we see that iceberg queries occur commonly in practice. We discuss more such examples in [FSGM+98]. In the next few sections, we propose techniques to solve such queries efficiently. Since iceberg queries are general, in the following sections we discuss these techniques in their general formulation. Subsequently, we discuss how these techniques are applicable for our FIND-ALL operation.

3.4 Techniques for thresholding

For simplicity, we present our algorithms in the next few sections in the context of a materialized relation R, with $\langle target, rest \rangle$ pairs. We assume for now we are executing a simple iceberg query which groups on the single target in R, as opposed to a set of targets. As we will discuss later, our algorithms can be easily extended for unmaterialized R as well as multiple target sets.

We start by establishing some terminology. Let V be an ordered list of targets in R, such that $V[r]$ is the r^{th} most frequent target in R (r^{th} highest rank). Let n be $|V|$. Let $Freq(r)$ be the frequency of $V[r]$ in R. Let $Area(r)$ be $\sum_{i=1}^{r}[Freq(i)]$, the total number of tuples in R with the r most frequent targets. A special case is $Area(n)$, which is equal to N $(= |R|)$. Figure 3.2 shows a typical frequency curve $Freq(r)$. The leftmost value on the horizontal axis is 1, representing the rank of most frequent target value.

Our prototypical iceberg query (Section 3.3.1) selects the target values with frequencies higher than a threshold T. That is, if we define r_t to be $\max\{r|Freq(r) \geq T\}$, then the

answer to our query is the set $H = \{V[1], V[2], \ldots, V[r_t]\}$. We call the values in H the *heavy* targets, and we define L to be the remaining *light* values.

The algorithms we describe next answer the prototypical iceberg query, although they can be easily adapted to other iceberg queries. In general, these algorithms compute a set F of *potentially* heavy targets, that contains as many members of H as possible. In the cases when $F - H$ is non-empty the algorithm reports *false positives* (light values are reported as heavy). If $H - F$ is non-empty the algorithm generates *false negatives* (heavy targets are missed). An algorithm can have none, one, or both form of errors. These errors can be eliminated through post-processing:

1. **Eliminating False Positives:** After F is computed, we can scan R and explicitly count the frequency of targets in F. Only targets that occur T or more times are output in the final answer. We call this procedure *Count(F)*. This post-processing is efficient if the targets in F can be held in main-memory along with say $2 - 4$ bytes per target for counting. If F is too large, the efficiency of counting deteriorates. In fact, as $|F| \rightarrow n$, the post-processing will take about the same time as running the original iceberg query.

2. **Eliminating False Negatives:** In general, post-processing to "regain" false negatives is very inefficient, and may in fact be as bad as the original problem. However, we can regain false negatives efficiently in some high skew cases where most R tuples have target values from a very small set.[1] In particular, suppose that we have obtained a partial set of heavy targets $H' = F \cap H$, such that most tuples in R have target values in H'. Then we can scan R, eliminating tuples with values in H'. The iceberg query can then be run on the remaining small set of tuples (either by sorting or counting) to obtain any heavy values that were missed in H'.

We now present two simple algorithms to compute F, that we use as basic blocks for our subsequent, more sophisticated algorithms. Each algorithm uses some simple data-structures such as lists, counters and bitmaps for efficient counting. For ease of presentation, we assume that the number of elements in each structure is much smaller than $|V|$, and that all structures fit in main memory. In Section 3.8 we evaluate the memory requirements more carefully.

[1] The $80 - 20$ rule is an instance of high skew. When the rule applies, a very small fraction of targets account for 80% of tuples in R, while the other targets together account for the other 20% [Zip49].

3.4.1 A Sampling-Based Algorithm (SCALED-SAMPLING)

Sampling procedures are widely adopted in practice for several applications [Olk93]. We now consider a simple sampling-based algorithm for iceberg queries. The basic idea is as follows: Take a random sample of size s from R. If the count of each distinct target in the sample, scaled by N/s, exceeds the specified threshold, the target is part of the candidate set, F. This sampling-based algorithm is simple to implement and efficient to run. However, this algorithm has both false positives and false negatives, and removing these errors efficiently is non-trivial, as we discussed above. We will show how to remove these errors using our HYBRID algorithms in the next section.

3.4.2 Coarse counting by bucketizing elements (COARSE-COUNT)

"Coarse counting" or "probabilistic counting" is a technique often used for applications such as query size estimation, for computing the number of distinct targets in a relation [FM85, WVZT90], and for mining association rules [PCY95]. The simplest form of coarse counting uses an array $A[1..m]$ of m counters and a hash function h_1, which maps target values from $\log_2 n$ bits to $\log_2 m$ bits, $m << n$. The *CoarseCount* algorithm works as follows: Initialize all m entries of A to zero. Then perform a linear scan of R. For each tuple in R with target v, increment the counter $A[h_1(v)]$ by one. After completing this *hashing scan* of R, compute a bitmap array $BITMAP_1[1..m]$ by scanning through array A, and setting $BITMAP_1[i]$ if bucket i is *heavy*, i.e. if $A[i] \geq T$. We compute $BITMAP_1$ since it is much smaller than A, and maintains all the information required in the next phase. After $BITMAP_1$ is computed, we reclaim the memory allocated to A. We then compute F by performing a *candidate-selection* scan of R, where we scan R, and for each target v whose $BITMAP_1[h_1(v)]$ is one, we add v to F. Finally we remove the false positives by executing *Count(F)*. Note that there are no false negatives in our coarse-counting approach.

The candidate-selection scan in this simple coarse-counting algorithm may compute a large F (that may be many times as large as the given memory), since light targets may be hashed into *heavy* buckets. A bucket may be heavy if it has (1) one or more heavy elements, or (2) many light elements whose combined counts are above the specified threshold.

3.5 HYBRID techniques

We now present three different approaches to combine the sampling and counting approaches we presented earlier. Each approach first samples the data to identify *candidates* for heavy targets; then it uses coarse-counting principles to remove false negatives and false positives. By this two-stage approach, we manage to reduce the number of targets that fall into heavy buckets – this leads to fewer light targets becoming false positives. We refer to the three approaches as the *HYBRID* class of algorithms.

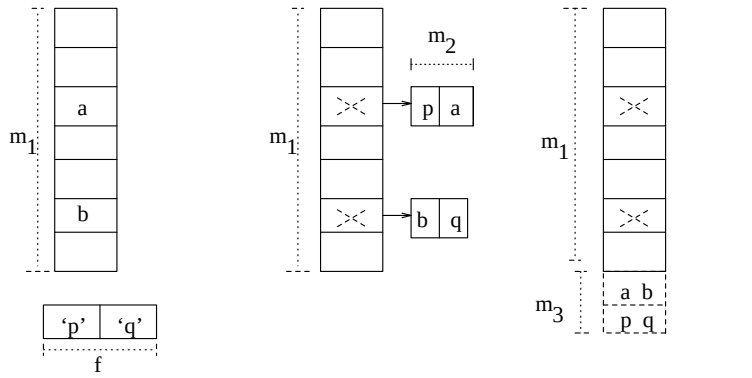
3.5.1 DEFER-COUNT Algorithm

First, compute a small sample (size $s << n$) of the data using sampling techniques discussed in Section 3.4.1. Then select the $f, f < s$, most frequent targets in the sample and add them to F. (These targets are likely to be heavy, although we do not know for sure yet.) Now execute the hashing scan of COARSE-COUNT, but do not increment the counters in A for the targets already in F. Next perform the candidate-selection scan as before, adding targets to F. Finally, remove false positives from F by executing $Count(F)$.

We see an example of this approach in Figure 3.3 (a). Consider the case when p and q are heavy targets, and targets a and b are light targets. In this case, p and q were identified in the sampling phase to be potentially heavy, and are maintained explicitly in memory (denote by 'p' and 'q') so they are not counted in the buckets (as are a and b).

The intuition behind the DEFER-COUNT algorithm is as follows. Sampling is very good for identifying some of the heaviest targets, even though it is not good for finding all the heavy targets. Thus, we select f so that we only place in F targets that have a very high probability of being heavy. Then, for each of these targets v that is identified in advance of the hashing scan, we avoid pushing $A[h_1(v)]$ over the threshold, at least on account of v. This optimization leads to fewer heavy buckets, and therefore fewer false positives.

The disadvantage of DEFER-COUNT is that it splits up valuable main memory between the sample set, and the buckets for counting. Even if f is small, we maintain the explicit target. For instance, if we use DEFER-COUNT to count heavy item pairs (two-field target set) in data mining, we need eight bytes to store the item pair. This gets progressively worse if we start counting heavy item triples, or heavy item quadruples, and so on. Another problem with implementing DEFER-COUNT is that it is hard to choose good values for s and f that are useful for a variety of data sets. Yet another problem with DEFER-COUNT



(a) DEFER-COUNT (b) MULTI-LEVEL (c) MULTI-STAGE

Figure 3.3: Alternate HYBRID techniques to combine sampling and coarse-counting.

is that for each target, we incur the overhead of checking if the target exists in f during the hashing scan.

3.5.2 MULTI-LEVEL Algorithm

We now propose an algorithm that does not explicitly maintain the list of potentially heavy targets in main memory like DEFER-COUNT. Instead MULTI-LEVEL uses the sampling phase to identify potentially heavy buckets as follows.

First, perform a *sampling* scan of the data: For each target v chosen during this sampling scan, increment $A[h(v)]$, for hash function h. After sampling s targets, consider each of the A buckets. If $A[i] > T * s/n$, we mark the i^{th} bucket to be *potentially heavy.* For each such bucket allocate m_2 *auxiliary* buckets in main memory. (We will sometimes refer to the A buckets as *primary* buckets, to maintain the distinction.)

Next, reset all counters in the A array to zero. Then perform a hashing scan of all the data. For each target v in the data, increment $A[h(v)]$ if the bucket corresponding to $h(v)$ is not marked as potentially heavy. If the bucket is so marked, apply a second hash function $h_2(v)$ and increment the corresponding auxiliary bucket.

We show an example of this procedure in Figure 3.3 (b). In the sampling phase, two buckets (marked with dotted X's) are identified to be potentially heavy, and are each allocated $m_2 = 2$ auxiliary buckets. During the subsequent scan, when targets $\{a, b, p, q\}$ fall into the heavy buckets, they are rehashed using h_2 to their corresponding auxiliary buckets. Note that we do not explicitly store the targets in the auxiliary buckets as indicated in the figure; we continue to maintain counters in them.

The idea behind the MULTI-LEVEL algorithm is very similar to the concept of *extensible indices* commonly used in databases [Ull88] – these indices grow over-populated buckets by adding auxiliary buckets dynamically. However, the difference is that in the case of extensible indices the entire key that is being indexed is stored. Hence when buckets are over-populated, we can dynamically add auxiliary buckets efficiently. Recall that we cannot afford to store the targets explicitly in main memory, and can only maintain counters. This is the reason we perform the pre-scan to pre-allocate auxiliary buckets for potentially heavy buckets. Also notice that MULTI-LEVEL does not store the sample set explicitly like DEFER-COUNT does. This is useful especially when the size of targets is very large.

One problem with MULTI-LEVEL is that it splits a given amount of main memory between the primary and auxiliary buckets. Deciding how to split memory across these two structures is not a simple problem – we have to determine good splits for specific datasets empirically. Also, the cost of rehashing into the auxiliary buckets could be expensive, if a second hash function is employed. In practice, however, we can avoid this by using one hash function: we can use fewer bits for the first hashing, and use the residual bits to "hash" the target into the auxiliary buckets.

We now discuss one important detail for implementing the above scheme. In Figure 3.3, we maintain pointers to auxiliary buckets. In some cases, maintaining eight bytes per pointer may be expensive, especially if the number of potentially heavy buckets is high. In such cases, we can allocate all the auxiliary buckets for all potentially-heavy buckets contiguously in main memory starting at base address B. For the i^{th} potentially-heavy bucket, we can store in A the offset into the auxiliary buckets. We can then compute the auxiliary buckets for potentially heavy bucket $A[i]$, to be in locations $[B + (A[i] - 1) \times m_2, B + A[i] \times m_2)$.

3.5.3 MULTI-STAGE Algorithm

We now propose a technique that uses available memory in a different way compared to the MULTI-LEVEL algorithm. MULTI-STAGE has the same pre-scan sampling phase as MULTI-LEVEL, where it identifies potentially heavy buckets. However, MULTI-STAGE does not allocate auxiliary buckets per potentially heavy bucket. Rather it allocates a common *pool* of auxiliary buckets $B[1, 2, \ldots, m_3]$. Then it performs a hashing scan of the data as follows. For each target v in the data, it increments $A[h(v)]$ if the bucket corresponding to $h(v)$ is not marked as potentially heavy. If the bucket is so marked, apply a second hash function h_2 and increment $B[h_2(v)]$.

We present an example of this procedure in Figure 3.3 (c). We mark the common B arrays with $m_3 = 2$ using dotted lines. Note that the targets $\{a, b, p, q\}$ are remapped into the auxiliary buckets, using a second hash function that uniformly distributes the targets across the common pool of auxiliary buckets. It is easy to see that in this example there is a 50% chance that both the heavy targets p and q will fall into the same bucket. In such cases, targets a and b are no longer false positives due to p and q. Indeed in the figure, we present the case when p and q do fall into the same bucket. We have analysed MULTI-LEVEL based on the above intuition in Ref. [FSGM+98].

The main intuition behind sharing a common pool of auxiliary buckets across potentially heavy buckets is that several heavy targets when rehashed into B could fall into the same bucket as other heavy targets (as illustrated in the example). MULTI-LEVEL does not have this characteristic since the heavy targets are rehashed into their local auxiliary structures. Hence we can expect MULTI-STAGE to have fewer false positives that MULTI-LEVEL, for a given amount of memory.

MULTI-STAGE shares a disadvantage with MULTI-LEVEL in that determining how to split the memory across the primary buckets and the auxiliary buckets can only be determined empirically.

3.6 Optimizing HYBRID with MULTIBUCKET algorithms

The HYBRID algorithms may still suffer from many false positives if many light values fall into buckets with (1) one or more heavy targets, or (2) many light values. The sampling strategies we outlined in the last section alleviate the first problem to a certain extent. However the heavy targets not identified by sampling could still lead to several light values falling into heavy buckets. Also HYBRID cannot avoid the second problem. We now propose how to improve the HYBRID techniques of the last section, using multiple sets of primary and auxiliary buckets, to reduce the number of false positives significantly. We analyze the same idea in two different contexts, in the following subsections based on the number of passes required over the data.

For clarity, we describe the techniques of this section, in the context of the simple DEFER-COUNT algorithm, even though the techniques are also applicable to the MULTI-LEVEL, and MULTI-STAGE algorithms. Furthermore, for the techniques we present below we continue to perform the sampling scan to identify potentially heavy targets, and store

them in F. We do not count these targets during the hashing scans, but count them explicitly in the candidate-selection phase. After the candidate-selection phase we continue to execute $Count(F)$ to remove false positives. Since these steps are common to all the following techniques, we do not repeat these steps in the following discussion.

3.6.1 Single scan DEFER-COUNT with multiple hash functions (UNISCAN)

We illustrate UNISCAN using two hash functions h_1 and h_2 which map target values from $\log_2 n$ bits to $log_2(m/2)$ bits, $m << n$. The memory allocated is first divided into two parts for the two counting and bitmap arrays. That is, we now have $A_1[1..m/2]$, $A_2[1..m/2]$, $BITMAP_1[1..m/2]$ and $BITMAP_2[1..m/2]$. We then execute the pre-scan sampling phase in DEFER-COUNT and identify f potentially heavy candidates, and store them in F. Next, we do one pass over the input data and for each tuple in R with value v, $v \notin F$, we increment both $A_1[h_1(v)]$ and $A_2[h_2(v)]$ by one. Finally we set $BITMAP_1[i]$ to 1 if $A_1[i] \geq T$, $1 \leq i \leq m/2$. Similarly for $BITMAP_2$, and then deallocate A_1 and A_2. In the candidate-selection phase, we do one pass of the data and for each tuple with value v, we add v to F only if both $BITMAP_1[h_1(v)]$ and $BITMAP_2[h_2(v)]$ are set to one. We can easily generalize the above procedure for k different hash functions $h_1, h_2, \ldots, h_k$. As mentioned earlier, for now we assume that A, the k bitmaps, and F all fit in main memory. We will discuss our model for memory usage in Section 3.8.

Choosing the right value of k is an interesting problem, for a given amount of main memory. As we choose a larger value of k, we have many hash tables but each hash table is smaller. While the former helps in reducing the number of false positives, the latter increases the number of false positives. Hence there is a natural trade-off point for choosing k. We discuss in the Appendix how to choose a good value of k for UNISCAN.

3.6.2 Multiple scan DEFER-COUNT with multiple hash functions (MULTISCAN)

Rather than use multiple hash functions within one hashing scan and suffer an increased number of false positives due to smaller hash tables, we can use the same idea across multiple hashing scans as follows. After the sampling pre-scan, execute one hashing scan with hash function h_1. Store the corresponding $BITMAP_1$ array on disk. Now perform

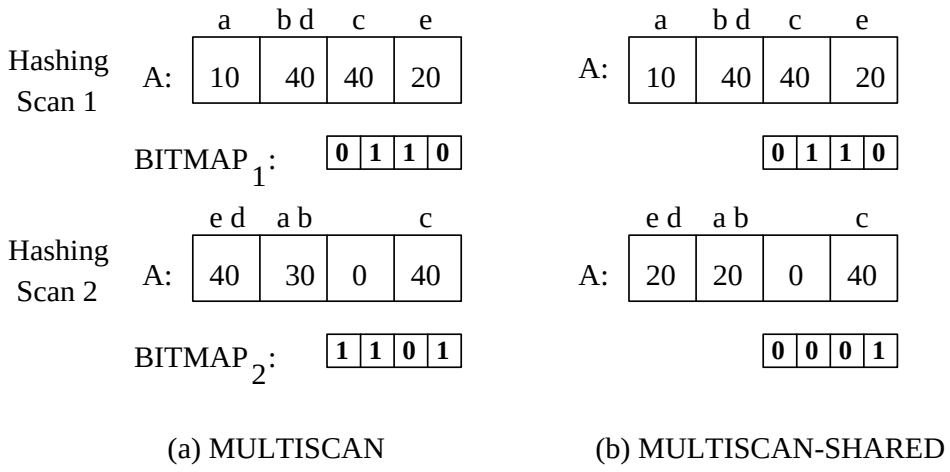


Figure 3.4: Comparing MULTISCAN versus MULTISCAN-SHARED.

another hashing scan with a different hash function h_2. Store the corresponding $BITMAP_2$ array on disk. After performing k hashing scans, leave the last $BITMAP$ in memory and retrieve the $k-1$ $BITMAP$ arrays from disk. Then execute the candidate-selection scan, and add value v to F if $BITMAP_i[h_i(v)] = 1, \forall i, 1 \leq i \leq k$.

3.6.3 MULTISCAN with shared bitmaps (MULTISCAN-SHARED)

In MULTISCAN we performed each hashing scan independently of the previous scans, even though the $BITMAP$ information from previous scans was available. In MULTISCAN-SHARED we assume that in the i^{th} hashing scan, bitmaps from all i previous hashing scans are retained in memory. This optimization works as follows: During the $(i+1)^{st}$ hashing scan, for target v, increment $A[h_{i+1}(v)]$ by one, only if $BITMAP_j[h_j(v)] = 1$, for all j, $1 \leq j \leq i$.

The following example illustrates how MULTISCAN-SHARED reduces the number of false positives over MULTISCAN. Consider the case when we have the following $\langle$ target, frequency $\rangle$ pairs in R: $\langle a:10\rangle$, $\langle b:20\rangle$, $\langle c:40\rangle$, $\langle d:20\rangle$, $\langle e:20\rangle$, i.e., target a occurs in ten tuples in R, b occurs in 20 tuples in R, and so on. Let $T = 30$, and $m = 4$. Let h_1 map the targets to the following buckets, set of targets pairs: $[0:\{a\}, 1:\{b,d\}, 2:\{c\}, 3:\{e\}]$ as shown in Figure 3.4, i.e., $h_1(a) = 0$, $h_1(b) = h_1(d) = 1$, etc. Similarly h_2 maps the targets to the following buckets $[0:\{e,d\}, 1:\{a,b\}, 2:\{\}, 3:\{c\}]$. In Figure 3.4(a) we show the

counts in array A and the corresponding $BITMAP$ after the first hashing scan when we execute MULTISCAN. Similarly we compute A and $BITMAP_2$ after the second hashing scan. Now in the candidate selection scan of MULTISCAN. we would choose $\{b, c, d\}$ to be part of F, since targets b, c, d fall into heavy buckets under both hash functions.

Now consider the execution of MULTISCAN-SHARED in Figure 3.4(b). The first hashing scan remains the same as before. The second scan however computes a different bitmap since the second hashing scan uses the information in $BITMAP_1$ before incrementing A. To illustrate, consider how e is counted by each algorithm in the second hashing scan. In MULTISCAN, $A[h_2(e)]$ is incremented for each of the 20 occurrences of e. However in MULTISCAN-SHARED, $A[h_2(e)]$ is not incremented for the 20 occurrences of e, since we already know that e is light (because $BITMAP_1[3] = 0$). Since e does not increment $A[0]$ in the second hashing scan, d is no longer a part of the candidate set. In fact in the candidate-selection scan, the only target chosen by the MULTISCAN-SHARED will be $\{c\}$, as opposed to the $\{b, c, d\}$ chosen by MULTISCAN.

3.6.4 Variant of MULTISCAN-SHARED (MULTISCAN-SHARED2)

We propose a variant of MULTISCAN-SHARED that uses less memory for $BITMAP$s. In this variant, we maintain the $BITMAP$s only from the last q hashing scans while performing the $(i+1)^{st}$ $(q \leq i)$ hashing scan, rather than maintaining all i prior $BITMAP$s. The conjecture is that the q latest $BITMAP$s from hashing scans $i - q + 1$ through i have fewer and fewer bits set to one. Therefore these $BITMAP$s have significant pruning power in terms of pruning away many light values at the cost of lower memory usage. We use MULTISCAN-SHARED2 to denote this algorithm.

3.7 Extending iceberg algorithms

In this section we briefly describe some variations to the schemes we presented earlier.

1. **Collapsing candidate-selection scan with final counting-scan:** The MULTISCAN algorithm (and its extensions that were proposed in Sections 3.6.3 and 3.6.4) performs k hashing scans, one candidate-selection scan, and finally one counting scan where false positives were eliminated. In cases where the size of F is expected to be small, we can collapse the last two scans into one as follows. When executing the candidate-selection scan, we add an in-memory counter to each element of F. In that

scan, as we add each target to F (because it appeared in heavy buckets for all k-hash functions), we check if the target was already in F. If so, we increment its counter; if not, we add it to F with its counter initialized to 1. We can dispense with the final counting-scan because we already have a count of how many times each F target appears in R. Targets whose count exceed the threshold are in the final answer.

2. **Parallelizing hashing scans for MULTISCAN:** We can parallelize the hashing scans of MULTISCAN across multiple processes. In such a case, the time for the hashing scans drops from the time for k sequential scans, to the time for a single scan. Of course, we cannot use the same optimization for MULTISCAN-SHARED and MULTISCAN-SHARED2 since they use bitmaps from previous iterations.

3. **Executing** FIND-ALL **when registered and query sets are different:** For simplicity, we assumed that the registered and query document sets are the same. However notice that the iceberg algorithms can be applied even when the two sets are different, as follows. Here we compute a sorted *DocChunk* relation for registered documents, and one for query documents. We then scan the two relations in parallel. For each registered document and query document that share a chunk, we apply our iceberg algorithms to count the number of shared chunks and threshold with T.

3.8 Case studies

We have discussed a relatively large number of techniques each of which are parameterized in different ways (e.g., how much of data we should sample, s and how many values to retain to be potentially heavy). Given the choices, it is difficult to draw concrete conclusions without focusing on particular application scenarios. Rather than explore our techniques only in the context of our FIND-ALL operation, in this chapter we evaluate our techniques for a few different iceberg queries (including FIND-ALL). Our goal here is to better understand the scenarios under which particular techniques work well (e.g., for what data distributions and choice of parameters).

Specifically, we chose three distinct application scenarios and designed our experiments to answer the following questions: (1) How does each scheme perform as we vary the amount of memory allocated? We report the performance both in terms of number of false positives ($|F|$) produced, and the total time each scheme takes to produce F, as well as to remove the

false positives using $Count(F)$. (2) How does each scheme perform as we vary the threshold? As above, we report both $|F|$ and the total time. (3) How do schemes perform for different data distributions? That is, if the input data follows a skewed Zipfian distribution [Zip49] (also called "$80-20$" distribution), as opposed to less skewed distributions, how are the schemes affected by sampling?

Before we present our results, we discuss how we allocate memory in our experiments. We experimented with a variety of ways to split the available memory between the sample set of size f (in case of DEFER-COUNT based algorithms), the primary and the auxiliary buckets. We found the following approach to work best for our data.

1. **Allocate f:** For algorithms based on DEFER-COUNT, choose a small f for the sampling scan and allocate memory for that set. We discuss later what should be the value of f, for each application.

2. **Allocate auxiliary buckets:** Allocate p_{aux} percent of the remaining memory after the first step to auxiliary buckets. As the algorithm executes we may discover that this amount of allocated memory was insufficient for the auxiliary buckets. If that happens, we greedily select the buckets with highest A counter values, and assign as many of these as possible to the auxiliary area. The remaining potentially heavy buckets, that could not be assigned to the limited auxiliary area, are treated as any other primary bucket during the hashing scan.

3. **Allocate primary buckets and bitmaps:** Allocate the balance of the memory to the primary buckets and their bitmaps. In case of UNISCAN we need to this memory among the k primary buckets and their bitmaps (based on the value of k chosen by the analysis in the Appendix).

In our experiments, we found p_{aux} between $15-20\%$ to be good values for splitting up our memory. Before the candidate-selection scan, we reclaim the memory allocated to the primary buckets and allocate that to store F.

In the following experiments, if the final F (input to $Count(F)$) does not fit in main memory, we stream the tuples in F onto disk, and we execute $Count(F)$ using a disk-based sorting algorithm. Our implementation is enhanced with *early aggregation* [BD83] so that it integrates counting into the sorting and merging processes, for efficient execution. As we discussed earlier, this algorithm is merely one way to execute $Count(F)$. Hence the

reader should not interpret the results of this section as absolute predictions, but rather as illustrations of performance trends. For the following experiments, we used a SUN ULTRA/II running SunOS 5.6, with 256 MBs of RAM and 18 GBs of local disk space.

Case 1: Market basket query

First, we evaluate our techniques for the market basket query that finds commonly occuring word pairs. For our experiments, we use 100,000 web documents crawled and stored by the Google webcrawler [BP99]. The average length of each document is 118 words. From this data we computed the *DocWord* relation to be ⟨`docID, wordID`⟩, if document with identifier `docID` had a word with identifier `wordID`. This relation was about 80 MBs, when we used 4-byte integers for `docID`s and `wordID`s. Note that we removed entries corresponding to 500 pre-defined stop words from this relation [SB88]. Recall that the R over which the iceberg query is to be executed has all pairs of words that occur in the same document. If R were to be materialized on disk, it would require about 29.4 GBs to store R; in addition, we may require temporary storage while performing the aggregation. Since this storage requirement is high even for a small input size, we do not discuss this technique any more in this section.

We can answer iceberg queries without explicitly materializing R as discussed in our extension for executing FIND-ALL when registered and query sets are different. That is, we can simply scan *DocChunk* and produce $\langle c_i, c_j \rangle$ for each c_i, c_j pair that occurs in the same document. Rather than explicitly storing such tuples, we stream the tuples directly to the algorithm we use to execute the iceberg query. For instance, if we use DEFER-COUNT to execute the iceberg query (assume $s = 0$), increment $A[h(c_i, c_j)]$ as soon as tuple $\langle c_i, c_j \rangle$ is produced. Notice that we cannot apply a similar optimization for sorting or hybrid hashing based schemes, since the tuples have to be materialized explicitly (for sorting), or will need to be stored in the hash table (for hybrid hashing).

We now discuss a few representative schemes for specific values of K to illustrate some of the trade-offs involved. Specifically, we present results for MULTISCAN/D, MULTISCAN-SHARED/D and UNISCAN/D, the corresponding multi-bucket optimization of DEFER-COUNT. We also evaluate MULTI-STAGE for $K = 1$. We found a 1% sample of n ($s = 1\%$) and $f = 1000$ to work well in practice for this data.

In Figure 3.5 we show how $|F|$, the number of candidate pairs, varies as the amount of memory allocated increases. We see that $|F|$ drops as more memory is allocated, as expected. Also we see that MULTISCAN/D [K = 2] and MULTISCAN-SHARED/D [K

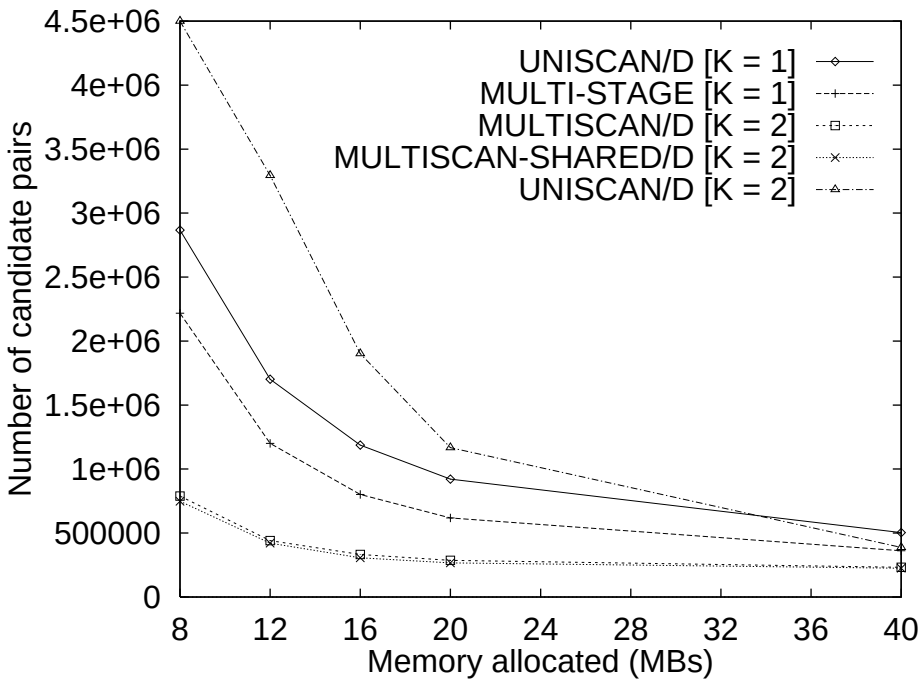


Figure 3.5: $|F|$ as memory varies ($T = 500$).

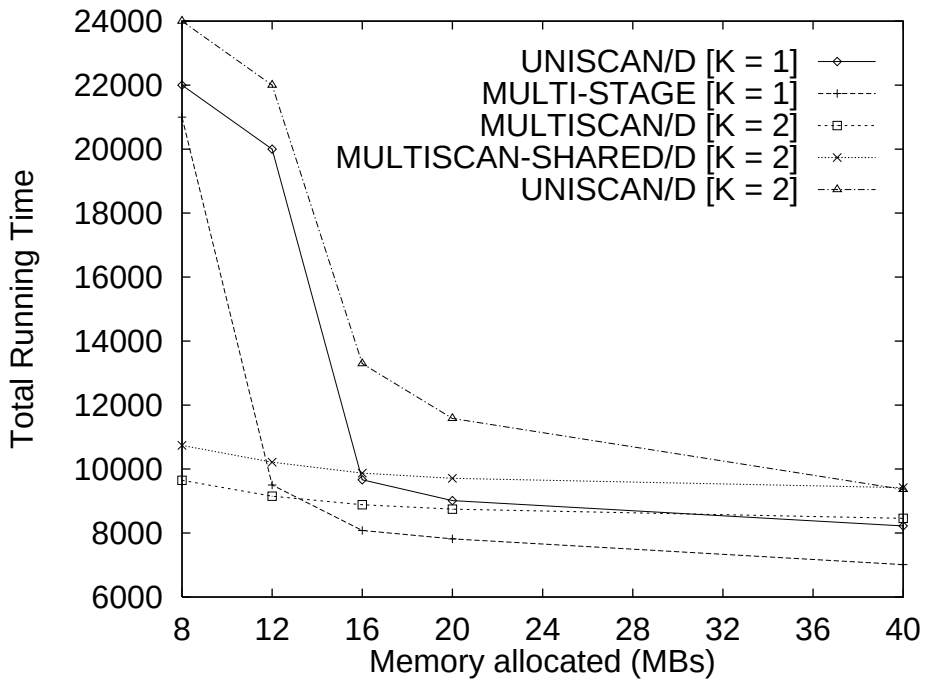


Figure 3.6: Total time as memory varies ($T = 500$).

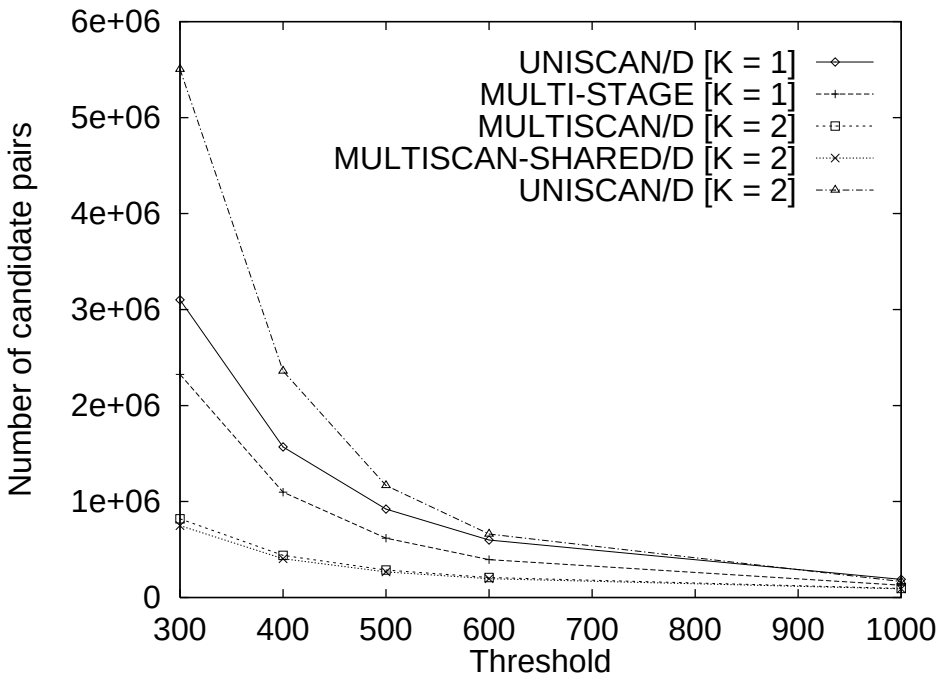


Figure 3.7: $|F|$ as threshold varies ($M = 20$ MB).

= 2] perform best, in terms of choosing the smallest $|F|$. This improvement is because when the amount of memory is small, doing multiple passes over the data using most of the available memory for the A array, helps prune the number of false positives significantly. UNISCAN/D [K = 2] performs poorly initially since the amount of main memory is very small, but the difference between UNISCAN/D [K = 1] and UNISCAN/D [K = 2] drops with larger memory. For memory more than about 34 MBs, we see that UNISCAN/D [K = 2] performs better than its $K = 1$ counterpart.

In Figure 3.6 we see the total time to answer the iceberg query as the amount of memory varies. We see that MULTISCAN/D and MULTISCAN-SHARED/D perform steadily across the different memory sizes, since they do not produce too many false positives. On the other hand, MULTI-STAGE [K = 1] performs badly when memory is limited; beyond about 14 MBs it performs best. This behavior is because (1) the number of false positives is relatively small and hence counting can be done in main memory, (2) MULTI-STAGE scans the data one less time, and uses less CPU time in computing fewer hash functions than the other multi-bucket algorithms (such as MULTISCAN/D).

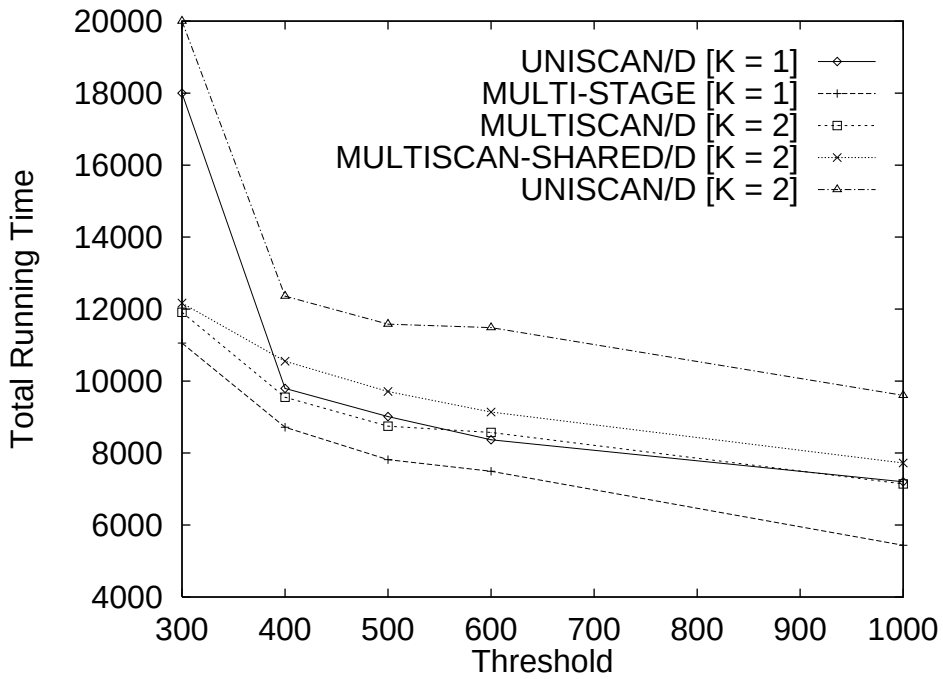


Figure 3.8: Total time as threshold varies ($M = 20$ MB).

In Figure 3.7 we study how $|F|$, the number of candidates, varies as the threshold is varied. We see that MULTISCAN/D [K = 2] and MULTISCAN-SHARED/D [K = 2] tend to have the smallest $|F|$. Again, we see that performing multiple passes over the data using multiple hashing functions helps prune away many false positives. In Figure 3.8 we see the corresponding total time to answer the iceberg query. We see that MULTI-STAGE performs the best in this interval, again because (1) F is relatively small, and (2) it performs one fewer scan over the data, and needs to compute fewer hash functions than MULTISCAN/D and MULTISCAN-SHARED/D.

In summary, we see that MULTI-STAGE works best since this application had very little data.

Case 2: Computing StopChunks

We now consider how sensitive our schemes are to skews in data distribution, using an IR example. We discussed in Section 3.3.1 how IR systems compute a set of stop words for efficiency. In general, IR systems also compute "stop chunks," which are syntactic units of text that occur frequently. By identifying these popular chunks, we can improve phrase searching and indexing. For instance, chunks such as "Netscape Mozilla/1.0" occur frequently in web documents and may not even be indexed in certain implementations of IR systems (such as in [SGM95, SGM96]), to reduce storage requirements.

For this set of experiments, we used 300,000 documents we obtained from the Stanford Google crawler (as above). We defined chunks based on *sliding windows* of words as in Chapter 2. We say we use "$C = i$" chunking, if the j^{th} chunk of a given document is the sequence of words from j through $j + i - 1$. For a corpus of documents, we compute the *DocChunk*$(C = i)$ relation as we discussed earlier. For our experiments we computed four different *DocChunk* tables for $C = 1, 2, 5, 10$.

Our first two graphs illustrate the nature of the data, and not a specific algorithm. In Figure 3.9 we show, on a log-log plot, the frequency-rank curves of the four different chunkings. As expected, the smaller the C used to construct a chunk, the fewer the distinct target values, and the larger the data skew. For instance, with $C = 1$, the number of distinct chunks, n, is over 1.5 million, and the heaviest target occurs about 4.5 million times in *DocChunk*. For $C = 10$, $n = 27.7$ million, while the heaviest target occurs only 0.21 million times. The size of each *DocChunk* relations was about 4.2 gigabytes (Note that we did not remove pre-computed stop words from these relations as we did in the

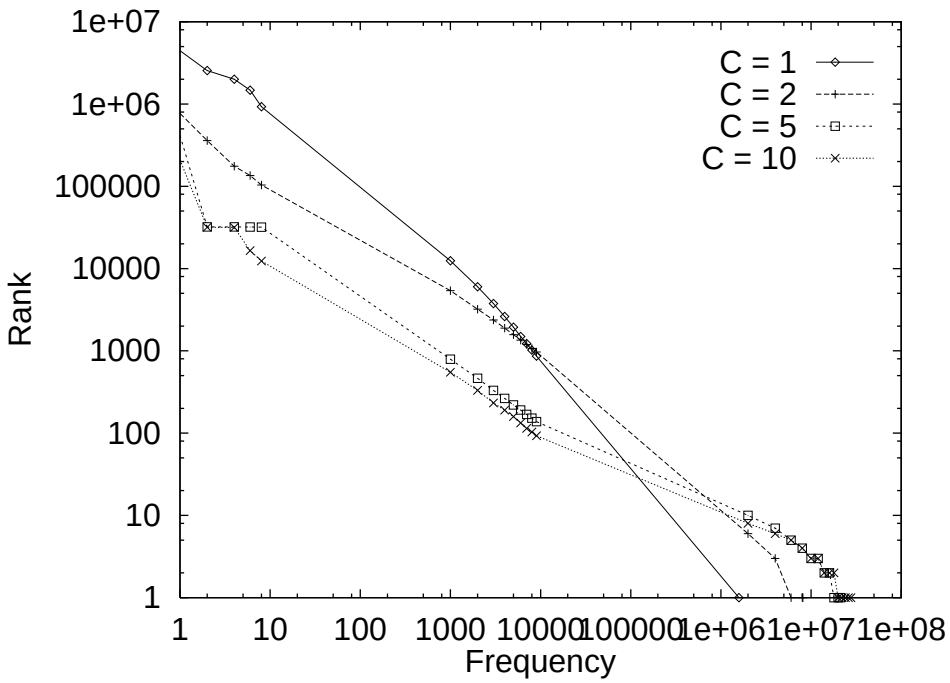


Figure 3.9: Frequency-rank curves for different chunkings.

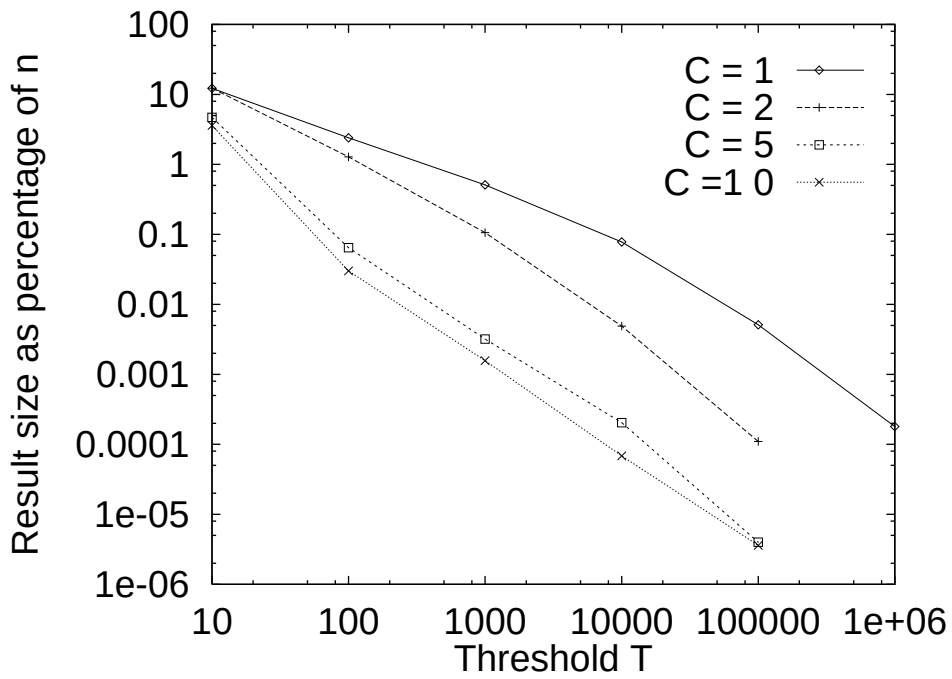


Figure 3.10: Result sizes for different thresholds.

market-basket query.)

In Figure 3.10 we show (again on a log-log plot) what percentage of the n unique terms are actually heavy, for different thresholds. We see in the figure that, as expected, the number of heavy targets (the tip of the iceberg) drops significantly as T increases.

In the following two graphs, Figure 3.11 and 3.12, we study how the number of hashing scans K, and the number of hash buckets m affect false positive errors. We present the results of MULTISCAN-SHARED2/D, with $q = 2$ (the number of previous bitmaps cached in memory). The vertical axis in both figures is the percentage of false positives ($100 * \frac{FP}{n}$, where FP is the number of false positives). As we expected, the percentage of false positives drops dramatically with increasing k. For instance for $C = 1$, the percentage drops from about 70% for $k = 1$ to less than 10% for $k = 4$. Also it is interesting to note that the number of false positives drops as the data is less skewed (from $C = 1$ through $C = 10$), especially as the number of hashing scans increases. We attribute this behavior to three factors: (1) there are fewer heavy targets (Figure 3.10), (2) since data is not very skewed, fewer light targets fall into buckets that are heavy due to heavy targets, and (3) as more hashing scans are performed, fewer light targets fall into heavy buckets across each of the hashing scans.

In summary, these experiments quantify the impact of skew, and provide guidelines for selecting the number of hashing scans needed by MULTISCAN-SHARED2/D, as the "tip of the iceberg" changes in size. We observe similar behavior for the other schemes.

Case 3: FIND-ALL operation

In Figure 3.13 we present the total time to execute the FIND-ALL operation using MULTISCAN and MULTISCAN-SHARED techniques as the amount of memory (M) changes. We executed the query on the *DocChunk* relation from Case 2, when $C = 1$. Since the data was unskewed for this query, we avoid the sampling scan, i.e., $s = 0\%$.

In Figure 3.13 we see that MULTISCAN-SHARED2 [q = 1] performs best, when the amount of memory is small, but progressively becomes inferior to MULTISCAN and MULTISCAN-SHARED as memory increases. MULTISCAN-SHARED [q = 2] is in between MULTISCAN-SHARED [q = 1] and MULTISCAN-SHARED, for small values of memory. The above behavior of MULTISCAN-SHARED2 compared to MULTISCAN-SHARED is due to the following competing factors: (1) MULTISCAN-SHARED2 uses fewer bitmaps than MULTISCAN-SHARED, thereby allocating more memory for primary

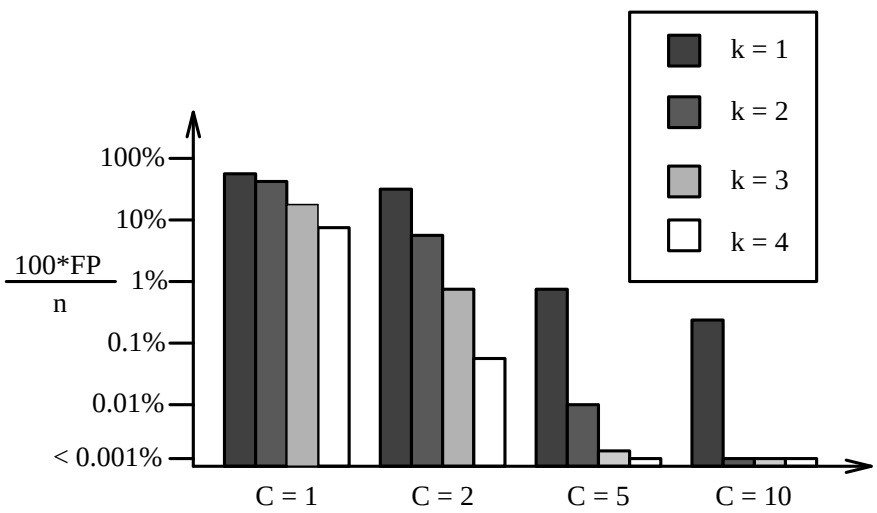


Figure 3.11: Performance of MULTISCAN-SHARED2/D with k ($T = 1000, m = 1\%$ of n).

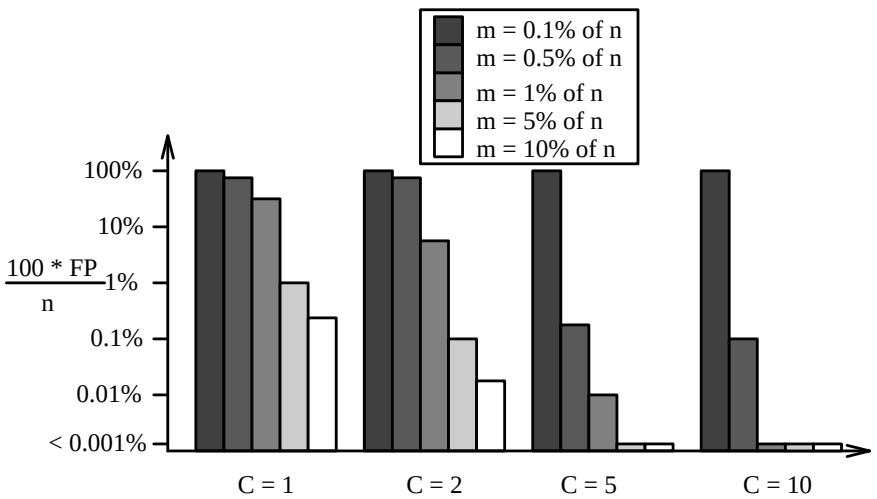


Figure 3.12: Performance of MULTISCAN-SHARED2/D with m ($T = 1000, k = 2$).

buckets. (2) For a given amount of memory, MULTISCAN-SHARED prunes more light targets than MULTISCAN-SHARED2, as we discussed earlier. For small values of memory, MULTISCAN-SHARED2 performs better than MULTISCAN-SHARED, since the first factor dominates. For larger values of memory, the extra space allocated to the additional bitmaps for MULTISCAN-SHARED still leaves enough memory for the primary buckets. Hence the second factor dominates. We also see that MULTISCAN does not perform too well for small memory, since it does not use bitmaps to prune away light targets, as we discussed earlier. Hence we see that choosing $q = 1$ or 2 may be useful for small sized memory while still leaving sufficient main memory for primary buckets.

The size of R, if materialized, is 52 GBs. If we assume disks can execute sequential scans at the rate of 10 MB/sec, it would take $52 * 1024/10 \approx 5300$ seconds each to read and write R. However, notice that MULTISCAN-SHARED2 [q = 1] would be done executing even before R is written once and read once! Of course, since R has to be sorted to execute the iceberg query, it is easy to see that sorting based execution would require too much disk space to materialize and sort R, and will take much longer than our schemes.

In this chapter, our focus was primarily on evaluating different algorithms to execute the CDS operations. Hence our experiments were on relatively small data sets. In the next chapter, we show that our techniques indeed scale by considering much larger data sets.

3.8.1 Summary

Based on our case studies, we propose the following informal "rules of thumb" to combine schemes from the HYBRID and MULTIBUCKET algorithms:

1. **HYBRID algorithms:** MULTI-LEVEL rarely performs well in our experiments, while DEFER-COUNT and MULTI-STAGE tend to do very well under different circumstances. If you expect the data distribution to be very skewed (such as in Zipfian distributions [Zip49] where very few targets are heavy, but constitute most of the relation), use DEFER-COUNT with a small f set. If you expect the data not to be too skewed, use MULTI-STAGE since it does not incur the overhead of looking up the values in f. If you expect the data distribution to be flat, do not use the sampling scan.

2. **MULTIBUCKET algorithms:** In general we recommend using MULTISCAN-SHARED2 with $q = 1$ or $q = 2$. For relatively large values of memory, we recommend

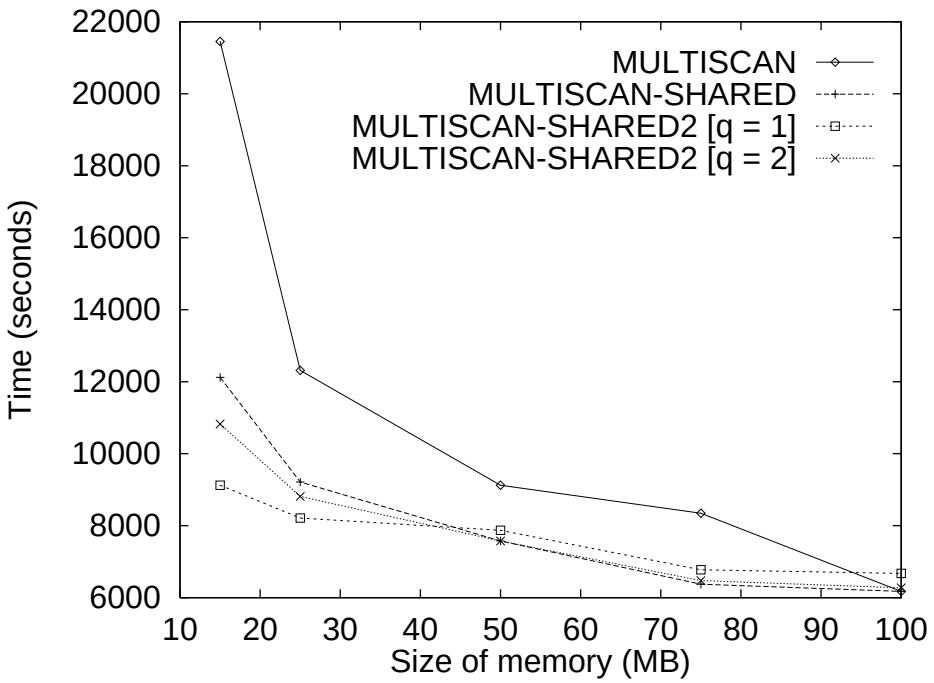


Figure 3.13: Performance of algorithms with M for DocumentOverlap query for $C = 1$.

UNISCAN with multiple hash functions, since we can choose $K > 1$ and apply multiple hash functions within one hashing scan.

3.9 Conclusion

In this chapter we first briefly discussed how to efficiently execute the FIND operation. Then we focussed on how to execute FIND-ALL operations. We noted that FIND-ALL is a special instance of an *iceberg* query, an important class of queries with widespread application in data-warehousing, data mining and information retrieval. We then showed that traditional approaches to compute such queries do not scale when the amount of data is large, since these techniques are not output sensitive. We then proposed algorithms that compute the result, the "tip of the iceberg," much more efficiently than conventional schemes. We evaluated our algorithms using a case study approach in three real applications (including our FIND-ALL operation), and observed that the savings are indeed very significant.

Mirror, mirror on the wall, who is the fairest of us all?

– Snow White and the Seven Dwarfs, Grimm Brothers

Chapter 4

Running Text CDS on the Web

4.1 Introduction

In this chapter, we report results of running our text CDS on 60 million web pages, about 400 gigabytes of textual data crawled by the Google crawler [BP99]. We organized these experiments with the following two goals in mind.

1. We show that the algorithms we discussed in Chapter 3 indeed scale for a large document corpus. Here we report timing results and storage requirements for some of our techniques.

2. We show that the textual features we extract and the algorithms we proposed in this dissertation are very general and are applicable in domains other than copyright protection.

With these goals in mind, we organize this chapter as follows. We first motivate some issues that arise due to *mirroring* of textual content on the Internet. Then we discuss how our techniques can be applied to identify such mirroring automatically and to improve a variety of different applications. We then show report how our techniques perform for these different applications.

4.2 Automatic detection of mirrored collections

Many documents are replicated across the web. For instance, a document containing JAVA Frequently Asked Questions (FAQs) is replicated at many sites. Furthermore, entire "collections" of hyperlinked pages are often copied across servers for fast local access, higher availability, or "marketing" of a site. We illustrate such replicated collections in Figure 4.1. The figure shows four actual sites that make available two collections: the JAVA 1.0.2 API documentation, and the Linux Documentation Project (LDP) manual. Other examples of commonly replicated collections include JAVA tutorials, Windows manuals, C++ manuals and even entire web sites such as ESPN's Sportszone and Yahoo. In general, these collections constitute several hundreds or thousands of pages and are *mirrored*[1] in several tens or hundreds of web sites. For instance, the LDP collection is a 25 megabyte collection of several thousand pages, mirrored in around 180 servers across the world.

Our goal in this chapter is to study how to automatically identify mirrored "collections" on the web, so that a variety of tasks can be performed more effectively. These tasks include:

- **Crawling:** A crawler fetches web pages for use by search engines and other data mining applications. A crawler can finish its job faster if it skips replicated pages and entire collections that it has visited elsewhere.

- **Ranking:** A page that has many copies, in a sense, is more important than one that does not. Thus, search results may be ranked by replication factor. If a replicated page is a member of a collection, it may also be useful to indicate this in the search result, and perhaps to provide some main entry point to the collection (e.g., its root or home page).

- **Archiving:** An archive stores a subset of the web, for historical purposes [Ale99]. If the archive cannot store the entire web, it may give priority to collections that are known to be replicated, because of their importance and because they may represent coherent units of knowledge.

- **Caching:** A cache holds web pages that are frequently accessed by some organization. Again knowledge about collection replication can be used to save space. Furthermore,

[1] The term mirror is often used to denote a replicated collection or web site; often a mirror site has the connotation of being a "secondary copy."

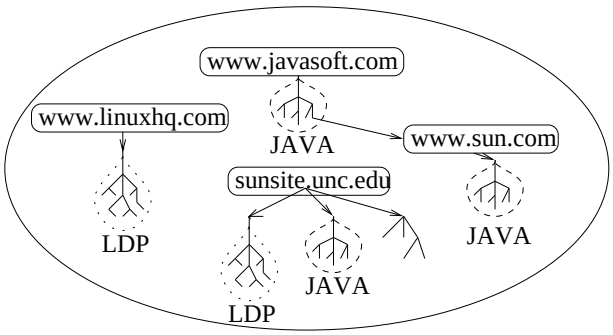


Figure 4.1: Mirrored document collections.

caching collections as a whole may help improve hit ratios (e.g., if a user is accessing a few pages in the LDP, chances are he will also access others in the collection).

As we will see in this chapter, replication is widespread on the web, so the potential savings are very significant for each of the above tasks. For instance, consider the potential savings to the crawling task. A crawler that only visits one of the 180 LDP collections can avoid fetching about 4.5 gigabytes (25 MB * 180) of data. Indeed we will see in Section 4.8 that the crawler saves significant (over 40%!) amounts of bandwidth and time by avoiding major mirrored collections, such as LDP, JAVA API documents and Linux manuals. We will also see that we gain significant improvements in other tasks listed above as well. Recently, Bharat and Broder [BB99] observed similar results on data crawled by the AltaVista web crawler.

We have intuitively motivated the usefulness of detecting "replicated collections." However, there are important challenges in (1) defining the notion of a replicated collection precisely, (2) developing efficient algorithms that can identify such collections, and (3) effectively exploiting this knowledge of replication. One of the major difficulties in detecting replicated collections is that many replicas may not be strictly identical to each other. The reasons include:

1. **Update frequency:** The primary copy of a collection may often be constantly updated, while mirror copies are updated only daily, weekly, monthly, or by operator control. For instance, the Javasoft corporation maintains the primary JAVA API manuals, and the University of Edinburgh hosts the LDP collection. These primary copies

are regularly updated to incorporate the most recent bug fixes and documentation updates. However the mirrors of these collections are usually out of date, depending on how often they are updated.

2. **Mirror partial coverage:** Mirror collections differ in how much they overlap with the primary. In many cases, a collection is replicated entirely. In other cases, only a subset of a collection may be mirrored, and hyperlinks point to uncopied portions in another mirror site, or at the primary.

3. **Different formats:** The documents in a collection may *not* themselves appear as exact replicas in another collection. For instance, one collection may have documents in HTML while another collection may have them in PostScript, Adobe PDF or Microsoft Word. Similarly, the documents in one collection may have additional buttons, links and inlined images that make them slightly different from other versions of the document. Still, for some applications we may wish to consider two collections in different formats to be "similar."

4. **Partial crawls:** In most cases we need to identify replicated collections from a snapshot of the web that has been crawled or cached at one site, not by examining the original data. Thus, the data we are examining may be incomplete, e.g., because the crawler does not have the resources to fetch all data. So even if two collections are perfect replicas, their snapshots may not overlap fully. Furthermore, if the snapshots were crawled at different times, they may represent different versions, even if the originals were in perfect synchrony.

In such a scenario, it is challenging to even define what is a replicated collection, or even what is the "boundary" of a collection. For example, suppose that one site has a set S_1 of 10 pages, a second site has a set S_2 that is an exact copy of S_1, and a third site has a set S_3 containing only 5 of the S_1 pages. In this case we have at least two choices: we could say that S_1 is a replica of S_2 (ignoring S_3), or we could say that there are two additional replicas of the S_3 collection, each with 5 replicated pages. That is, in the former case we have a 10-page collection at two sites, and in the latter case we have a 5-page collection at 3 sites. If we wish to consider page and link structure similarity, we have even more choices. For instance, suppose that only 8 of the 10 pages in S_2 are identical copies of their S_1 counterparts. The remaining 2 are variants, e.g., they may have slightly different

content, or may have missing or extra links. Do we still consider S_1 and S_2 to be replicas, or do we only consider corresponding sub-collections with 8 identical pages to be replicas? In the former case we have a larger collection, but in the latter case we have a more faithful replica. In this chapter we study these issues, and propose mechanisms for making such decisions.

The amount of data on the web is staggering, on the order of hundreds of millions of pages and hundreds of gigabytes, and growing rapidly. Thus, whatever techniques we use for identifying replicated pages and collections must scale to very large sizes. So we develop new techniques for analyzing and manipulating our data.

After replicated collections have been identified, a final challenge is how to exploit effectively this information to improve crawling, ranking, archiving, caching and other applications. For instance, should a crawler try to identify replicas as it performs its first crawl, or should it first do a full crawl, then identify replicas, and skip mirrors on subsequent crawls? If replicas are skipped, should the crawler still visit them infrequently to make sure they are still copies? In this chapter while we do *not* focus on this category of challenges, we briefly touch on some of the crawling and ranking issues in Section 4.8, where we present our experimental results.

In summary, the contributions of this chapter are the following:

- **Computable similarity measures:** We define similarity measures for collections of web pages, and study options for defining boundaries of replicated collections. We carefully design these measures so that in addition to being "good" measures, we can efficiently compute these measures over hundreds of gigabytes of data on disk. Our work is in contrast to recent work in the Information Retrieval domain [PP97] where the emphasis is on accurately comparing link structures of document collections, when the document collections are small. We then develop efficient heuristic algorithms to identify replicated sets of pages and collections.

- **Improved crawling:** We discuss how we use replication information to improve web crawling by avoiding redundant crawling in the Google system. We report on how our algorithms and similarity measures performed on over 400 gigabytes of textual data crawled from the web (about 60 million web pages). One immediate impact of our work is that we have subsequently designed and built the *Pita* crawler, a new crawler that exploits the replication information we show how to compute in this chapter.

This new crawler now feeds data to a variety of data mining research projects at Stanford without taxing the networking or storage resources available.

- **Reduce clutter from search engines:** Finally, we discuss how we used replication information for another task, that of improving how web search engines present search results. We have built an alternative search interface to the Google web search engine. This prototype shows significantly less clutter than in current web search engines, by clustering together replicated pages and collections. Based on informal user testing, we have observed this simple new interface helps the user quickly focus on the key results for a search over web data.

In Sections 4.3, 4.4, and 4.5 we discuss our similarity measures for pages and collections. In Sections 4.5 and 4.6.2 we discuss how we compute these measures. Finally, in Sections 4.8 and 4.8 we discuss our experimental results over 60 million web pages.

4.3 Similarity of web pages

A web page at a web site has two properties: (1) the text content, and (2) hyperlinks (with anchor text) to other pages. Henceforth when we refer to a page, we refer only to its text content. We refer to the hyperlinks in the web page as the page's hyperlinks.

Consider two pages p_i and p_j, where the URL of p_i is `http://machine`$_i$`/filename`$_i$ and the URL of p_j is `http://machine`$_j$`/filename`$_j$. We can define pages p_i and p_j to be identical (i.e., $p_i \equiv p_j$) or similar (i.e., $p_i \cong p_j$) in many ways, including the following:

- **Location based:** Pages p_i and p_j are identical if (1) the IP addresses of `machine`$_i$ and `machine`$_j$ are the same, and (2) `filename`$_i$ and `filename`$_j$ are textually the same. For instance, the pages corresponding to `http://www-db.stanford.edu/A/B/C.html` and `http://phoebe.stanford.edu/A/B/C.html` are exact copies because `www-db` and `phoebe` have the same IP address, and the filename portion of the URLs (`A/B/C.html`) are textually the same.

  By using location information to compute page similarity, we can denote pages to be identical even before retrieving pages. However it is hard to generalize the above idea to compute similarity between pages in machines with unrelated IP addresses, even if they offer the same content.

- **Content based:** As discussed in Chapter 2, pages p_i and p_j are identical if their text content is exactly the same. Also we can compute similarity between p_i and p_j based on the number of chunks of text they share in common. While this definition is simple and purely syntactic, we discussed earlier why it is effective in approximating the human notion of page similarity. (Also see [BGM97, BB99].) Hence in the rest of the chapter, we use this definition for computing similarity between web pages.

We now expand on how to define page similarity in the content-based approach. We first convert each page from its native format (e,g., HTML, PostScript) into simple textual information using standard converters[2] on UNIX. The resulting page is then chunked into smaller textual units (e.g., 5-grams) as discussed in Chapter 2. Each textual chunk is then hashed down to a 32-bit *fingerprint*. For threshold T, recall we defined the chunk-based similarity measure as follows.

Definition 4.3.1 (**Page similarity**) We define the similarity between pages p_i and p_j, $sim(p_i, p_j)$, to be the number of fingerprints p_i and p_j share in common. We also define $p_i \cong p_j$ if $sim(p_i, p_j) \geq T$, for threshold T. Finally, we define $p_i \approx p_j$ if there is some sequence of pages $p_1, p_2, \ldots, p_k$ in our data-set, such that $p_i \cong p_1$, $p_2 \cong p_3$, $\ldots, p_k \cong p_j$. □

4.4 Similarity of collections

We find it useful to define the following terms.

Definition 4.4.1 (**Web graph**) Given a set of web pages, the web graph $G = (V, E)$ has a node v_i for each web page p_i, and a directed edge from v_i to v_j if there is a hyperlink from page p_i to p_j. □

Definition 4.4.2 (**Collection**) A collection is an induced subgraph[3] of the web graph, where the vertices correspond to pages in a single web site. A collection is a *full-collection* if the collection contains all pages from a web site. The number of pages in the collection is the *collection size*. Collection C' is a *subcollection* of C, if it is an induced subgraph of C. □

[2]Converters such as `ps2ascii` and `html2ascii` are not always exact converters and cannot handle non-ASCII objects such as figures and equations in the text. But for our application, we believe this problem is not significant.

[3]Recall that an induced subgraph $G' = (V', E')$ of a graph $G = (V, E)$, only has edges between the vertices in the subgraph.

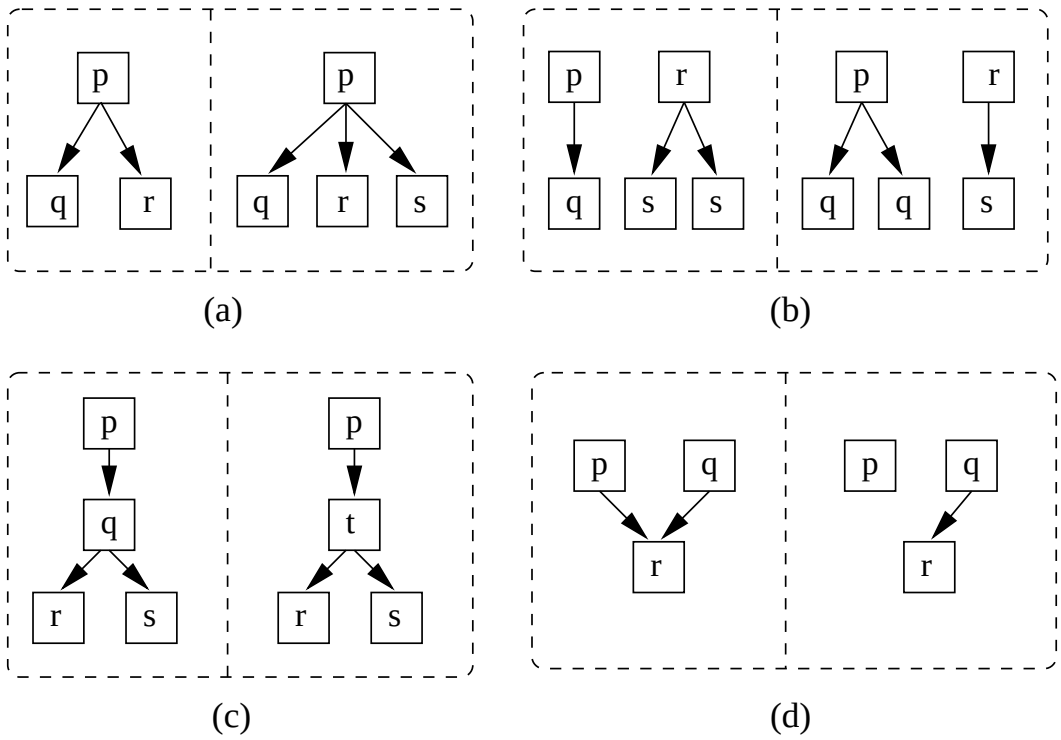


Figure 4.2: Example pairs of collections.

Definition 4.4.3 (**Identical**[4] **collections**) Equi-sized collections C_1 and C_2 are identical ($C_1 \equiv C_2$) if there is a one-to-one mapping M that maps all C_1 pages to C_2 pages (and vice-versa) such that:

- **Identical pages:** For each page $p \in C_1$, $p \equiv M(p)$. That is, the corresponding pages are identical.
- **Identical link structure:** For each link in C_1 from page a to b, we have a link from $M(a)$ to $M(b)$ in C_2.

□

We now consider how to extend the above definition to similar collections. That is, how do we define a mapping M that maps pages from one collection to another (and vice-versa)

[4]These are sometimes referred to as *label preserving isomorphs* in graph terminology. We use the term *identical* to be consistent with other definitions we introduce.

so that the mapped pages and link structures are "similar." Before we present different issues in defining similar collections, we wish to reiterate that our goal in defining similar collections is two-fold: (1) our definition matches a human's notion of similar collections, and (2) we can compute similar collections from the web graph automatically and efficiently. In this context, we first discuss different issues that arise in defining these collections. In each case, we make certain choices primarily for efficiency considerations, and later justify our choices experimentally by showing our measures help us identify mirrored collections.

We now illustrate some dimensions to consider while defining collection similarity using the example collection pairs in Figures 4.2 (a) – (d). In each example, we refer to the first collection as $[i]$ and the second collection as $[ii]$. We use squares to denote web pages and page identifiers $p, q, \ldots, z$ to identify similar pages (i.e., $[i].p$ is similar to $[ii].p$ in Figure 4.2 (a)). We use arrows to depict the hyperlink[5] structure (i.e., page $[i].p$ has a hyperlink to page $[i].q$).

- **Collection sizes:** In Figure 4.2 (a), we show two collections. They are identical except for the additional page $[ii].s$ and the link from $[ii].p$ to $[ii].s$. When defining collection similarity, we can either require collections to be equi-sized, or not. One advantage in requiring collections to be equi-sized is that computing similarity is easier (as we will see soon) and more natural. On the other hand, this requirement is restrictive, since two collections with several pages each may be identical except for one page. In this dissertation, we require similar collections to be equi-sized.

- **One-to-one:** In Figure 4.2 (b), we show two equi-sized collections. Notice that $[ii].p$ points to two copies of $[ii].q$, while $[i].p$ points to one copy of $[i].q$. Similarly, with $[i].r$ and $[ii].r$ pointing to two copies of $[i].s$ and one copy of $[ii].s$ respectively. In this case, we can easily compute a mapping M that maps all pages in one collection to some similar page in the other collection. However, this mapping is not one-to-one. When defining collection similarity, we can either require the mapping M to be one-to-one, or not. In this dissertation, we require the mapping to be one-to-one for simplicity.

- **Link similarity:** In Figure 4.2 (c) we see that the two link structures look similar, with the exception that pages $[i].q$ and $[ii].t$ are different. We can define the two collections to be similar since we can compute a mapping M that maps $[i].p$ to $[ii].p$,

[5] We do not show anchor texts on hyperlinks in this example for simplicity.

$[i].r$ to $[ii].r$ and $[i].s$ to $[ii].s$ (and vice versa). However, $[i].q$ and $[ii].t$ have different content and cannot be mapped to each other. When defining collection similarity, we can either require all pages to be mapped, or some large fraction of pages to be mapped. As we will see later, we can compute collection similarity more efficiently when we require all pages to be mapped. Hence in this dissertation, we require all pages to be mapped.

Figure 4.2 (d) highlights another issue. The two collections will be dissimilar if we require the following: For each page p_i in a collection and its mapped page $M(p_j)$, all parents of p_i should be mapped to parents of $M(p_j)$. For instance, according to this requirement $[i].r$ and $[ii].r$ cannot be mapped onto each other, since $[i].r$ has two parents and $[ii].r$ has only one parent. Alternatively, we may allow similar pages to be mapped to each other, as long as at least one of their parents (if any) are also mapped to each other. We will see later that this requirement allows us to compute collection similarity efficiently.

Based on the above considerations, we choose the following definition that makes specific choices for each of the above dimensions. As we will see in Section 4.8, this definition is a good definition and closely matches a human's notion of similarity.

Definition 4.4.4 (Similar collections) Equisized collections C_1 and C_2 are similar (i.e, $C_1 \cong C_2$) if there is a one-to-one mapping M (and vice-versa) that maps all C_1 pages to all C_2 pages such that:

- **Similar pages:** Each page $p \in C_1$ has a matching page $M(p) \in C_2$, such that $p \approx M(p)$.
- **Similar links:** For each page p in C_1, let $P_1(p)$ be the set of pages in C_1 that have a link to page p. Similarly define $P_2(M(p))$ for pages in C_2. Then we have pages $p_1 \in P_1(p)$ and $p_2 \in P_2(M(p))$ such that $p_1 \approx p_2$ (unless both $P_1(p)$ and $P_2(M(p))$ are empty). This definition means that two corresponding pages should have at least one parent (in their corresponding collections) that are also similar pages.

□

Conceptually, our definition matches two collections C_i and C_j if we can find at least one "tree" (out of the web graph) in C_i that is identical to a "tree" in C_j. As we mentioned

earlier, we experimented with other definitions as well (e.g., map pages only if they share at least two or more parents in common). However, we chose Definition 4.4.4 since it helps identify similar collections effectively and efficiently over large amounts of data, as we will see in Section 4.8.

4.5 Defining similar clusters

Definition 4.5.1 (**Cluster**) We define a set of equi-sized collections to be a *cluster*. The number of collections in the cluster is the *cluster cardinality*. If s is the collection size of each collection in the cluster, the *cluster size* is s as well. □

Definition 4.5.2 (**Identical cluster**) Cluster $R = \{C_1, C_2, \ldots, C_n\}$ is an *identical cluster* if for $\forall i, j, C_i \equiv C_j$. That is, all its collections are identical. □

Definition 4.5.3 (**Similar cluster**) For cluster $R = \{C_1, C_2, \ldots, C_n\}$, consider graph $G = (V, E)$, where each vertex v_i corresponds to collection C_i and edge $\langle v_i, v_j \rangle$ indicates C_i is similar to C_j. Cluster R is a similar cluster if G is connected. □

Notice that Definition 4.5.3 defines "chained" clusters as similar. Consider the case $C_i \cong C_j$, $C_j \cong C_k$, but C_i is not $\cong C_k$. However, we will empirically show in Section 4.7 that this definition is a good definition.

4.6 Computing similar clusters

In the previous section, we discussed how to check if two given clusters are similar. Now we consider what are "good" clusters to compute, when we are given a web graph. For instance, in Figures 4.3 and 4.4 we show two possible similar clusters we can compute given the three collections shown. In Figure 4.3 we show the case where the cluster cardinality and size are two and five (by excluding collection (iii)), while in Figure 4.4 the cardinality and size are three each. In general, we may choose similar clusters based on some objective function parameterized by both the cluster cardinality and size. Unfortunately, computing similar clusters for any arbitrary objective function may not always be feasible for the following reasons.

- One naive approach is to enumerate all possible subcollections and clusters and choose similar clusters based on the objective function. However, this approach does not scale

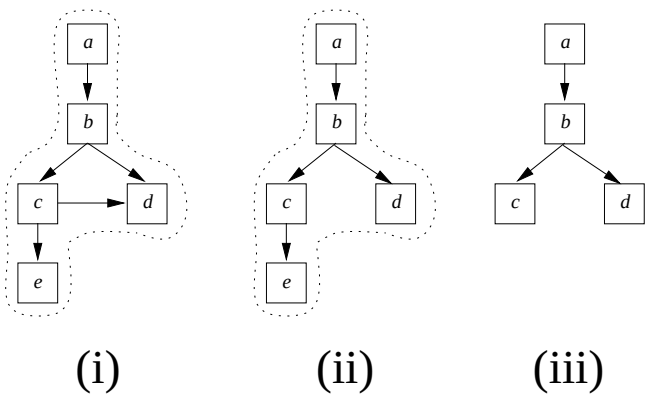


Figure 4.3: One possible similar cluster.

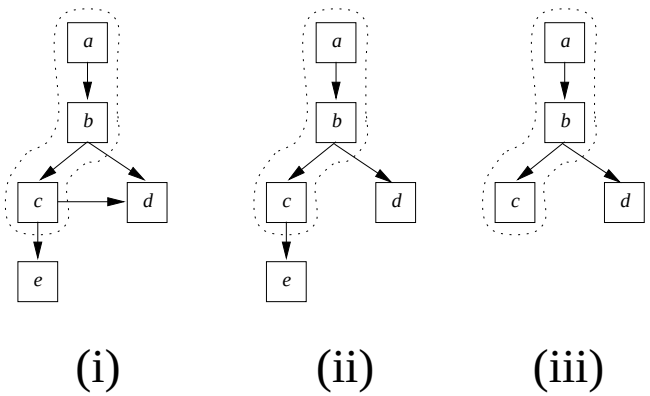


Figure 4.4: Another possible similar cluster.

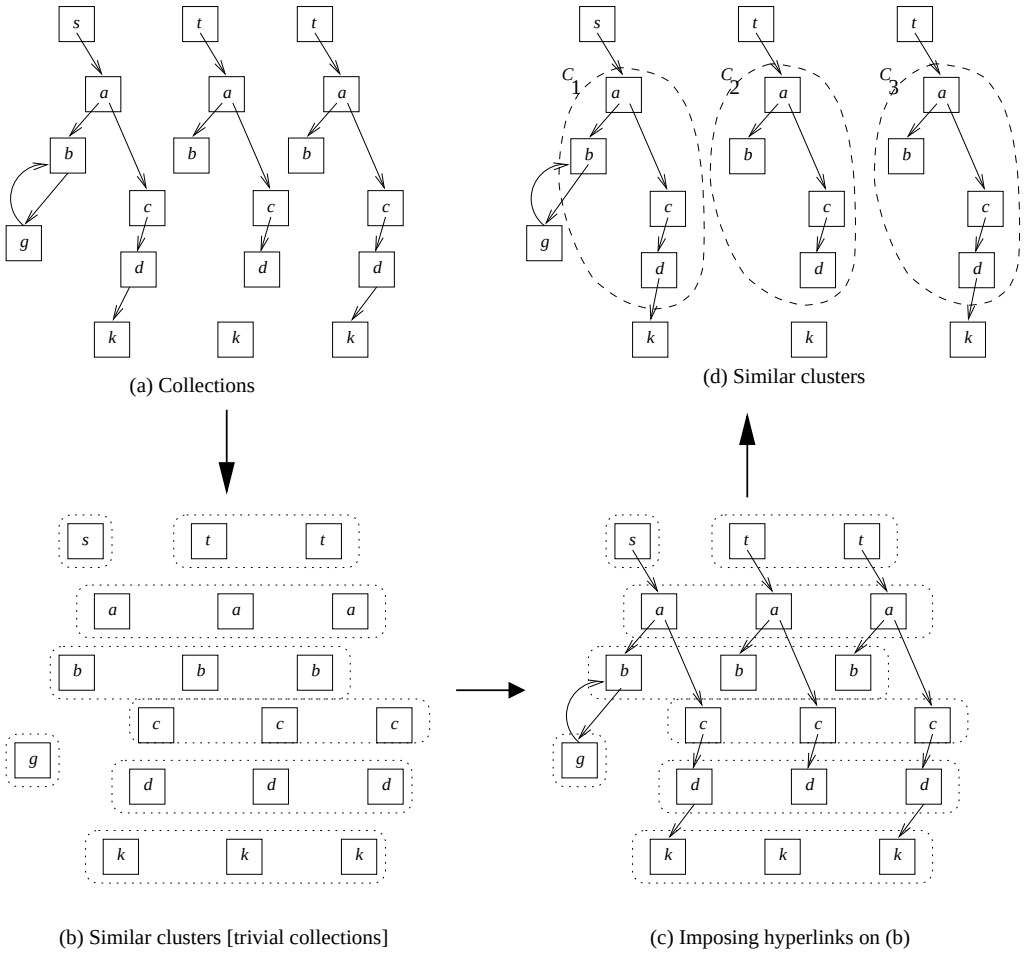


Figure 4.5: Computing similar clusters from a set of web pages.

because the number of subcollections is exponential in the number of pages in each web site.

- The amount of data we need to process is on the scale of hundreds of gigabytes of web pages. In this scenario, we cannot even afford to maintain something as simple as the link information between pages (i.e., page p_i points to page p_j) in main memory, since that requires tens of gigabytes in compressed, delta-encoded form. Hence while choosing an objective function, we need to be sensitive to such scalability requirements.
- Even if we chose a good objective function, the number of possible similar clusters may be too large.

In this context, we propose the following *cluster growing* algorithm (illustrated in Figure 4.5) to produce *maximal cardinality* clusters efficiently on large amounts of web data. As we will see later, this algorithm does indeed produce "good" clusters. In the next section, we discuss how to implement this algorithm efficiently.

1. **Compute trivial similar clusters:** First, we ignore hyperlinks between pages and compute similar clusters of trivial collections (cluster size equal to one). We illustrate this step in the transition from Figure 4.5 (a) to (b).
2. **Grow trivial clusters into larger similar clusters:** We impose the hyperlink structures on the trivial similar clusters we computed in the previous step, and grow these clusters into larger similar clusters as we discuss below.

4.6.1 Growth strategy

We now discuss how to grow clusters abstractly using standard graph theoretic terminology, to give the reader the motivation behind our approach. In the next section, we will discuss the implementation details behind our growth strategy.

Consider two trivial similar clusters $R_i = \{p_1, p_2, \ldots, p_n\}$ and $R_j = \{q_1, q_2, \ldots, q_m\}$, where p_k and q_k $(1 \leq k \leq n)$ are pages in R_i and R_j. We define (1) $s_{i,j}$ to be the number of pages in R_i with links **to** at least one page in R_j, (2) $d_{i,j}$ to be the number of pages in R_j with links **from** at least one page in R_i, (3) $|R_i|$ to be the number of pages in R_i, and (4) $|R_j|$ to be the number of pages in R_j.

Now consider the set of all trivial clusters $R = \cup R_i$. Consider graph $G = (V, E)$, where each vertex v_i in V corresponds to trivial cluster R_i. We have an edge from v_i to v_j if

each page in R_i has a hyperlink to a unique page in R_j. That is, $|R_i| = s_{i,j} = d_{i,j} = |R_j|$. We call this condition, the *merging* condition. Conceptually this edge indicates that all the pages in R_i have hyperlinks to pages in R_j, and hence are part of a larger collection. We illustrate the above condition using the following example. In Figure 4.6 (a), we see the trivial clusters marked in dotted boxes. In Figure 4.6 (b), we show graph G. There are hyperlinks from the trivial cluster for a to the clusters for b and c, and from c to d since these clusters satisfy the above merging condition. We then grow clusters based on the merging condition. That is, we compute connected components in G, and each connected component in G yields a similar cluster as illustrated in Figure 4.6 (c). The clusters so produced are *maximal* cardinality clusters – these clusters cannot be grown further (according to the merging condition) to produce larger cardinality clusters.

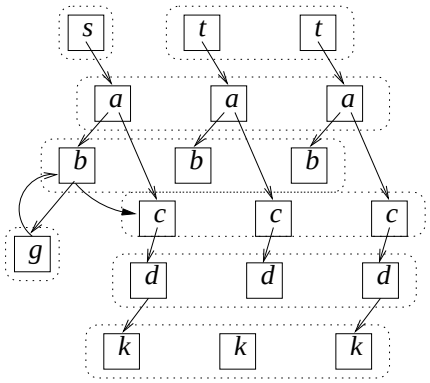
Though the above procedure computes maximal cardinality clusters automatically, the procedure merely identifies which pages belong to each cluster. While this output suffices for some applications (e.g., improving search results), for other applications it may be useful to also identify the "roots" of the collections. However identifying the root page of a collection is often tricky. One simple heuristic is to denote the page with no incoming hyperlinks as the root, if some such page exists. But in some collections, many pages in the collection in fact do have hyperlinks to the root page, as a convenient "short-cut" for web surfers. Another promising heuristic is to denote the page with the largest number of incoming hyperlinks from other web sites to be the root page [BP99]. For the applications we consider in this dissertation, this additional step of identifying the collection roots is not important. We believe that a useful direction for future research is a careful evaluation of algorithms to compute collection roots.

4.6.2 Implementing the cluster growing algorithm

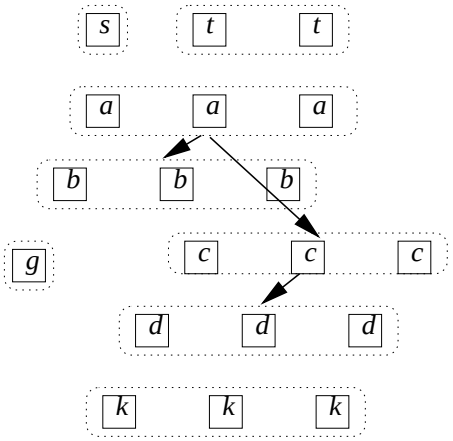
We now discuss how to implement the cluster growing algorithm we described in the previous section.

Computing trivial clusters

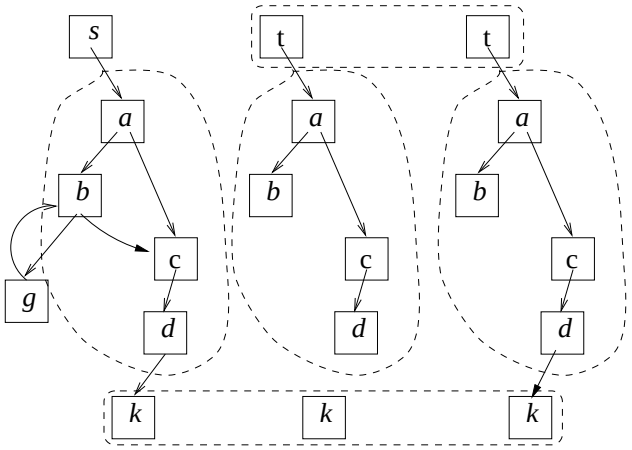
We compute all similar page pairs using the techniques we introduced in Chapter 3, and store these pairs into a table. We can then perform a simple disk-based UNION-FIND [CLR91] algorithm that performs sequential scans over this table and computes trivial similar clusters. The output of this phase is a table TRIVIAL(rid, pid), which contains tuple $\langle R_i, p_j \rangle$ to



(a) Trivial clusters



(b) Growing clusters



(c) Maximal cardinality clusters

Figure 4.6: Growing similar clusters.

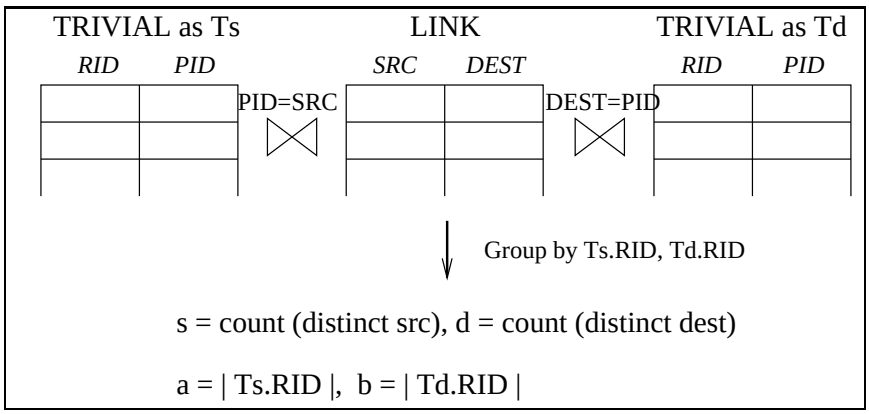


Figure 4.7: Join-based construction of *LinkSummary* table

indicate that p_j is in trivial cluster R_i. Note that this table materializes all trivial similar clusters as per Definition 4.5.3.

Growing trivial clusters into larger clusters

Figure 4.8 shows how to implement the cluster growing algorithm efficiently. It expects as input the following two tables: (1) TRIVIAL(rid, pid) from Section 4.6.2 and (2) LINK$(src, dest)$, a table with tuple $\langle p_i, p_j \rangle$ if p_i has a hyperlink to p_j. Conceptually, in Step [1] we are preprocessing the data to compute link statistics ($s_{i,j}, d_{i,j}, |R_i|$, and $|R_j|$) for every pair of trivial clusters, R_i and R_j. In Steps [2] – [4] we are computing all pairs of trivial clusters that satisfy the merging condition we discussed in the previous section. Finally, we compute our maximal cardinality clusters in Step [5] using the classical UNION-FIND algorithm [CLR91]. In this step, we are conceptually regaining the collection structure (i.e., Figure 4.6 (c)) by merging collections based on computing connected components.

4.7 Quality of similarity measures

In this chapter, we defined similarity between pairs of pages, collections and clusters. While defining each of these concepts, we made certain choices because there were several different ways in which we could define these concepts. Based on these measures, we then presented the cluster growing algorithm to compute a specific kind of similar clusters from a given set of web pages. In this section, we discuss how good are the choices we made as follows. For a dataset of web pages (described in detail below), we compute similar clusters using the

Algorithm 4.6.1 *Cluster growing algorithm*
Inp : Tables *Trivial* and *Link*,
Out : The set of similar clusters
Procedure
[0] **S** $\leftarrow$ {} `// S: the set of similar clusters`
[1] Construct *LinkSummary* table with schema
$\langle\ Ts.RID,\ Td.RID,\ a,\ b,\ s,\ d\ \rangle$ based on Figure 4.7
[2] For each entry in *LinkSummary*
[3] If $(a = s = d = b)$
[4] **S** $\leftarrow$ **S** $\cup \{\langle Ts.RID,\ Td.RID\rangle\}$ `// Coalesce` $Ts.RID$ `and` $Td.RID$
[5] Return "UNION-FIND(**S**)" `// Find connected components`

Figure 4.8: Cluster growing algorithm.

cluster growing algorithm. Then we examine the clusters so computed and check if they are "good" clusters.

We chose 25 widely replicated web collections (e.g., Perl, XML, JAVA manuals) from their primary sites (e.g., www.perl.org). The sizes of these collections varied between 50 and 1000 pages. Then for each such collection, we automatically downloaded between five and ten mirrored versions of these collections from the web. The mirrored collections were different versions of the primary version, since they were either older versions of the primary version or had minor modifications (e.g., additional links or inlined images). The total number of web pages so downloaded was approximately $35,000$. In addition, we added $15,000$ randomly crawled pages to this data set. We assume these pages were unrelated to each other and the collections we had already downloaded.

We then computed similar clusters using our cluster growing algorithm on the above data set. We then manually examined these clusters to see if they corresponded to the mirrored collections. In general, we noticed that the clusters produced by our algorithm were of high quality. That is, the clusters had the "core" of the mirrored collections. However, we also noticed that often our techniques split up a cluster of mirrored collections into several smaller similar clusters. For instance, the five XML collections in our data set were split up into seven clusters each with four to five sub-collections each. We examined such clusters further and observed the following to be the reasons for such splits.

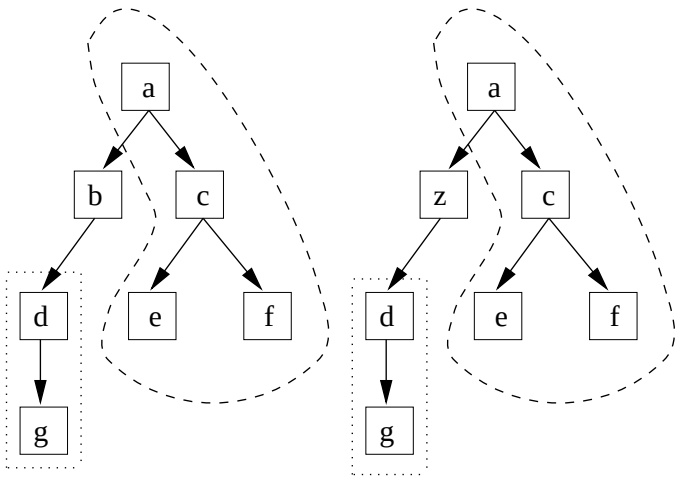


Figure 4.9: Example of "break point."

1. **"Break points:"** In Figure 4.9 we show two versions of the same collection. Notice that the two collections are identical except for $[i].b$ and $[ii].z$. In such a case, the cluster growing algorithm produces two clusters with two sub-collections each, as indicated by the dotted and dashed lines. This scenario occurs when a page (the "break point") is radically modified from one version to another, and splits up a cluster into several smaller clusters (as in Figure 4.2 (c)). In our data set, this scenario was relatively infrequent and hence we do not discuss this further.

2. **Partial mirrors:** In Figure 4.10 we show a collection and its partial mirror. Here the partial mirror (collection $[ii]$) is a sub-collection of collection $[i]$. The pages in this sub-collection point to the unmirrored portions of the first collection. In our data set, we observed that this form of partial mirroring was quite common. Notice that our cluster growing algorithm identifies $[i].\{a, b, c\}$ and $[ii].\{a, b, c\}$ to be a maximal cluster. In some applications (e.g., in avoiding redundant crawling), these clusters are natural. However in other applications (e.g., in improving web searching), we may wish to identify collection $[ii]$ to be identical to collection $[i]$, due to the partial mirroring. For such cases, we modify the merging condition in our cluster growing algorithm as follows. We merge trivial clusters R_i and R_j whenever $|R_i| = s_{i,j} = d_{i,j} \geq |R_j|$. Notice that this condition arises when pages in R_j have "virtual" links from pages in R_i, i.e., the R_j pages are not mirrored in as many sites as the R_i pages. We call

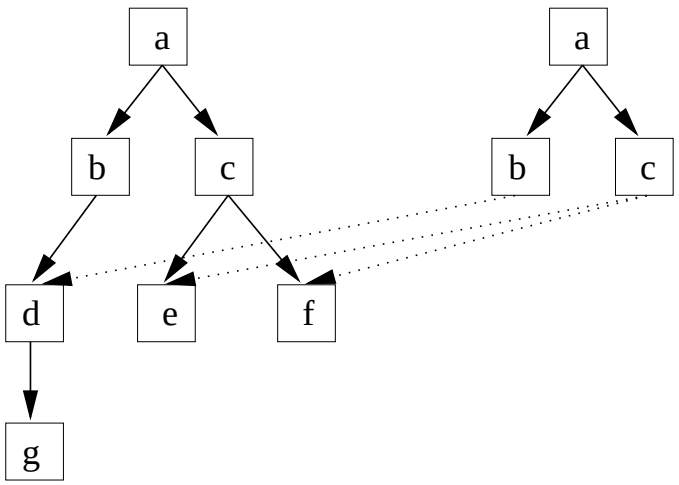


Figure 4.10: Example of partial mirrors.

the clusters produced by our modified merging condition as *maximal clusters*. In our experiments, we observed that such clusters corresponded to natural collections (e.g., JAVA manuals). In the next section, we visualize maximal clusters on a much larger data set to show the quality of our clusters.

4.8 Performance experiments

We performed experiments to evaluate the following: (1) how our algorithms scaled for hundreds of gigabytes of data, and (2) how we can use our techniques in this dissertation to improve the performance of two applications we discussed in Section 4.1. For our experiments, we again used data crawled by the Stanford Google web crawler [BP99]. We ran our experiments on a SUN UltraSPARC with dual processors, 256 MBs of RAM and 1.4 GBs of swap space, running SunOS 5.5.1.

4.8.1 Resources wasted while crawling

For the first application, we wanted to perform a comprehensive performance analysis of web crawlers to identify how much redundant data they crawl. That is, we wanted to quantify the resources (such as network bandwidth) crawlers waste in visiting multiple copies of pages and collections. We believe such a performance analysis of web crawlers

	Measures / Signatures	Entire document	Four lines	Two lines
Space	Fingerprints	800 MBs	2.4 GBs	4.6 GBs
Time	Compute fingerprints	44 hrs	44 hrs	44 hrs
	Compute trivial similar clusters	97 mins	302 mins	630 mins

Table 4.1: Storage and time costs for computing trivial similar clusters.

is crucial because web crawlers are complex pieces of software [BP98]: they are usually "hacked up" and are not the result of rigorous research efforts. Indeed, we will see in this section that our performance analysis helped us improve a fully-operational crawler.

Initially, we chose to "debug" the Google web crawler. We chose this particular crawler since it was developed within our group and we had direct access to its data. In any case, the main goal of our experiments is to test the scalability of our algorithms and the quality of our definitions. Hence we believe our choice in using data crawled by Google does not affect our performance results. Also in some recent work, other researchers report results similar to our own in the context of the AltaVista web crawler [BGM97, BB99].

The Google crawler fed us approximately 25 million web pages crawled primarily from domains located in the United States of America. This dataset corresponds to about 150 gigabytes of textual information. Rather than use 5-grams as our similarity measure, we used the following three chunkings to reduce the storage requirements of our signatures: (1) one fingerprint for the entire document, (2) one fingerprint for every four lines of text, and (3) one fingerprint for every two lines of text. We computed the corresponding *DocChunk* with these fingerprints, along with a 32-bit integer to identify each unique URL. To give the reader a flavor for the resources utilized, we report in Table 4.1 the costs in storing the signatures, as well as the time breakdown in computing the set of trivial similar clusters in Table 4.1, with thresholds $T = 15$ and $T = 25$ for the "four line" and "two line" fingerprints respectively.

In Figure 4.11 we report the number of similar pages for each the three chunkings. For instance, let us consider the case when we compute one fingerprint on the entire document. About 64% of the 25 million web pages has no replicas (the left-most darkest bar) except itself: about 18% of pages have an additional exact copy – that is, there are about $\frac{1}{2} * \frac{18}{100} * 25 * 10^6$ distinct pages that have one exact copy among the other $\frac{1}{2} * \frac{18}{100} * 25 * 10^6$ pages in this category. Similarly, about 5% of pages have between 10 and 100 replicas. As expected, the percentage of pages with more than one similar page increases when we relax

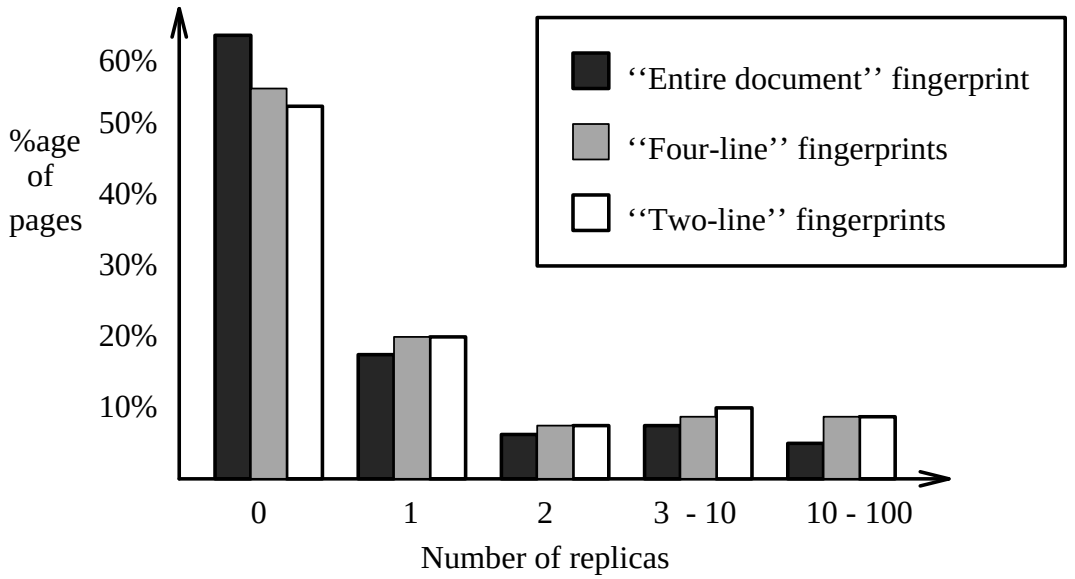


Figure 4.11: Document replication on 25 million web pages.

the notion of similarity from 36% (100 – 64%) for exact replicas, to about 48% (100 – 52%) for "two-line" chunks.

From the above experiments, it is clear that the Google crawler wastes significant resources crawling multiple copies of pages. Since our goal was to improve the Google crawler, we had to identify why it was wasting so much resources. For this, we computed maximal clusters for the trivial clusters as computed by the "four-line" fingerprints. The overall process of computing maximal clusters from trivial similar clusters took about twenty hours.

In Figure 4.12, we visualize each maximal cluster as a point in the graph, based on the cluster cardinality and size. For the reader's convenience, we annotate some of the points that correspond to document collections (we manually labeled these points after automatically computing maximal clusters). For instance, the LDP point (near the top right corner) indicates that the LDP collection constitutes 1349 pages and has 143 replicas in our crawled data set. We examined several maximal clusters in Figure 4.12 to understand why there were so many replicas and also to evaluate the quality of our similarity definitions. Based on this examination, we roughly partitioned the clusters into the following categories:

1. **Widely replicated collections:** We found several maximal clusters that corresponded to widely replicated collections. We list the five clusters sets with the largest

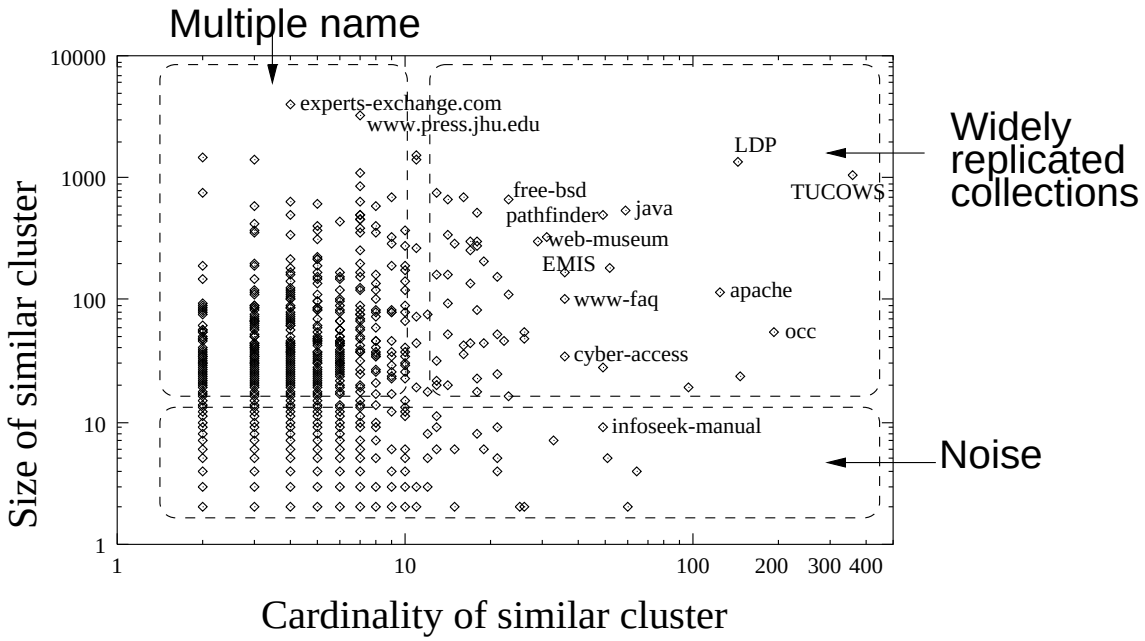


Figure 4.12: Distribution of maximal clusters.

Rank	Description	Similar cluster cardinality	Similar cluster size
1	TUCOWS WinSock utilities `http://www.tucows.com`	360	1052
2	LDP Linux Documentation Project `http://sunsite.unc.edu/LDP`	143	1359
3	Apache Web server `http://www.apache.org`	125	115
4	JAVA 1.0.2 API `http://java.sun.com`	59	552
5	Mars Pathfinder `http://mars.jpl.nasa.gov`	49	498

Table 4.2: Popular web collections.

cardinalities that correspond to real collections in Table 4.2. For instance, the TUCOWS WinSock utilities is replicated at 360 web sites across the world, constitutes about 1052 web pages and the principal copy is located at the listed URL.

2. **Multiple name servers:** In many cases, a set of machines in a single domain share content even if the machines have different IP addresses. Web sites such as `www.experts-exchange.com` fall into this category.

3. **Noise:** In our data set, we observed that we had a few false positives primarily because a set of pages were similar in format but had different content. For instance, may web sites have HTML versions of PowerPoint slides created by the "Save As HTML" feature in Microsoft PowerPoint. Readers may be aware that PowerPoint saves each slide as an image file (such as in .GIF format) and creates a slide-show of HTML pages. Each such HTML page has the same content and hyperlink structure but points to the inlined GIF rendering of the corresponding slide. Since our system computes page similarity based on textual content and not based on image similarity, PowerPoint slide-shows are placed into the same maximal cluster. Also in some cases, we had small clusters induced by "break point" pages.

We examined several of the maximal clusters and roughly classified these clusters based purely on cluster cardinality and size. We observed that maximal clusters with size less than ten pages, often were not "good" clusters according to a human (i.e., false positives). Most

maximal clusters with size greater than ten and cardinality less than ten, were typically a set of machines with multiple names and IP addresses in the same domain offering the same content. Finally, maximal clusters with size and cardinality greater than ten were indeed widely replicated collections.

Based on the above rough classification, we precoded information about the widely replicated collections and machines with multiple names, into the Google web crawler. We then ran the crawler again and the crawler avoided the precoded collections and machines. This time, 35 million web pages corresponding to about 250 gigabytes of textual data were crawled. We again computed the set of similar pages as in the earlier experiments and observed that the number of similar pages had dropped from a staggering 48% in the previous crawl to a more reasonable 13%. Similarly, the number of identical copies also fell to about 8%.

Note that the number of similar and exact pages did not drop to zero in the new crawl. This is because we crawled many more pages in the new crawl, and we identified additional maximal clusters. For the next crawl, we will update the information we precoded into the crawler to avoid visiting these pages as well. From initial observations, however we also believe that the number of collections we need to add to the crawler's "avoid" list will drop with time since many of these replicated collections will be identified after the first few crawls. In addition, we will need to keep updating the precoded information to remove sites and collections that are no longer replicas. In the future, we plan to derive an "update model" for pages and collections as well so that we avoid visiting only similar collections.

4.8.2 Improving search engine result presentation

For some applications, it makes sense for the crawler to continue crawling multiple copies of a collection for a variety of reasons (e.g., reliability, data mining, or to avoid "down times" at remote sites). Even in this case, these applications may choose to remove redundant copies of collections after crawling. For instance, consider how a web search engine operates today. When we search for concepts such as "object-oriented programming" or "Linux" on the web, search engines return a list of pages ranked using some proprietary ranking function. However, the displays are often distracting due to the following:

1. Multiple pages within a hyperlinked collection are displayed. For instance, in Figure 4.13 we show an example of how the Google search engine displays results for a search for "XML databases." We see that several pages in `www.techweb.com` satisfy

this query and links to all these pages are displayed.

2. Links to replicated collections are displayed several times. For instance, in Figure 4.13 we show the case when the TechWeb collection that is available at `www.techweb.com` and `techweb.cmp.com` (among other sites), is displayed multiple times.

We used the algorithms we discussed earlier in this chapter and computed similar clusters. We then implemented a prototype web search engine interface on top of the Google search engine to address the above problems. In Figure 4.14 we show how our prototype displays the results for the user query "XML databases." As we can see, the prototype "rolls up" collections so that it displays only the link of one page in a collection even if multiple pages within the collection may satisfy the query. Also the prototype automatically "rolls up" replicas of collections as well. Notice that in our prototype we also have two additional links marked **Collection** and **Replica** for each result. When a user clicks on the **Collection** link, the collection is rolled down and all pages satisfying the user query are displayed. Similarly, when the user clicks on the **Replica** link, the prototype displays all the replicas of the collection. We believe such an interface is useful to a user, since the interface gives the user a "high-level" overview of the results rather than just a long list of URLs.

We evaluated the above search interface informally within our group. In general, users have found this prototype to be valuable especially for queries in technical areas. For instance, when our users tried queries in topics such as Latex, C++, Latex, JAVA, XML and GNU software, search engines typically yield too many redundant results. The same queries yielded much better results in our prototype due to the clustering. Hence we believe that by computing similar clusters and rolling up collections and clusters, we can improve current search engines significantly.

4.9 Conclusion

In this chapter, we considered additional applications of the techniques we introduced earlier for building a text CDS. These applications required us to extend our definitions of page similarity to that of collection similarity on the web. We then discussed how to efficiently identify such pages and collections from several tens of millions web pages and several hundreds of gigabytes of textual data. We also presented two real-life case studies where

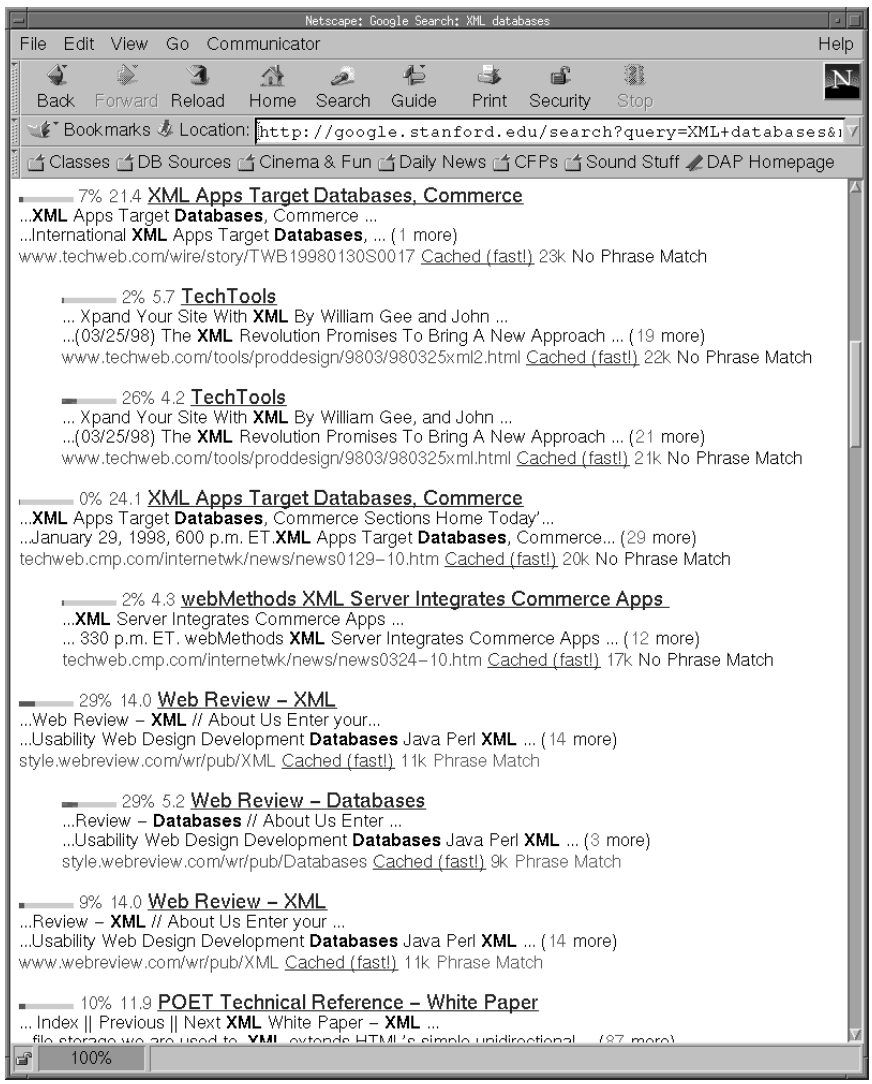


Figure 4.13: Search results for "XML databases" from Google search engine.

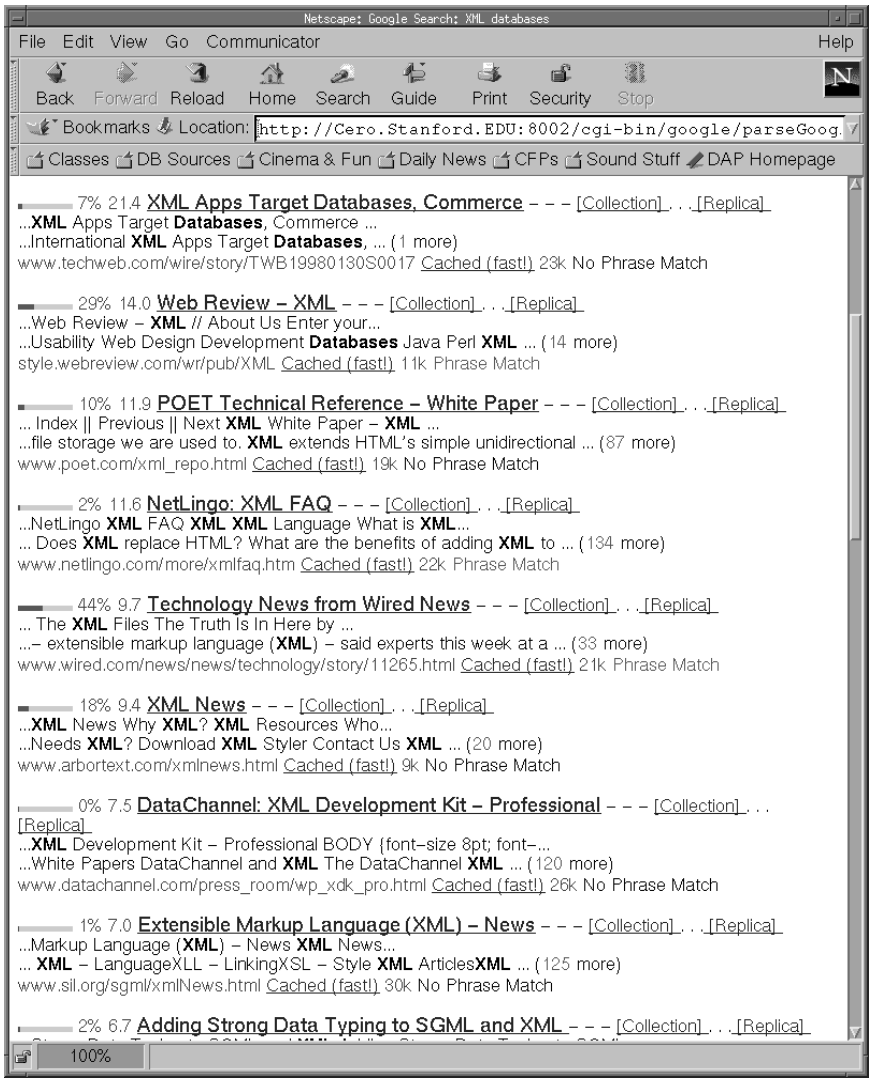


Figure 4.14: Rolled up search results for "XML databases".

we used the information we compute to improve (1) the performance of a web crawler by up to 40%, and (2) how search engines present results.

For this chapter, we made many choices when designing similarity measures. As discussed earlier, the main challenge was to design these measures in such a way that they were "good" measures and were also efficient to compute over hundreds of gigabytes of data. While we tested different options for these measures, often we were constrained by our own disk and CPU resources. For instance, we could not evaluate some similarity measures because the corresponding algorithms were quadratic in nature (or worse), which was unacceptable given the tens of millions of pages we dealt with. Similarly, other similarity measures would have required us to perform random seeks on disk, again unacceptable due to the several hundreds of gigabytes of data we need to process. Recall that even computing similar clusters for the 4-line fingerprints using our sub-quadratic algorithms that process our data with no random seeks required three days cumulatively on a dedicated machine. Hence while the similarity measures for collections are a ripe area for further future work, we believe the similarity measures we introduced in this chapter are good in terms of both efficiency and quality for at least the two applications we discussed.

I hope you get a thesis chapter out of this [plagiarism case].

– Dr. Maurice Herlihy, May 1995

Chapter 5

Discovery-based Plagiarism Detection

5.1 Introduction

In an infamous plagiarism case in 1995, a computer science graduate student researcher (to whom we will refer as Mr. X) plagiarized several technical reports and conference papers written by other researchers. More than fifteen of these papers passed undetected through the paper review process and were accepted to several conferences and journals. The topics of these papers ranged from Steiner routing in VLSI CAD, to massively parallel genetic algorithms, complexity theory, and network protocols. Papers that were plagiarized from the database field include a paper from the Journal of Distributed and Parallel Databases (DAPD) by Tal and Alonso [TA94] on three-phase locking, a paper from the International Conference on Very Large Databases (VLDB '92) by Ioannidis et al. [INSS92] on parametric query optimization, and a paper from the International Conference on Data Engineering (ICDE '90) by Leung and Muntz [CTL90] on temporal query processing. Our SCAM text CDS prototype played a key role in identifying the papers that Mr. X had plagiarized as we discuss in this chapter [Den95, Ros96, Goo97]. This case motivated the need for automated "discovery" mechanisms for potential copies of documents, the main focus of this chapter.

5.1.1 History of plagiarism case

In May 1995, Dr. Maurice Herlihy from Brown University reviewed a paper on three-phase locking that he realized was a plagiarized version of an earlier paper by Alonso

and Tal [TA94]. He posted the paper's abstract to DBWORLD (a world-wide mailing list of database researchers) as a "contest" soliciting guesses on where this paper was plagiarized from. Dr. Herlihy also noted that Mr. X had a few other papers according to INSPEC (a commercial database of abstracts of papers in Computer Science and Electrical Engineering) and wondered if these other papers were also plagiarized. Since this was a good case to test out SCAM (an initial version was operational by then), we proceeded as follows:

1. From INSPEC, we downloaded the abstracts of eight other papers written by Mr. X, i.e., the suspicious documents.

2. We then selected databases that were likely to contain the original papers. Based on the content of the suspicious documents we decided that the INSPEC and CS-TR (an emerging digital library of computer science technical reports) databases were the most appropriate.

3. We manually chose some keywords from Mr. X's abstracts, and issued the disjunct of these words as a query to the above databases. For instance, for the sample abstract in Figure 5.1, we issued the boolean query "wirelength or Steiner or Routing" against the above databases.

4. We retrieved all abstracts (about 35,000 overall) that matched the above queries, registered them in SCAM, tested the suspicious documents against them and identified the original papers. For example, SCAM used its similarity measures and identified the abstract in Figure 5.2 as the closest match. Finally, we notified the authors of the original papers.

While notifying one of the authors, we realized that Dr. Costas Halatsis from the Programming Languages community was heading a similar effort. Mr. X had submitted a plagiarized paper to the EuroPAR'95 conference, and Dr. Halatsis in his role as Program Committee Chair was heading an inquiry into Mr. X. At this point, Dr. Halatsis and we joined forces to crack this plagiarism case. Dr. Halatsis then obtained more abstracts "written" by Mr. X from Mr. X's supervisor. Given these additional abstracts, we repeated the above process using SCAM and identified more original papers that were plagiarized.

By June, we had 30 abstracts written by Mr. X and SCAM identified 15 of these to be abstracts of previously published papers. While we believe many of the other abstracts were also plagiarized, we could not identify the original versions of the other papers since INSPEC

Title: Provably optimal algorithms for signal routing.
Author: Mr. X
Published in: *Computer Systems Science and Engineering*, vol.9, no.4, p. 211-19, 0267-6192 Oct. 1994.

In this paper, we propose a provably good performance-driven global routing algorithm for both cell-based and building block design. The algorithm simultaneously minimizes both routing cost and the longest interconnection path, so that both are bounded by small constant factors away from optimal. Our method is based on the following two results. First, for any given value of a parameter epsilon , we can construct a routing tree with a cost of at most (1+(1/ epsilon)) times the minimum spanning tree weight, and with a longest interconnection path length of at most (1+(1/ epsilon)).R. Moreover for Steiner global routing in arbitrary weighted graphs, we achieve a wiring cost within a factor of the optimal Steiner tree cost, again with a longest path length of at most (1+(1/ epsilon)).R. In both cases, R is the minimum possible length from the source to the furthest sink. We also show that geometry helps in routing: in the Manhattan plane, the total wirelength for Steiner routing improves to 3/2.(1+(1/ epsilon)) times the optimal Steiner tree cost, while in the Euclidean plane, the total cost is further reduced.

Figure 5.1: Example abstract "written" by Mr. X.

only contains abstracts of published papers, and not of technical reports written in Universities and research labs. Based on our work, the Association of Computing Machinery (ACM) filed a case against Mr. X for violating ACM's copyright on papers he plagiarized [Den95].

5.1.2 Automatic plagiarism detection

Based on the motivation from the above plagiarism case, we develop PROBE-CDS, a system that automates the above process. In particular,

- *Discovery:* Given many databases (e.g., web sites or Dialog's databases) and a suspicious document, PROBE-CDS efficiently identifies the databases that may contain documents that a CDS would consider copies. PROBE-CDS should flag a database even if it contains a single document that overlaps significantly with the suspicious document.

- *Extraction:* PROBE-CDS automatically generates a query that retrieves potential copies from the underlying databases. The CDS will then subsequently analyze these retrieved documents to check if they are copies.

Title: Provably good algorithms for performance-driven global routing.
Authors: Cong, J.; Kahng, A.; Robins, G.; Sarrafzadeh, M.; Wong, C.K.; Dept. of Comput. Sci., UCLA and UCSD.
Published in: 1992 *IEEE International Symposium on Circuits and Systems* (Cat. No.92CH3139-3), p. 2240-3 vol.5 New York, NY, USA IEEE 1992.

The authors propose a provably good performance-driven global routing algorithm for both cell-based and building block design. The approach is based on a new bounded-radius minimum routing free formulation, which simultaneously minimizes both routing cost and the longest interconnection path, so that both are bounded by small constant factors away from optimal. The authors' method generalizes to Steiner global routing in arbitrary weighted graphs, and to the case where varying wirelength bounds are prescribed for different source-sink paths. Extensive simulations confirmed that the approach gave very good performance, and exhibited a smooth tradeoff between the competing requirements of minimum delay and minimum total wirelength.

Figure 5.2: Closest matching abstract from INSPEC according to SCAM.

Automating the above tasks is crucial especially as the number of document databases grows and as publishers rely more on digital publishing. However it is difficult to automate the discovery and extraction processes, so that these processes are also efficient. For instance, for the extraction problem, a naive solution is to simply query a database for all documents containing any of the terms in the suspicious (query) document. For instance, if the suspicious document contains words $w_1, w_2, \ldots, w_n$, the PROBE-CDS extractor can issue the boolean query $w_1 \vee w_2 \vee \ldots \vee w_n$ (or alternatively, request all documents with w_1, then all the ones with w_2, and so on) to the database. Clearly, any potential copy would be extracted in this way. However this approach may be expensive since it may retrieve too many documents from the underlying database. In this chapter, we discuss one strategy that relies on collecting some advance "metainformation" from the underlying databases such that the discovery and extraction algorithms are more efficient.

5.1.3 Summary of our methodology

For the discovery phase, we build on previous work in the resource-discovery area, more specifically, on the *GlOSS* approach [GGMT94, GGM95]. The idea is to collect in advance "metainformation" about the candidate databases. This metainformation can include, for example, information on how frequently terms appear in documents at a particular database.

This information is much smaller than a full index of the database a CDS would need to detect copies. Then, based on this information, PROBE-CDS can rule out databases that do not contain documents that the text CDS would consider copies.

Notice that while PROBE-CDS is structurally similar to *GlOSS*, there is a fundamental difference. *GlOSS* attempts to discover databases that satisfy a given query (e.g., Boolean or vector space queries). The *GlOSS* problem is a simpler one since all we need to know is that the candidate database contains the necessary terms. However, for PROBE-CDS, finding documents that have similar terms to those of the suspicious document is not enough. For example, if the suspicious document only contains a subset that is a copy, then there are terms in the non-copy portion that are not relevant. Thus, simply treating the suspicious document as a *GlOSS* query will not lead us to the right databases.

Instead, PROBE-CDS will need to keep more sophisticated statistics than *GlOSS* does, enough to let it identify sites that may have even a single copy, not just documents that are "similar" to the suspicious one in an information retrieval sense. The key challenge is to collect as little information as possible in PROBE-CDS to be able to perform this difficult discovery task. Section 5.7 reports experiments that indeed show that our techniques are successful at isolating the databases with potential copies, with relatively few false positives.

In this chapter we discuss two types of discovery techniques: *conservative* and *liberal* ones. Conservative techniques only rule out a database if it is certain that the CDS would not consider *any* document there a potential copy. The clear advantage of conservative techniques is that they do not miss potential copies. In contrast, liberal techniques might in principle miss databases with potential copies. However, this is rarely the case, as we will see, and the liberal techniques search fewer unnecessary databases. In practice, the choice between conservative and liberal depends on how exhaustive the search must be and what resources are available.

We solve the extraction problem by showing how to find the "minimal query," under two different cost metrics, that can extract all the desired documents. For example, one of our cost measures is the number of words in the submitted query. We will see that we can reduce such a number drastically by bounding the maximum "contribution" of every word to a potential copy.

In this chapter, we assume that the underlying databases support a simple Boolean search interface [Sal88]. Recall from Chapter 2 that IR databases use word-based measures such as the Cosine Similarity Measure (CSM) [Sal88] for ranking documents, rather than

chunking based measures (Chapter 2). That is, Boolean databases in general do not support chunkings. Also recall from Chapter 2 that the RFM measure performs better than the CSM measure. Therefore in this chapter, we use the (RFM) measure for building our PROBE-CDS.

In Section 5.2 we reintroduce the RFM definition. In Section 5.3 we describe the data that PROBE-CDS keeps about the databases. We use this data in Section 5.4 to define the conservative copy discovery schemes for PROBE-CDS, while in Section 5.5 we relax these schemes to make them liberal. In Section 5.6 we present the extraction mechanisms. Finally, in Section 5.7 we discuss an experimental evaluation of our techniques, using a collection of 50 databases and two sets of suspicious documents.

5.2 RFM measure

Given a suspicious document S and a registered document D, recall that the RFM definition denotes D to be a potential copy of S if they overlap significantly. We reintroduce our definition using some terminology that will help motivate our PROBE-CDS strategies. The RFM measure first focuses on the words that appear a similar number of times in S and D, and ignores the rest of the words. More precisely, given a fixed $\epsilon > 2$, the *closeness set* for S and D, $c(S, D)$, contains the words w_i with a similar number of occurrences in the two documents:

$$w_i \in c(S, D) \Leftrightarrow \frac{F_i(S)}{F_i(D)} + \frac{F_i(D)}{F_i(S)} < \epsilon$$

where $F_i(d)$ is the frequency of word w_i in document d. If either $F_i(S)$ or $F_i(D)$ are zero, then w_i is not in the closeness set. Given ϵ, S determines a range of frequencies $Accept(w_i, F_i(S))$ such that w_i is in the closeness set for S and D if and only if $F_i(D) \in Accept(w_i, F_i(S))$. We then have

$$sim(S, D) = \max\{subset(S, D), subset(D, S)\}$$

where:

$$subset(D_1, D_2) = \sum_{w_i \in c(D_1, D_2)} \frac{F_i(D_1)}{|D_1|} \cdot F_i(D_2)$$

($|D| = \sum_{i=1}^{N} F_i^2(D)$ is the norm of document D and N is the number of terms.) If $sim(S, D) > T$, for some user-specified threshold T, then D is a potential copy of the suspicious document S.

EXAMPLE 5.2.1 Consider a suspicious document S and a database *db* with two documents, D_1 and D_2. There are four words in these documents, w_1, w_2, w_3, and w_4. The following table shows the frequency of the words in the documents.

Document	F_1	F_2	F_3	F_4
S	1	3	3	9
D_1	1	3	0	0
D_2	0	8	5	0

For example, w_3 appears three times in S ($F_3(S) = 3$), five times in D_2 ($F_3(D_2) = 5$), and it does not appear in D_1 ($F_3(D_1) = 0$). For $\epsilon = 2.5$ (Chapter 2), $Accept(w_3, F_3(S)) = Accept(w_3, 3) = [2, 5]$. Thus, w_3 is in $c(S, D_2)$, the closeness set for S and D_2, because $F_3(D_2) = 5$ is in $Accept(w_3, F_3(S))$. Although $F_3(D_2)$ is higher than $F_3(S)$, these two values are sufficiently close for $\epsilon = 2.5$. In effect, $\frac{F_3(S)}{F_3(D_2)} + \frac{F_3(D_2)}{F_3(S)} = \frac{3}{5} + \frac{5}{3} = 2.27 < \epsilon = 2.5$. For the remaining cases, $Accept(w_1, F_1(S)) = [1, 1]$, $Accept(w_2, F_2(S)) = [2, 5]$, and $Accept(w_4, F_4(S)) = [5, 17]$. Then, $c(S, D_1) = \{w_1, w_2\}$, and $c(S, D_2) = \{w_3\}$.

We then have $|S| = F_1^2(S) + F_2^2(S) + F_3^2(S) + F_4^2(S) = 1^2 + 3^2 + 3^2 + 9^2 = 100$. Similarly, $|D_1| = 10$ and $|D_2| = 89$. To compute the similarity $sim(S, D_2)$ we just consider w_3, the only word in the closeness set for S and D_2. Then,

$$\begin{aligned} sim(S, D_2) &= \max\{F_3(D_2) \cdot \frac{F_3(S)}{|S|}, \frac{F_3(D_2)}{|D_2|} \cdot F_3(S)\} \\ &= \max\{5 \cdot \frac{3}{100}, \frac{5}{89} \cdot 3\} \\ &= 0.17 \end{aligned}$$

Similarly, $sim(S, D_1) = 1$, because RFM denotes D_1 as a strict "subdocument" of S. So, for $T = 0.80$, RFM would not consider D_2 to be a potential copy of S. However, RFM would find D_1 suspiciously close to S. □

5.3 The PROBE-CDS information about the databases

PROBE-CDS needs information to decide whether a database *db* has potential copies of a suspicious document S. This information should be concise, but also sufficient to identify any such database. We use the same "collector" based approach adopted by *GlOSS* to compute these statistics [GGMT94]. PROBE-CDS maintains the following statistics (or a

subset of them) for each database db and word w_i, where db_i is the set of documents in db that contain w_i:

- $f_i(db) = \min_{D \in db_i} F_i(D)$: $f_i(db)$ is the minimum frequency of word w_i in any document in db that contains w_i
- $F_i(db) = \max_{D \in db_i} F_i(D)$: $F_i(db)$ is the maximum frequency of word w_i in any document in db that contains w_i
- $n_i(db) = \min_{D \in db_i} |D|$: $n_i(db)$ is the minimum norm of any document in db that contains w_i
- $R_i(db) = \max_{D \in db_i} \frac{F_i(D)}{|D|}$: $R_i(db)$ is the maximum value of the ratio $\frac{F_i(D)}{|D|}$ for any document $D \in db$ that contains w_i
- $d_i(db)$ is the number of documents in db that contain word w_i

EXAMPLE 5.3.1 (Example 5.2.1 cont.) The following table shows the PROBE-CDS metadata for our sample database db. Note that there are no entries for w_4, since it does not appear in any document in db.

Statistics	w_1	w_2	w_3
f_i	1	3	5
F_i	1	8	5
n_i	10	10	89
R_i	$\frac{1}{10}$	$\frac{3}{10}$	$\frac{5}{89}$
d_i	1	2	1

As mentioned earlier, db has two documents, D_1 and D_2. Document D_1 contains w_2 three times, and document D_2, eight times. Therefore, $f_2(db) = \min\{3, 8\} = 3$ and $F_2(db) = \max\{3, 8\} = 8$. Also, $|D_1| = 10$ and $|D_2| = 89$, so $n_2(db) = \min\{10, 89\} = 10$. Finally, $R_2(db) = \max\{\frac{3}{10}, \frac{8}{89}\} = \frac{3}{10}$, and $d_2(db) = 2$, since w_2 appears in both D_1 and D_2. □

Notice that the table above is actually larger than our earlier table that gave the complete word frequencies. This case happens because our sample database contains only two documents. In general, the information kept by PROBE-CDS is proportional to the number of words or terms appearing in the database, while the information needed by the CDS

is proportional to the number of words times the number of times the words appear in different documents. In a real database, many words appear in hundreds or thousands of documents, and hence the CDS information is typically much larger than the PROBE-CDS information. We will return to this issue in Section 5.7.

5.4 The conservative approach

Given a set of databases, a suspicious document S, and a threshold T, PROBE-CDS selects all databases with potential copies of S, i.e., all the databases with at least one document D with $sim(S, D) > T$. To identify these databases, PROBE-CDS uses the metadata of Section 5.3. In this section we focus on conservative techniques that never miss any database with potential copies. In other words, PROBE-CDS cannot produce any *false negatives* with the techniques of this section. However, PROBE-CDS might produce *false positives*, and consider that a database has potential copies when it actually does not. In Section 5.7 we report experimental results that study how often the latter takes place.

The information described in Section 5.3 can be used by PROBE-CDS in a variety of ways. We present two alternatives, starting with the simplest. The more sophisticated technique will be less conservative: it will always identify the databases with potential copies of a document, but it will have fewer false positives than the simpler technique.

Given a database db, a suspicious document S, and a technique A, PROBE-CDS computes an upper bound $Upper_A(db, S)$ on the similarity of any document in db and S. In other words,

$$Upper_A(db, S) \geq sim(S, D)$$

for every document $D \in db$. Thus, if $Upper_A(db, S) \leq T$, then there are no documents in db close enough to S as determined by the threshold T, and we can safely determine that database db has no potential copies of S. The two strategies below differ in how they compute this upper bound.

The *Range* strategy

Consider a word w_i in S. Suppose that w_i appears in some document D in db. We know that D contains w_i between $f_i(db)$ and $F_i(db)$ times. Also, w_i is in the closeness set for S and D if and only if $F_i(D) \in Accept(w_i, F_i(S))$. Let $[f_i(db), F_i(db)] \cap Accept(w_i, F_i(S))$ be denoted

by $[m_i, M_i]$. So, w_i is in the closeness set for S and D if and only if $F_i(D) \in [m_i, M_i]$. If the range $[m_i, M_i]$ is empty, then w_i is not in the closeness set for S and D, for any document $D \in db$, and therefore w_i does not contribute to $sim(S, D)$ for any D. If the range $[m_i, M_i]$ is not empty, then w_i can be in the closeness set for S and D, for some document D. For any such document D, $F_i(D) \leq M_i$. We then define the *maximum frequency* of word $w_i \in S$ in any document of db, $M_i(db, S)$, as:

$$M_i(db, S) = \begin{cases} M_i & \text{if } [m_i, M_i] \neq \phi \\ 0 & \text{otherwise} \end{cases}$$

Putting everything together, we define the upper bound on the similarity of any document D in db and S for technique *Range* as:

$$Upper_{Range}(db, S) = \max\{Upper1_{Range}(db, S), Upper2_{Range}(db, S)\}$$

where:

$$Upper1_{Range}(db, S) = \sum_{i=1}^{N} M_i(db, S) \cdot \frac{F_i(S)}{|S|} \quad (5.4.1)$$

$$Upper2_{Range}(db, S) = \sum_{i=1}^{N} \frac{M_i(db, S)}{n_i(db)} \cdot F_i(S) \quad (5.4.2)$$

Note that since $n_i(db) \leq |D|$ for every $D \in db$ that contains w_i, then $Upper_{Range}(db, S) \geq sim(S, D)$ for every $D \in db$. Also note that the *Range* technique does not use the R_i statistics.

EXAMPLE 5.4.1 (Example 5.2.1 cont.) Consider the db statistics and the suspicious document S. We have already computed $Accept(w_1, F_1(S)) = [1, 1]$, $Accept(w_2, F_2(S)) = [2, 5]$, $Accept(w_3, F_3(S)) = [2, 5]$, and $Accept(w_4, F_4(S)) = [5, 17]$. Also, PROBE-CDS knows, for example, that word w_2 appears in db with in-document frequencies between $[f_2(db), F_2(db)] = [3, 8]$. Then, the interesting range of frequencies of w_2 in db is $[m_2, M_2] = [3, 8] \cap [2, 5] = [3, 5]$. The maximum such frequency is $M_2(db, S) = 5$. (Notice that there is no document D in db with $F_2(D) = 5$. $M_2(db, S)$ is in this case a strict upper bound for the frequencies of w_2 in db that are in $Accept(w_2, F_2(S))$.) Similarly, $M_1(db, S) = 1$,

$M_3(db, S) = 5$, and $M_4(db, S) = 0$. Therefore,

$$\begin{aligned} Upper1_{Range}(db, S) &= 1 \cdot \frac{1}{100} + 5 \cdot \frac{3}{100} + 5 \cdot \frac{3}{100} \\ &= 0.31 \\ Upper2_{Range}(db, S) &= \frac{1}{10} \cdot 1 + \frac{5}{10} \cdot 3 + \frac{5}{89} \cdot 3 \\ &= 1.77 \\ Upper_{Range}(db, S) &= 1.77 \end{aligned}$$

Therefore, if our threshold T is, say, 0.80, we would search db. This decision is correct since D_1 in db is indeed a potential copy. □

The *Ratio* strategy

This technique is similar to the previous one, but uses the R_i statistics. Thus,

$$Upper_{Ratio}(db, S) = \max\{Upper1_{Range}(db, S), Upper2_{Ratio}(db, S)\}$$

where:

$$Upper2_{Ratio}(db, S) = \sum_{i|M_i(db,S)\neq 0} \min\{\frac{M_i(db, S)}{n_i(db)}, R_i(db)\} \cdot F_i(S) \qquad (5.4.3)$$

It is immediate from the definition above that $Upper_{Ratio}(db, S) \leq Upper_{Range}(db, S)$ for every database db and query document S. Therefore, *Ratio* is a less conservative technique than *Range*, and will tend to have fewer false positives than *Range*. Nevertheless, *Ratio* will always detect databases with potential copies of S, because $sim(S, D) \leq Upper_{Ratio}(db, S)$ for every $D \in db$.

EXAMPLE 5.4.2 (Example 5.2.1 cont.) We had computed $Upper1_{Range}(db, S)$ to be 0.31. Now,

$$\begin{aligned} Upper2_{Ratio}(db, S)\} &= \frac{1}{10} \cdot 1 + \frac{3}{10} \cdot 3 + \frac{5}{89} \cdot 3 \\ &= 1.17 \end{aligned}$$

which is lower than $Upper2_{Range}$. □

5.5 The liberal approach

The techniques of Section 5.4 are conservative: they never fail to identify a database with potential copies of a suspicious document (i.e., these techniques have no false negatives). A problem with these techniques is that they usually produce too many false positives. (See Section 5.7.) Consequently, we now introduce *liberal* versions of the *Range* and *Ratio* techniques. In principle, the new techniques might have false negatives. As we will see, false negatives occur rarely, while the number of false positives is much lower than that for the conservative techniques.

We modify the techniques of Section 5.4 in two different ways. First, we allow these techniques to focus only on the "rarest" words that occur in a suspicious document, instead of on all its words (or on all the words that CDS uses). (See Section 5.5.1.) This way PROBE-CDS can prune away databases where these rare words do not appear, thus reducing the search space. Second, we allow these techniques to use probabilities to estimate (under some assumptions) how many potential copies of a suspicious document each database is expected to have. (See Section 5.5.2.) Thus, the probabilistic techniques no longer compute upper bounds, again reducing the search space.

5.5.1 Counting only rare words

The techniques of Section 5.4 considered every word in a suspicious document S (i.e., every word that the CDS uses) to decide which databases to search for potential copies of S. Alternatively, PROBE-CDS can just focus on the *rarest* words in S, i.e., on the words in S that appear in the fewest number of databases. PROBE-CDS then decides to search a database only if at least a few of these rare words appear in it. If PROBE-CDS uses enough of the rare words in S, any potential copy of S will tend to contain a few of these words. Furthermore, since these words appear in only a few databases, they will help PROBE-CDS dismiss a significant fraction of the databases, thus reducing the number of false positives.

One specific way to implement these ideas is as follows. Given a suspicious document S, PROBE-CDS just considers k percent of its words. These are the $k\%$ words in S that appear in the fewest available databases. PROBE-CDS can tell which words these are from the metadata about the databases (Section 5.3). The remaining words in S are simply ignored.

EXAMPLE 5.5.1 (Example 5.2.1 cont.) Consider suspicious document S, with words

w_1, w_2, w_3, and w_4. Suppose that w_1 appears in 1 database, w_2 in 2, w_3 in 70, and w_4 in 20 databases. If PROBE-CDS uses only 50% of the words in S ($k = 50$), it chooses w_1 and w_2, and ignores w_3 and w_4. □

As we mentioned before, PROBE-CDS now ignores words in S that the CDS uses for copy detection. Therefore, PROBE-CDS might in principle miss a database with potential copies of S. However, as we will see in Section 5.7, we can find values for k for which PROBE-CDS has very few false negatives, while producing much fewer false positives than with the conservative techniques of Section 5.4.

Given k, we adapt the $Upper_{Range}$ and $Upper_{Ratio}$ bounds of Section 5.4 (Equations 5.4.1, 5.4.2, and 5.4.3) to sum only over the k% rarest words in S. We refer to the new values as Sum_{Range} and Sum_{Ratio}, because they are no longer upper bounds on the similarities of the documents in the databases and S.

As we use fewer words in S (i.e., only k% of them), we need to adjust the threshold T (Section 5.2) for PROBE-CDS accordingly. We refer to the adjusted threshold as T^k. For example, if we are just considering 10% of the words in S, we could compensate by reducing the threshold $T^{10} = 0.10 * T$. We explore different values for T^k in Section 5.7. If $Sum_{Range}(db, S)$ (respectively, $Sum_{Ratio}(db, S)$) is higher than T^k, PROBE-CDS will search db for potential copies of S.

EXAMPLE 5.5.2 (Example 5.2.1 cont.) In Section 5.4 we computed $Upper_{Range}(db, S) = 1.77$. Now, if PROBE-CDS only considers the 50% rarest words in S (i.e., w_1 and w_2), only those words are counted, and we have:

$$\begin{aligned} Sum1_{Range}(db, S) &= 1 \cdot \frac{1}{100} + 5 \cdot \frac{3}{100} \\ &= 0.16 \\ Sum2_{Range}(db, S) &= \frac{1}{10} \cdot 1 + \frac{5}{10} \cdot 3 \\ &= 1.6 \\ Sum_{Range}(db, S) &= 1.6 \end{aligned}$$

The original CDS threshold was $T = 0.80$. Since we are now considering only half of the words, we could scale down T to, say, $T^{50} = 0.5 \cdot T = 0.40$. At any rate, we would still search db, because $1.6 > 0.40$. This decision is correct, since D_1 in db is indeed a potential copy. □

5.5.2 Using probabilities

So far, the techniques for PROBE-CDS compute the maximum possible contribution of each word considered, and add these contributions. However, it is unlikely that any document in a database will contain all of these words with this maximum contribution. In this section, we depart from this "deterministic" model, and, given a database db, try to bound the probability that db has potential copies of a suspicious document. If this probability is high enough, PROBE-CDS will search db.

Our goal is to bound the probability that a document in db has a similarity with S that exceeds the adjusted threshold T^k. For this, we define two random variables $XRange1$ and $XRange2$ (corresponding to $Sum1_{Range}$ and $Sum2_{Range}$, respectively). These variables model the similarity of the documents in db and S. Then,

$$Prob_{Range} = \max\{P(XRange1 > T^k), P(XRange2 > T^k)\}$$

If $Prob_{Range} \geq \frac{1}{|db|}$, PROBE-CDS will search db for potential copies of S, since there is at least one expected document that exceeds the adjusted threshold T^k.

Actually, instead of computing $P(XRange1 > T^k)$ and $P(XRange2 > T^k)$, we use an upper bound for these values as given by Chebyshev's inequality. This bound is based on the expected value and the variance of $XRange1$ and $XRange2$.

We now define random variable $XRange1$, following the definition of $Sum1_{Range}$. (Random variable $XRange2$ is analogous, using the definition of $Sum2_{Range}$.) The $XRange1$ is actually a sum of random variables:

$$XRange1 = XRange1_{i_1} + \ldots + XRange1_{i_s}$$

where $w_{i_1}, \ldots, w_{i_s}$ are the $k\%$ rarest words in S. Random variable $XRange1_i$ corresponds to word w_i:

$$XRange1_i = \begin{cases} M_i(db, S) \cdot \frac{F_i(S)}{|S|} & \text{with probability } \frac{d_i(db)}{|db|} \\ 0 & \text{with probability } 1 - \frac{d_i(db)}{|db|} \end{cases}$$

This variable models the occurrence of word w_i in the documents of database db. Word w_i occurs in $d_i(db)$ documents in db, so the probability that it appears in a randomly chosen document from db is $\frac{d_i(db)}{|db|}$.

Technique	Conservative?	Words used	Word contribution	Similarity estimate
$Upper_{Range}$	Yes	All	*Range*	Maximum
$Upper_{Ratio}$	Yes	All	*Ratio*	Maximum
Sum_{Range}	No	rarest $k\%$	*Range*	Maximum
Sum_{Ratio}	No	rarest $k\%$	*Ratio*	Maximum
$Prob_{Range}$	No	rarest $k\%$	*Range*	Expected
$Prob_{Ratio}$	No	rarest $k\%$	*Ratio*	Expected

Table 5.1: Summary of the PROBE-CDS techniques, where $k < 100$. (When $k = 100$, $Upper_{Range}$ and $Upper_{Ratio}$ coincide with Sum_{Range} and Sum_{Ratio}, respectively.)

The expected value of $XRange1$ is:

$$\begin{aligned} E(XRange1) &= \sum_{j=1}^{s} E(XRange1_{i_j}) \\ &= \sum_{j=1}^{s} M_{i_j}(db, S) \cdot \frac{F_{i_j}(S)}{|S|} \cdot \frac{d_{i_j}(db)}{|db|} \end{aligned}$$

If we assume that words appear in documents following independent probability distributions, the variance of $XRange1$ is:

$$\begin{aligned} V(XRange1) &= \sum_{j=1}^{s} V(XRange1_{i_j}) \\ &= \sum_{j=1}^{s} (M_{i_j}(db, S) \cdot \frac{F_{i_j}(S)}{|S|})^2 \cdot \frac{d_{i_j}(db)}{|db|} \cdot (1 - \frac{d_{i_j}(db)}{|db|}) \end{aligned}$$

We define $Prob_{Ratio}$ in a completely analogous way. Table 5.1 summarizes the different strategies that PROBE-CDS can use.

5.6 Searching the databases with potential copies

Once PROBE-CDS has decided that a database *db* might have potential copies of a suspicious document S, it has to extract these potential copies from *db*. If database *db* happens to run a local CDS server, PROBE-CDS can simply submit S to this server and get back exactly those documents that the CDS considers potential copies. However, if *db* does not run a CDS server, we need an alternative mechanism. For this, we will assume that *db* can answer

boolean "or" queries, which most commercial search engines support. For example, we can retrieve from *db* all documents containing the words "copyright" or the word "violation" by issuing the query "copyright or violation." (Alternatively, if some search engine does not support "or" queries, we could issue a sequence of queries, and then merge the sequence of results.)

Let $w_1, \ldots, w_N$ be the words in S. In principle, we could issue the query $w_1 \vee \ldots \vee w_N$ to *db* and obtain all documents that contain at least one of these words. However, such a query is bound to return too many useless documents. In this section, we study how to choose a smaller set of words $\{w_{i_1}, \ldots, w_{i_n}\}$ that will not miss any potential copy from *db*. Furthermore, we would like to choose the "cheapest" such set according to the definitions below.

To choose a set of words to query, we define the *maximum contribution* $C_i(db, S)$ of word w_i in *db* as an upper bound on the amount that w_i can add to $sim(S, D)$, for any $D \in db$. We give two definitions of this maximum contribution, each corresponding to a technique of Section 5.4. The first of these is more conservative but uses less information. The other is less conservative but uses more information.

$$C_i(db,S) \;=\; \begin{cases} \max\{M_i(db,S)\cdot\frac{F_i(S)}{|S|},\;\; \frac{M_i(db,S)}{n_i(db)}\cdot F_i(S)\} & \text{for } \textit{Range} \\ \max\{M_i(db,S)\cdot\frac{F_i(S)}{|S|},\;\; \min\{\frac{M_i(db,S)}{n_i(db)}, R_i(db)\}\cdot F_i(S)\} & \text{for } \textit{Ratio} \end{cases}$$

Now, let $C(db, S) = \sum_{i=1}^{N} C_i(db, S)$, and let T be the CDS similarity threshold that the users specified. Then, any set of words $\{w_{i_1}, \ldots, w_{i_n}\}$ with the following property is sufficient to extract all the potential copies of S from *db*:

$$\sum_{j=1}^{n} C_{i_j}(db, S) \geq C(db, S) - T \tag{5.6.1}$$

To see why it is enough to use the query $w_{i_1} \vee \ldots \vee w_{i_n}$, consider a document $D \in db$ that does not contain any of these n words. Then, $sim(S, D) \leq C(db, S) - \sum_{j=1}^{n} C_{i_j}(db, S) \leq T$. Therefore, the similarity of D and S can never exceed the required threshold T. This approach is conservative: we cannot miss any potential copy of a document by choosing the query words as above. Alternatively, we explored a liberal approach that would retrieve all potential copies most of the time, but has much fewer "false positives." We revisit this approach in Section 5.7.

To choose among all sets of words that satisfy Condition 5.6.1, we associate a cost p_i with each word w_i. We then choose a set of words $\{w_{i_1}, \ldots, w_{i_n}\}$ that satisfies Condition 5.6.1 and minimizes $\sum_{j=1}^{n} p_{i_j}$. We consider two different cost models for a query:

The *WordMin* cost model

In this case we minimize the number of words that will appear in the query. Thus, $p_i = 1$ for all i. Then, our problem reduces to finding the smallest set of words that satisfies Condition 5.6.1. We can find such an optimal set in the following way. Assume that we sorted the words in S according to their maximum contribution, so that $C_i(db, S) \geq C_j(db, S)$ if $i \leq j$. Then, we can choose the smallest n such that $\sum_{i=1}^{n} C_i(db, S) \geq C(db, S) - T$. A query that retrieves all the desired documents using the smallest number of words is $w_1 \vee \ldots \vee w_n$.

The *SelMin* cost model

In this case we consider the selectivity of each word w_i that will appear in the query, i.e., the fraction of the documents in the database that contain word w_i. Thus, $p_i = Sel(w_i, db)$. By minimizing the added selectivity we will tend to minimize the number of documents that we retrieve from db.

We will find an optimal solution for this problem by reducing it to the *0-1 knapsack problem* [CLR91]. The new formulation of the problem is as follows. A thief robbing a store finds N items (the words). The ith item is worth p_i dollars (the selectivity of word w_i) and weighs $C_i(db, S)$ pounds (the maximum contribution of w_i). The thief wants to maximize the value of the load, but can only carry up to T pounds. The problem is to find the right items (words) to steal. This formulation of the problem actually finds the words that will not appear in the final query, and maximizes the added selectivity of these words. The weight of the words is at most T. Therefore, the words that are not chosen weigh at least $C(db, S) - T$, satisfy Condition 5.6.1, and have the lowest added selectivity among the sets satisfying Condition 5.6.1. Assuming that T, the C_i's, and the p_i's have a fixed number of significant decimals, we can use dynamic programming to solve the problem in $O(T \cdot N)$ time, where N is the number of words in the suspicious document [CLR91].

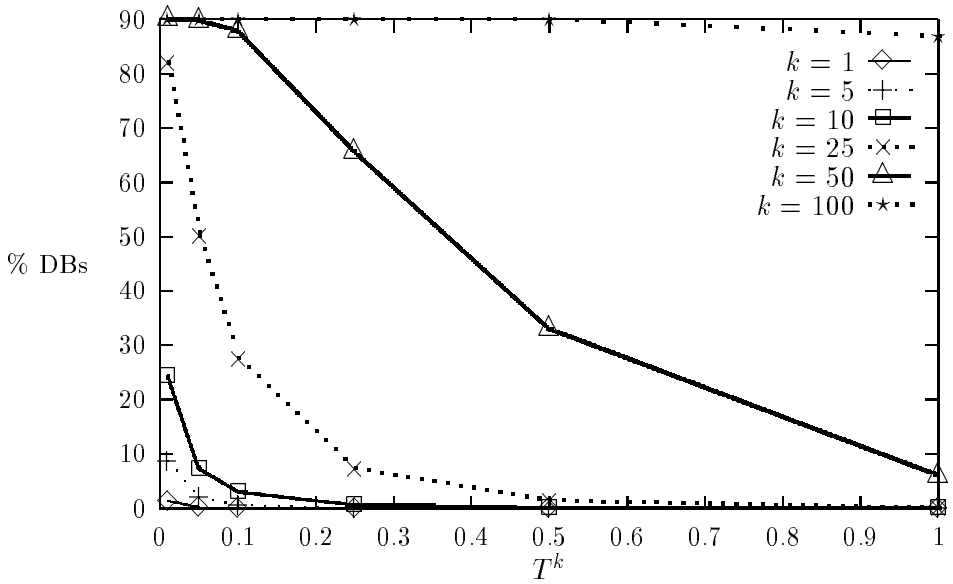


Figure 5.3: The percentage of the 50 databases that are searched as a function of the adjusted similarity threshold T^k (*Registered* suspicious documents; Sum_{Ratio} strategy; $T = 1$).

5.7 Experiments

This section presents experimental results for PROBE-CDS. We focus on three sets of issues: How many false positives do the PROBE-CDS techniques report, how many false negatives do the liberal PROBE-CDS techniques produce, and how effective is the document extraction step?

For the registered document databases, our experiments used a total of 63,350 ClariNet news articles. We split these articles evenly in 50 databases so that each database consists of 1,267 documents.

For the suspicious documents, our experiments used two different document sets. The first set, which we refer to as *Registered*, contains 100 documents from the 50 databases. Therefore, each suspicious document has at least one perfect copy in some database. (There could be more copies due to crosspostings of articles.) The second set, which we refer to as *Disjoint*, contains 100 later articles that do not appear in any of the 50 databases. This set models the common case when the suspicious documents are actually new documents that do not appear anywhere else.

Our first experiments are for the Sum_{Ratio} technique, which proved to work the best

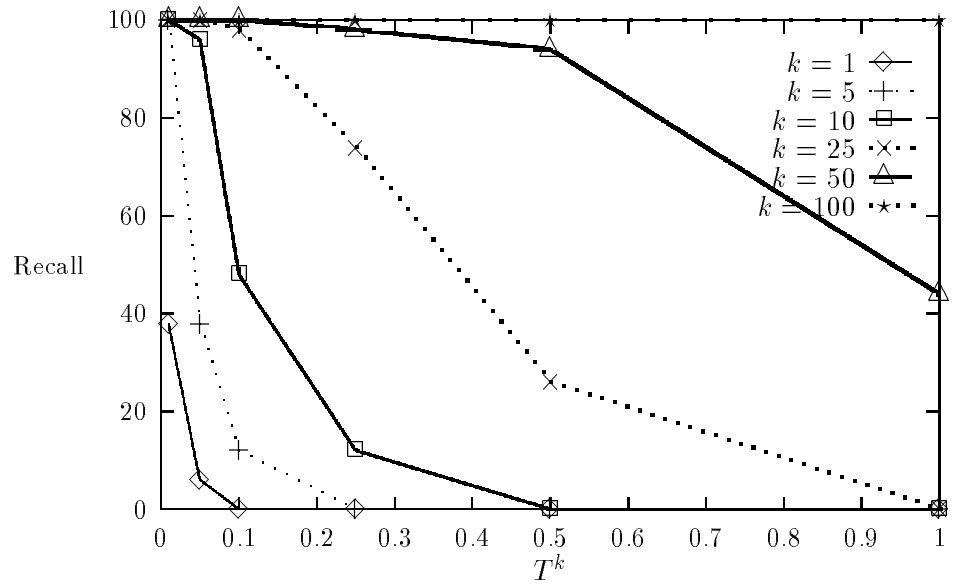


Figure 5.4: The average recall as a function of the adjusted similarity threshold T^k (*Registered* suspicious documents; Sum_{Ratio} strategy; $T = 1$).

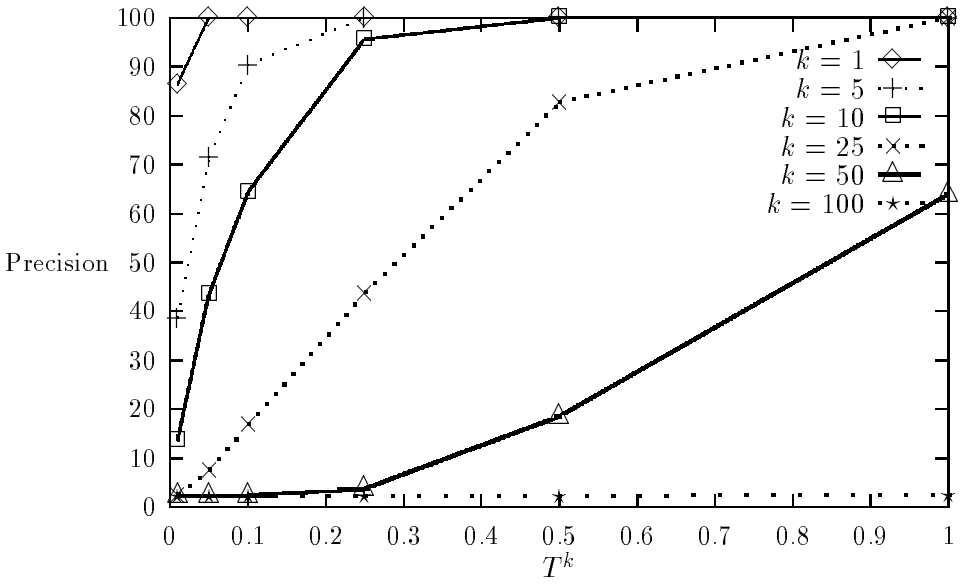


Figure 5.5: The average precision as a function of the adjusted similarity threshold T^k (*Registered* suspicious documents; Sum_{Ratio} strategy; $T = 1$).

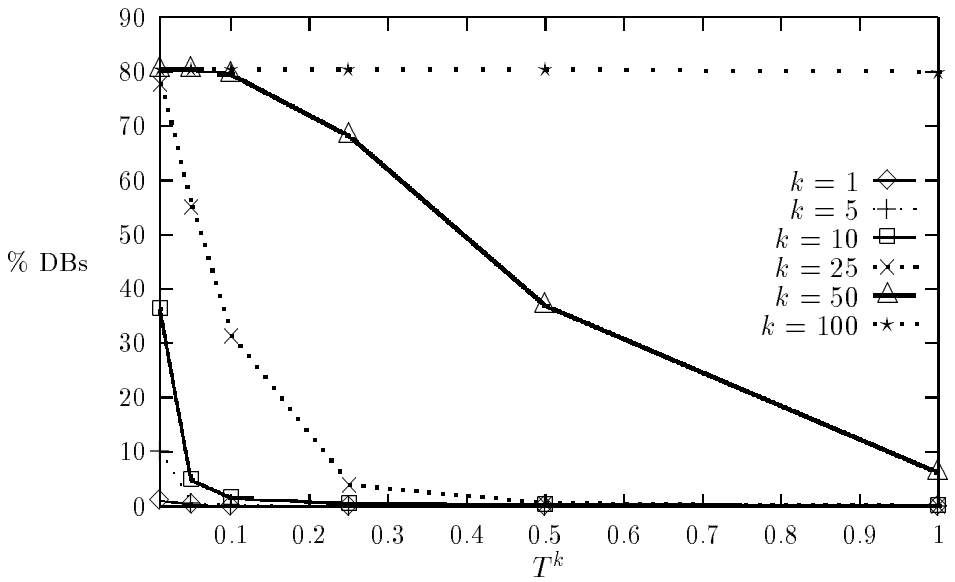


Figure 5.6: The percentage of the 50 databases that are searched as a function of the adjusted similarity threshold T^k (*Disjoint* suspicious documents; Sum_{Ratio} strategy; $T = 1$).

among the PROBE-CDS techniques of Table 5.1, as we will see later. Figures 5.3 through 5.6 show different interesting metrics as a function of the adjusted threshold T^k, and for different values of k. In all of these plots, the CDS threshold T is set to 1. For example, the curves for $k = 10$ correspond to considering only 10% of the words (the rarest ones) in the suspicious documents. Note that for $k = 100$ all of the words in the suspicious documents are used. In this case, Sum_{Ratio} coincides with the conservative technique $Upper_{Ratio}$.

One way to evaluate the PROBE-CDS strategies is to look at $d(S)$, the percentage of databases returned by PROBE-CDS for a suspicious document S. Figure 5.3 shows the average $d(S)$ (averaged over all S in the *Registered* set), as a function of T^k. The more words PROBE-CDS considers from the suspicious documents (i.e., the higher k), the more databases are searched: PROBE-CDS considers the words as ordered by how rare they are. Therefore, when PROBE-CDS starts considering "popular" words, more databases will tend to exceed the similarity threshold T^k. Also, for a fixed k, the higher T^k, the fewer databases that PROBE-CDS searches, since only databases that exceed T^k are searched. For low values of k, PROBE-CDS searches very few databases. For example, for $k = 10$ and $T^k = 0.05$, less than 10% of the databases are searched.

As we know, Sum_{Ratio} may produce false negatives for $k < 100$, i.e., it may tell us not

to search databases where the CDS would find potential copies. It is interesting to study what percentage of the databases with potential copies PROBE-CDS actually searches (or equivalently, what percentage of these databases are not false negatives). Let $s(S)$ be the number of databases with potential copies of S according to the CDS, and let $s'(S)$ be the number of databases with potential copies of S that PROBE-CDS searches. Then, the *recall* of the technique used by PROBE-CDS is the average value of $\frac{100 \cdot s'(S)}{s(S)}$ over our suspicious documents S.

Figure 5.4 shows the recall values for Sum_{Ratio} as a function of the adjusted threshold T^k. This figure is very similar to Figure 5.3: the more databases a technique searches, the higher its recall tends to be. Note, however, that some techniques have very few false negatives, while they search a low percentage of the databases. For example, for $k = 10$ and $T^k = 0.05$, recall is above 90%, meaning that for the average suspicious document, 90% of the databases with potential copies are chosen by PROBE-CDS. As we have seen, just under 10% of the databases are searched for this value of k and T^k.

As we mentioned above, PROBE-CDS produces false positives. We want to measure what percentage of the databases selected by PROBE-CDS actually contains potential copies. The *precision* of the technique used by PROBE-CDS is the average value of $\frac{100 \cdot s'(S)}{d(S)}$ over our suspicious documents S. Figure 5.5 shows the precision values for Sum_{Ratio} as a function of the adjusted threshold T^k. As expected, the more databases a technique searches, the lower its precision tends to be. For $k = 10$ and $T^k = 0.05$, precision is over 40%, meaning that for the average suspicious document, over 40% of the databases that PROBE-CDS searches have potential copies of the document. Actually, this choice of values for k and T^k is a good one: PROBE-CDS searches very few databases while achieving high precision and recall values.

We are evaluating PROBE-CDS in terms of how well it predicts the behavior of the CDS at each database. However, the CDS can sometimes be wrong as discussed in Chapter 2. For example, the CDS can wrongly flag a document D in *db* as a copy of a suspicious document S. PROBE-CDS might then also flag *db* as having potential copies of S. However, we do not "penalize" PROBE-CDS for this "wrong" choice: the best PROBE-CDS can do is to predict the behavior of the CDS, and that is why we define precision and recall as above. It would be unreasonable to ask a system like PROBE-CDS, with very limited information about the databases, to detect copies more accurately than a CDS system with complete information about the database contents.

To illustrate the storage space differences between PROBE-CDS and the CDS, let us consider the data that we used in our experiments. In this case, there are around 4 million word-document pairs, which is the level of information that a CDS server needs, whereas there are only around 791,000 word-database pairs, which is the level of information that a PROBE-CDS server needs. As the databases grow in size, we expect this difference to widen too, since the PROBE-CDS savings in storage come from words appearing in multiple documents. For example, if we consider our 50 databases as a single, big database, PROBE-CDS needs only 138,086 word-database pairs, whereas the CDS data remains the same. Therefore, PROBE-CDS has just around 3.36% as many entries as the CDS. We are considering alternatives to reduce the size of the PROBE-CDS data even further. As an interesting direction for future work, PROBE-CDS can store information on, say, only the 10% rarest words. Most of the time, the 10% rarest words that appear in a suspicious document will be among these 10% overall rarest words, so PROBE-CDS can proceed as usual. With this scheme, the PROBE-CDS space requirements would be cut further by an order of magnitude.

Figure 5.6 shows results for the *Disjoint* set of suspicious documents, again for the Sum_{Ratio} technique and $T = 1$. There are no potential copies of these documents in any of the 50 databases. Therefore, recall is always 100%, and precision is 0% if some database is selected. It then suffices to report the percentage of databases chosen for these documents (Figure 5.6). These values tend to be lower in general than those for the *Registered* suspicious documents of Figure 5.3, which is the right trend, since no database contains potential copies of the suspicious documents. For example, for $k = 10$ and $T^k = 0.05$, less than 5% of the databases are searched.

So far we have presented results just for the Sum_{Ratio} technique. Figures 5.7 through 5.10 show results also for Sum_{Range}, $Prob_{Range}$, and $Prob_{Ratio}$, as a function of the CDS threshold T. In all of these plots, we have fixed $k = 10$ and $T^k = 0.05 \cdot T$, which worked well for both the *Registered* and the *Disjoint* suspicious documents when $T = 1$. In Figure 5.7, $Prob_{Range}$ and $Prob_{Ratio}$ search fewer databases than Sum_{Range} and Sum_{Ratio}, at the expense of significantly lower recall values (Figure 5.8). Sum_{Range} and Sum_{Ratio} have very high recall values (above 95% for all values of T). Precision is also relatively high, especially for the Sum_{Ratio} strategy (Figure 5.9). From all these plots, Sum_{Ratio} appears as the best choice for PROBE-CDS, because of its high recall and precision, and low percentage of databases that it searches. Furthermore, Sum_{Ratio} is the technique that searches the fewest databases for the *Disjoint* suspicious documents (Figure 5.10). Also, note that

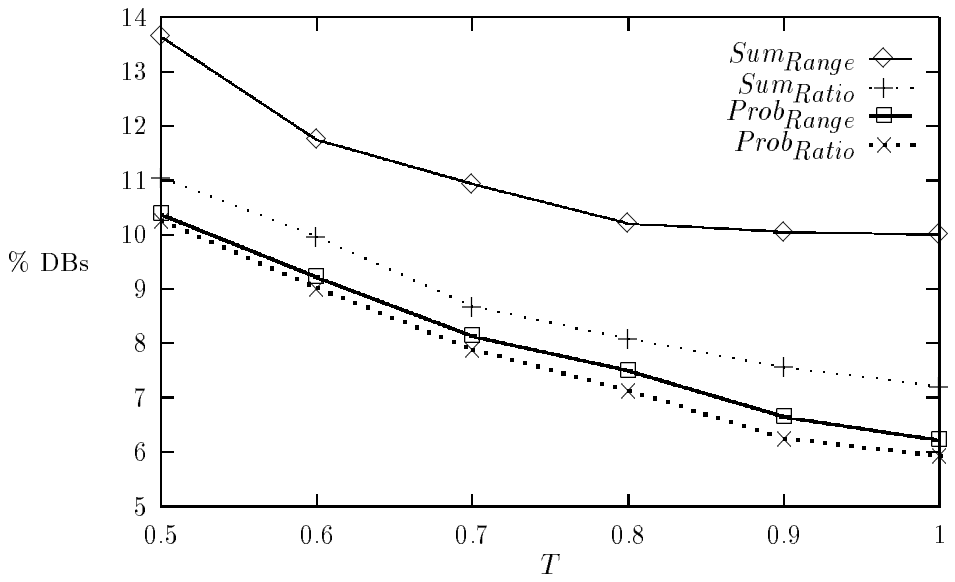


Figure 5.7: The percentage of the 50 databases that are searched as a function of the CDS threshold T (*Registered* suspicious documents; $k = 10$; $T^k = 0.05 \cdot T$).

Sum_{Ratio} does not need the d_i statistics, resulting in lower storage requirements than those of $Prob_{Ratio}$, for example. However, if we want to be conservative, and be sure that we do not miss any potential copy of a document, then the best choice is also Sum_{Ratio}, but with $k = 100$ and $T^k = T$. (This technique coincides with the conservative $Upper_{Ratio}$ technique of Section 5.4.)

To determine whether the results above will still hold for larger databases, we performed the following experiment. Initially we have a single database with 1,267 documents (one of the databases that we used in this Section). PROBE-CDS decides whether this database should be searched or not for each of the *Disjoint* suspicious documents, with $T = 1$, the Sum_{Ratio} strategy, $k = 10$, and $T^k = 0.05$. The answer should be "no" for each of these documents, of course. Figure 5.11 shows that PROBE-CDS decides to search this database for less than 10% of the tested documents. This decision corresponds to a 0.10 probability of false positives. Then, we keep enlarging our only database by progressively adding the documents from our original databases, until the database consists of all 63,350 documents. As we see from Figure 5.11, after an initial deterioration, PROBE-CDS stabilizes and chooses to search the database around 25% of the time. These important results show that PROBE-CDS scales relatively well to larger databases. That is, the probability of false

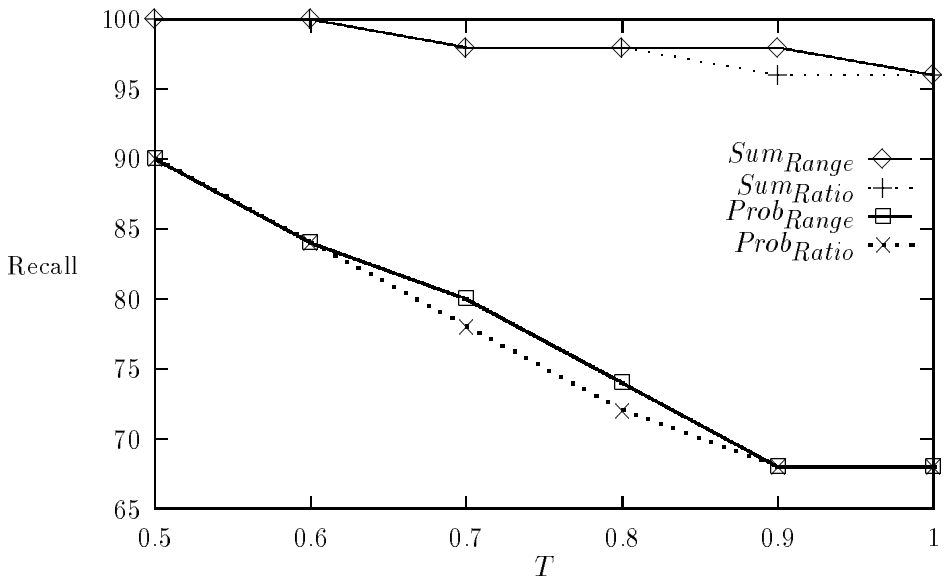


Figure 5.8: The average recall as a function of the CDS threshold T (*Registered* suspicious documents; $k = 10$; $T^k = 0.05 \cdot T$).

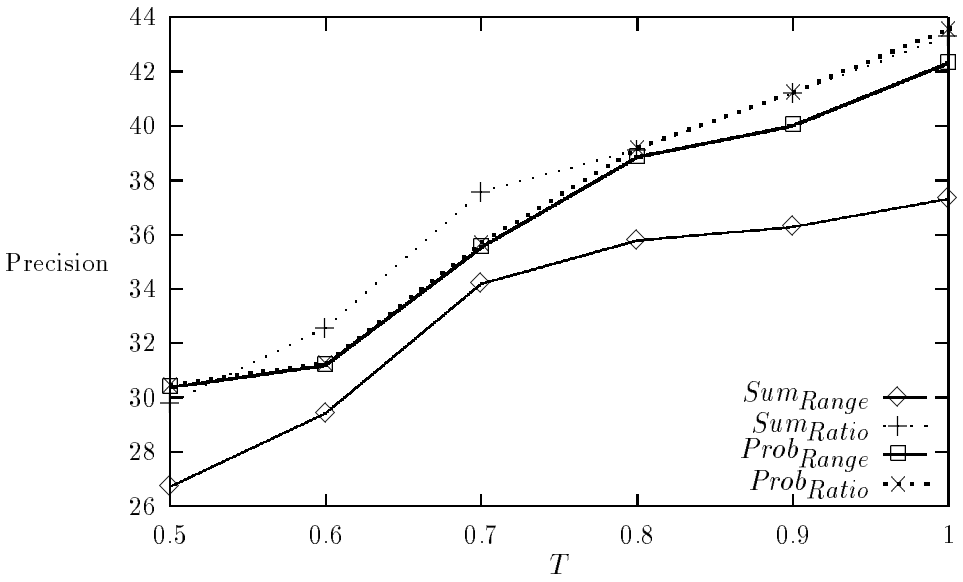


Figure 5.9: The average precision as a function of the CDS threshold T (*Registered* suspicious documents; $k = 10$; $T^k = 0.05 \cdot T$).

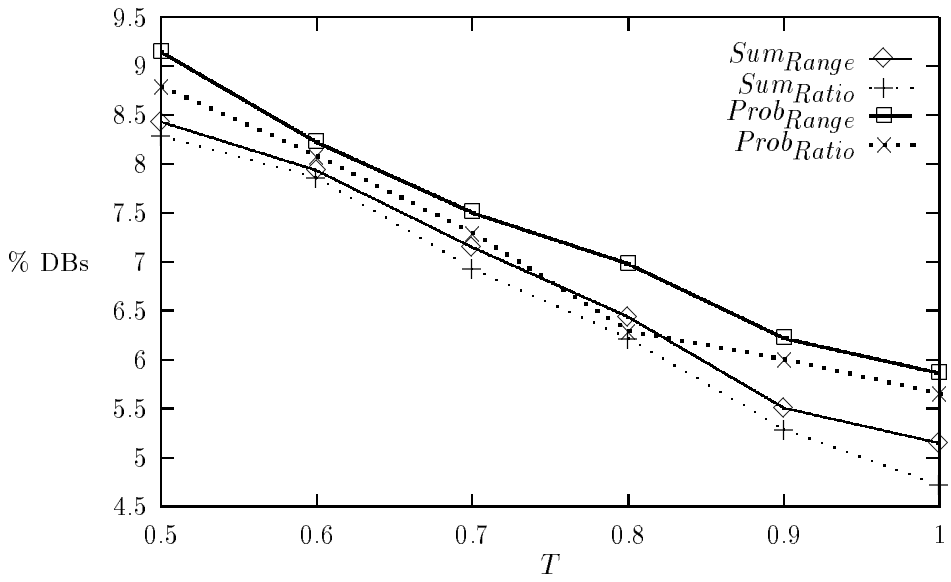


Figure 5.10: The percentage of the 50 databases that are searched as a function of the CDS threshold T (*Disjoint* suspicious documents; $k = 10$; $T^k = 0.05 \cdot T$).

positives is relatively insensitive (after an initial rise) to database size. Notice, incidentally, that the 25% false positive probability can be made smaller by changing the T^k and k values (at a cost in false negatives). So the key observation from this figure is simply that the value is flat as the database size grows.

5.7.1 Results for extraction techniques

Our final set of experiments is for the results of Section 5.6. In that section we studied how to choose the query for each database that PROBE-CDS selects. These queries retrieve all potential copies of the suspicious documents. There are many such queries, though. We presented two cost models, and showed algorithms to pick the cheapest query for each model.

Under our first cost model, *WordMin*, we minimize the number of words in the queries that we construct. Thus, we choose a minimum set of words for our query from the given suspicious document. Figure 5.12 shows the percentage of words in the suspicious document that are chosen to query the databases, for the *Registered* documents and for different values of T. The number of words in the queries decreases as T increases. In effect, Condition 5.6.1 in Section 5.6 becomes easier to satisfy for larger values of T. For example, for $T = 0.80$

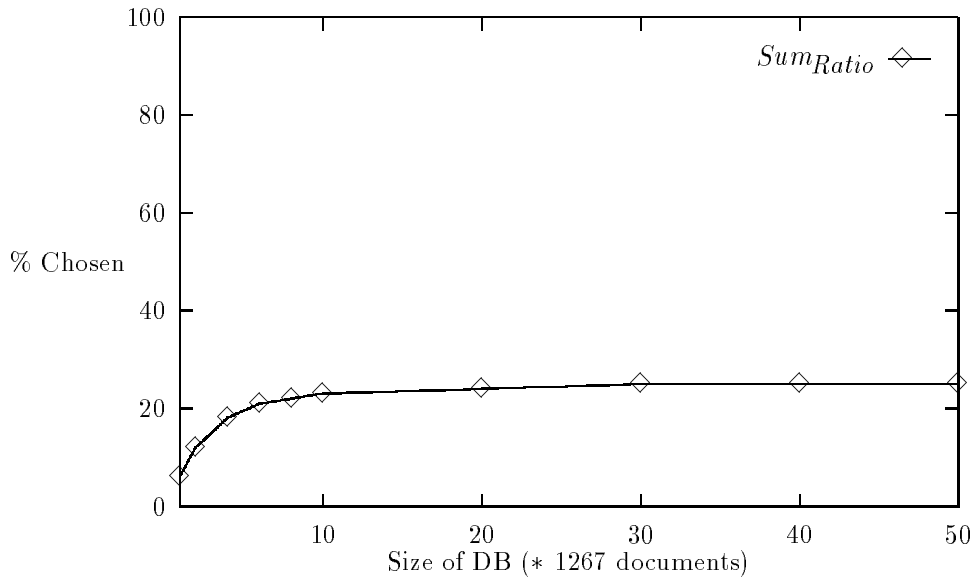


Figure 5.11: The average number of times that PROBE-CDS (incorrectly) chooses to search the (growing) database, as a function of the size of the database (*Disjoint* suspicious documents; Sum_{Ratio} strategy).

and *Ratio*, we need on average 9.99% of the suspicious-document words for our queries. If a particular database cannot handle so many words in a query, we should partition the query into smaller subqueries, and take the union of its results. As expected, the number of words chosen using the *SelMin* cost model is higher, because this cost model focuses on the selectivity of the words, and not on the number of words chosen.

While our second cost model, *SelMin*, uses the word selectivities, the *WordMin* cost model ignores these selectivities. Therefore, we analyze the selectivity for the queries to know what fraction of each database we will retrieve with such queries. Figure 5.13 shows the average value of this selectivity for the *Registered* suspicious documents.

The number of query words and the added selectivity of the query words are relatively high. However, if all a database has is a Boolean-query interface, we have no choice but to ask the right queries to the database to extract all the potential copies of a suspicious document. (How to deal with a vector-space query interface [Sal89] is part of our future work.) The results above show that we can do substantially better than the brute-force approach (i.e., when we use all the words in the suspicious document to build a big "or" query) by writing the queries as in Section 5.6.

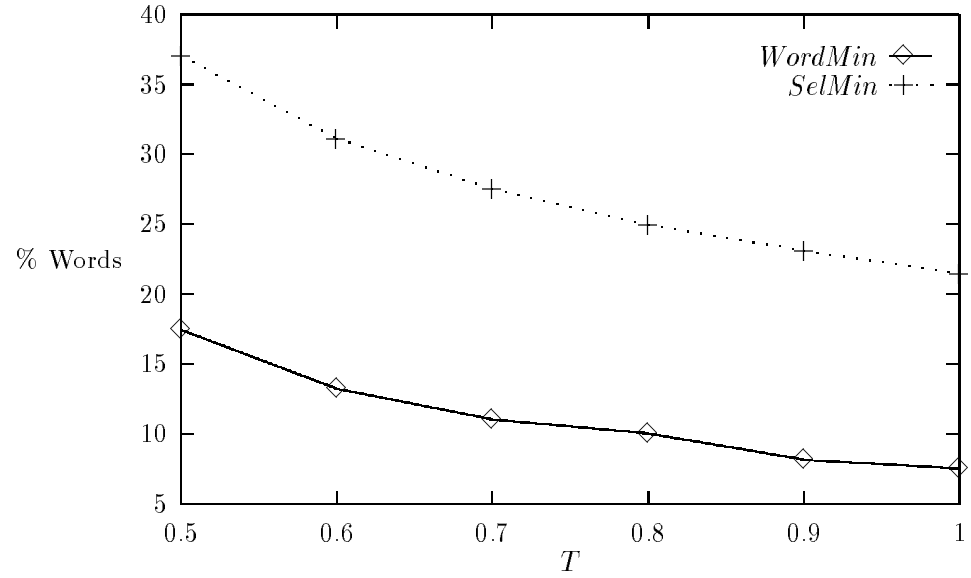


Figure 5.12: The percentage of words of the suspicious documents that are included in the query to extract the potential copies from the databases (*Registered* suspicious documents; *Ratio* strategy).

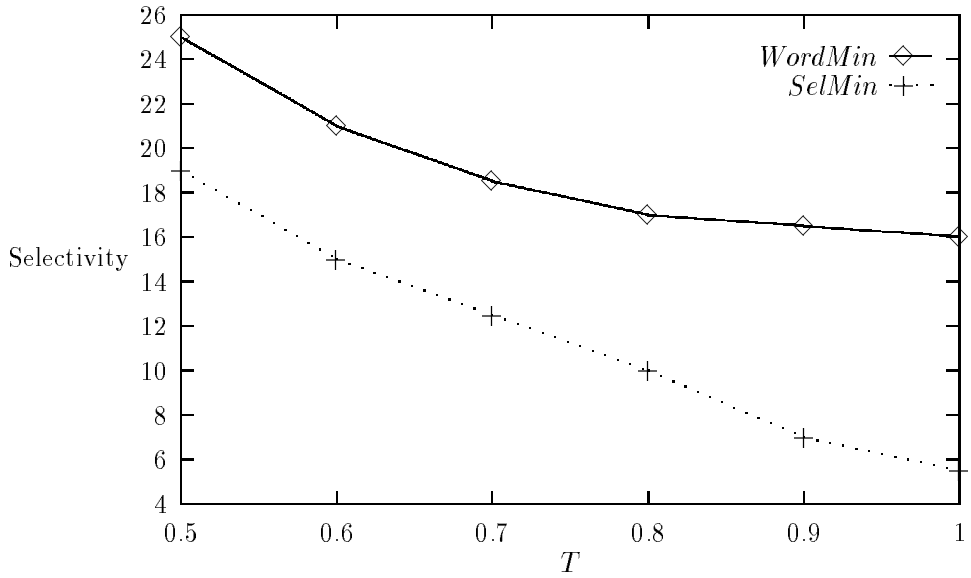


Figure 5.13: Average selectivity of the queries used to extract the potential copies from the databases, as a function of the CDS threshold T (*Registered* suspicious documents; *Ratio* strategy).

5.8 Conclusion

Discovering a potential copy that might exist in one of many databases is a fundamentally difficult problem. One might say that it is harder than finding a "needle in a haystack:" the haystack is distributed across the Internet, we do not want *similar* items (e.g., a nail), and we also want to find any *piece* of the needle if it exists. It is a harder problem than simply finding similar items, as in traditional information retrieval. Given this difficulty it is somewhat surprising that PROBE-CDS performs as well as we have found, especially when one considers the relatively small amount of index information it maintains. It is true that PROBE-CDS can miss some potential copies or can lead us to sites without copies, but with the right algorithm and parameter settings, these errors can be made tolerable. For example, we found that PROBE-CDS can miss fewer than 5% of the sites with potential copies, and for the sites it does lead us to, they actually have a potential copy roughly half the time.

Chapter 6

Building a Video Comparator

6.1 Introduction

We first discuss domain-specific attacks from a cyber-pirate that our video CDS should be resilient to. Keeping these attacks in mind, we discuss how to extract features and compute similarity between video sequences. We also discuss how to build novel index structures so we can execute the FIND and FIND-ALL operations efficiently. We experimentally evaluate our similarity measures and our index structures.

6.2 Attacks from a cyber-pirate

As in the case of the text CDS (Chapter 2), a cyber-pirate can modify his copy of a video in a variety of ways before making it available on the web, including the following.

1. **Format conversions:** The cyber-pirate can convert a video from one encoding format (e.g., MPEG, QuickTime or RealVideo) into another format. In addition, the screen version of a movie is shown at 24 frames per second (fps), while the television versions in North American and Europe are at 30 fps and 25 fps respectively. However, the MPEG version of the movie may be available on the web at 15 fps.

2. **Partial clips:** Similar to the textual case, the cyber-pirate may offer only a clip of the movie or may compile a video sequence out of several movies.

3. **Spatial attacks:** The aspect ratio (width versus height) of the on-screen version of a movie is different from the DVD and TV version of the same movie. The MPEG

version of the movie may be available at a different aspect ratio. Also the cyber-pirate may degrade the video quality by resampling the sequence, by lowering the overall brightness, adding a minor red shift to the video images or by adding random jitter to the copy.

The features we extract from a video should be resilient to "attacks" such as the above. That is, a cyber-pirate should not be able to fool our system by resizing the aspect ratio, resampling the sequence, converting the format or degrading video quality (e.g., by lowering the overall brightness or adding a minor red shift to defeat color based signatures), or simply copying a small video clip from a movie.

6.3 Feature extraction

Say we are given video clips v and v' which are similar to each other (e.g., same movie at different frame rates), and video clip w which is unrelated to v and v'. We need to extract features from these clips so that the features for v and v' are "similar," but are dissimilar to the features of clip w. Feature identification and extraction is a topic of extensive research in the content-based video retrieval community [FSN95, Gev95, OS95, Sri95, ACM96, DM97]. We outline some of these approaches and discuss their benefits and disadvantages.

We first present some terminology adopted in film theory. We can decompose a structured narrative such as a television news program, sports broadcast or a movie as a sequence of *scenes*. Each scene can be further decomposed as a set of *shots*, with each shot representing a single camera viewpoint. Each shot is made up of a set of individual frames. Each level of description (frames, shots, scenes and video) reveals different information. Typically, the description is more semantic at the higher end (video) and more syntactic at the lower end (frames) of the hierarchy.

6.3.1 Using spatial signatures

The common approach to content-based retrieval is the frame-based approach, where we consider each video sequence to be a sequence of images. That is, videos v, v' and w are decoded into sequences of still images. Then the images in each of these videos are compared using image comparison techniques [FSN95, JV96].

This approach suffers from two problems. Firstly, it is computationally expensive: (1) the histogram intersection process required to compute similarity between two images is

expensive [SB91], and (2) the number of such comparisons is very large. For example, an hour-long movie at 30fps has 108,000 images. If we compare this movie against a similar movie, we would need to perform quadratic number of pair-wise image comparisons. One way to reduce the number of comparisons is to extract *key frames* and only compute intersections for these frames. For example, we can choose one frame per movie scene to summarize that scene rather than use all the shots in that scene. Despite such optimizations to reduce the number of images to compare, this approach is still very expensive [IL98].

Secondly, this approach is not resilient to the spatial attacks we mentioned in the last section. For instance, an attacker can easily perform a perceptually undetectable color or illumination shift to each shot. However, current image comparison techniques are not resilient to such attacks.

6.3.2 Using temporal signatures

We exploit the fact that video, in contrast to still images, also has a temporal dimension. That is, information in video is not only revealed in spatial arrangements but also in the temporal evolution of events. Researchers have often interpreted the "meaning" of a sequence in different ways based on the temporal composition of individual shots that make up the sequence [RM82, Dmy84]. For instance, Iyengar et al. [IL98] showed how to analyze shot durations and activity within shots to *classify* television sequences and movie trailers based on the following intuition.

- **Action movies:** Action movies typically have very "busy imagery" in terms of high speed chases, explosions and flying debris. For example, the gutter motorcycle chase in Terminator II employs quick shot transitions to embody action. Similarly, explosions with flying debris last only a few seconds, but translate to high energy motion within the shots.

- **Character movies:** Character movies typically try to develop the complexities in its characters. In such movies, frequent shot changes are very distracting to the viewer during long dialogues and closeups for emotional moments. Hence the director uses very few shot changes, and each shot typically lasts longer than in an action movie (it is hard to have meaningful dialogues for less than a few seconds).

Based on the above intuition, we propose to use shot durations (the timing patterns of when shots change) as a signature for video clips. For instance, if shot transitions occur

in the video clip at time instants [4.2, 22.3, 45.2, ...] seconds, we use this *timing sequence* as the video's signature. (We will later discuss how to compute these shot transitions automatically, based on changes in the camera positions.)

- **Why does this signature make sense?** As we mentioned earlier, two movies in different genres (such as action versus character movies) will have different signatures. In addition, the timing sequences for two movies even within the same genre are typically different because shots evolve based on a variety of factors such as the particular director's style, the story, and the actors. Hence we believe the timing sequence of a movie is a good discriminating feature, and we will indeed ratify this in our experiments in Section 6.7. Also it is easy to see that different versions of the same movie will have similar signatures. Of course, the timing signatures will not be in phase for two different versions of the same clip, if the two clips start at different time instants (i.e., shots are shifted in time). We will later discuss how to handle attacks such as partial clips, translations in time and jittered video sequences.

- **Problems with such signatures:** A human may view movies and their remakes to be the "same" (e.g, Hitchcock's Psycho and the Gus Van Sant remake in 1998). But the timing sequences of the two movies are likely unrelated and our techniques will denote them to be different. However recall that our goal is not to compare the content of the two movies, but merely to compare if two versions of the same movie are "syntactically" similar for copy detection.

  Signatures based on timing sequences are not good discriminators of special video sequences such as news programs for the following reason. Even though the content of news broadcasts is different each day, the broadcast producer may precode the style and even the timing information of the different news segments. For example, the first 30 seconds may be devoted to international news, the next 45 seconds to sports and the last 30 seconds to weather. In such a case, the timing sequences of two different news broadcasts may be similar. In such a scenario, we can use the above signatures to filter away unrelated video clips. We can then employ more complex content-based features based on individual frames (e.g., the frame based approach we discussed in the last section) or the audio track to distinguish between such newscasts. We revisit this issue of using a set of inexpensive predicates (e.g., the inter-shot timing information) to filter input to expensive predicates (e.g., frame based comparisons),

in Chapter 7.

- **Why does this scheme differ from prior work?** Many video content-retrieval systems (such as CMU's Informedia [WKSS96]) in fact use shot transitions as a mechanism for extracting key frames. That is, they execute a shot transition algorithm (e.g., from [GKA98]) over a video clip and identify where the camera position changes. Then Informedia "aggregates" the frames between the shot transitions and either chooses a key frame or computes a "representative frame," by coalescing the intermediate frames. However, to our knowledge, we are the first to exploit the timing duration between shot transitions as a discriminating feature.

- **How can we attack these signatures?** It is easy to see that time-based features have the advantage that they are resilient to imperceptible visual changes such as illumination shifts, modifications in aspect ratios and format changes. However a pirate may try to modify the "temporal signature" of movies. For instance, the pirate may add random "jitter" to the video clip by repeating a few frames multiple times or by adding a few random frames (e.g., subliminal messages). As we will see later, our signatures are good discriminators even when the pirate makes a few modifications to the temporal signature. However, if the pirate makes substantial modifications to a clip, the temporal sequence may be radically modified and our schemes will fail. We have observed that such modifications in the clip often lead to very poor quality video (e.g., video jitter). Also it is usually difficult to resynchronize the audio track with the video track: voices from the previous set of frames will lead into the next set of frames, and the sound track will not be synchronized with lip movement. We believe that in this scenario, the commercial value of the clip is reduced and content publishers are less concerned about losing revenue (Chapter 1).

6.4 Similarity measure based on shot transitions

Before we discuss how to compute similarity between any two video sequences, we first consider the case when we have two similar video sequences, v and v'. For simplicity, for now we assume that both video sequences are "in phase" and we have only those portions of v and v' that are indeed similar (that is, neither v nor v' have a prequel or sequel that are different). This simplification allows us to ignore issues that arise in video clipping for

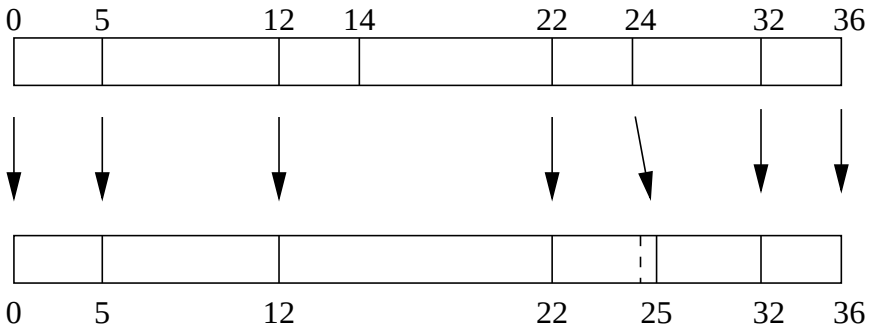


Figure 6.1: Similar video sequences.

now, and to focus on other key problems that arise. Notice that the clipping problem can be solved using some of the chunking techniques we introduced in Chapter 2 (we will soon discuss how).

When we run a shot transition algorithm (e.g., from [GKA98]) on v, we obtain the timing sequence $T(v) = [t_1, t_2, \ldots, t_n]$, where t_i $(1 \leq i \leq n)$ denotes that the time (in seconds) at which the i^{th} shot transition occurs in v. To simplify exposition, we assume all t_i's are positive integers[1] in the interval $\{0 \ldots M\}$. Similarly, we obtain $T(v')$. Ideally, the shot transition algorithm produces the same timing sequences for both v and v'. However, shot transition algorithms are imperfect and prone to the following two errors:

1. **Detection error:** Often the algorithm may detect a spurious transition. This happens when the scene has abrupt lighting changes (e.g., camera flash or an explosion in the scene.) However such misses are not a problem as long as the algorithm makes the same error in both v and v'. A more serious error occurs when the algorithm detects a spurious transition in one sequence and not in the other. For instance, sequence v' may be a "jittered" version of v either because of frame rate conversions, resampling or even format conversions. The shot transition algorithm may then report a transition at the "jitter point" in v' but not in v, because successive frames in v' at the jitter point may be sufficiently different. Similarly, the shot transition algorithm may lose a legitimate shot transition in v' again due to similar reasons. For example,

[1]Even though t_i's will be arbitrary real numbers, video sequences have a finite resolution in terms of frames-per-seconds. Depending on how much resolution we need to differentiate between similar sequences, we can normalize all t_i by multiplying by a large integer and make them integers as well. We will discuss this issue again when we report our experiments.

consider the two video sequences in Figure 6.1. The shot transitions for the 1^{st} sequence occurs at times $0, 5, 12, 14, \ldots, 36$ and at $0, 5, 12, \ldots, 36$ for the 2^{nd} sequence. Notice that the shot transitions at the 0^{th} and 5^{th} seconds match in the two video sequences. However, there is no matching shot transition at the 14^{th} second in the second sequence (i.e., detection error).

2. **Precision error:** Even when the shot transition algorithm detects transitions uniformly across the two sequences, the transitions may be reported with minor differences in the timings. That is, the algorithm detects a transition at time t_i in v, while detecting the corresponding transition at time $t_i + \delta$ for some small δ. For example, in a 30 fps version of a movie, a shot transition can occur only in multiples of $1/30^{th}$ of a second, while in a 25 fps version of the same movie, shot transitions can occur only in multiples of $1/25^{th}$ of a second. A similar problem occurs when the same movie occurs in two different formats. For example, in Figure 6.1, notice that the transition at the 24^{th} second in the first video has a corresponding matching transition at the 25^{th} shot transition.

In addition to the above errors introduced by the shot transition algorithm, recall that a cyber-pirate may tamper with the temporal signature of a video clip. Two common attacks we discussed in Section 6.2 were the following: (1) the pirate adds random "jitter" to the video clip, (2) the pirate repeats a few frames or adds a few random frames. However, observe that the attacks induce precision and detection errors (i.e., transitions are skewed or lost). Based on the above examples, we define v and v' to be similar if the *distance*, $dist(v, v')$, between video clips v and v' is "acceptably" small (we discuss this in our experiments). We now discuss different options for defining the distance between video clips.

6.4.1 Distance measure between video clips

We can define $dist(v, v')$ based on the following intuition.

- **Precision error only:** In case the shot transition algorithm has no detection errors, we have two equi-length timing sequences $T(v) = [t_1, t_2, \ldots, t_n]$ and $T(v') = [t'_1, t'_2, \ldots, t'_n]$. (Recall that we are ignoring video clipping issues for now.) One natural definition of $dist(v, v')$ would be the l_p norm, where the l_p norm is defined to

be $\sum_{i=1}^{n} |t_i - t'_i|^p$, for $p \geq 1$. For instance, the l_1 version of the measure would be $dist(v, v') = \sum_{i=1}^{n} |t_i - t'_i|$. Note that the distance between two identical timing sequences is zero. Also the larger the skew in matching shot transitions (i.e., for each t_i and t'_i), the larger the distance.

- **Detection error only:** In case the shot transition algorithm has no precision errors, we have two timing sequences $T(v)$ and $T(v')$ of potentially different lengths. These sequences will differ only for cases a transition is detected in one sequence, but missed at the same time instant in the other sequence. In this case, a natural distance measure between the two sequences would be the total number of transitions missed across the two sequences. One intuitive way of thinking about the distance measure would be as follows: For $T(v)$ construct its *characteristic* vector χ, a long string of zeros and ones, such that $\chi_i = 1$ iff there was a shot transition at time i in $T(v)$. Similarly, construct[2] χ' for $T(v')$. We can then define $dist(v, v')$ to be the Hamming distance between χ and χ'. For simplicity, we denote $dist(v, v')$ as $T(v) \oplus T(v')$, where $\oplus$ operator computes the Hamming distance between the corresponding characteristic vectors.

  EXAMPLE 6.4.1 Let $T(v) = [1, 4, 6, 10]$ and $T(v') = [1, 4, 10]$. The corresponding characteristic vectors will be $\chi = 1001010001$ and $\chi' = 1001000001$. We see that the Hamming distance between the two characteristic vectors is 1 since the 6^{th} bits in the two vectors are different. We see that the Hamming distance is a natural definition for $dist(v, v')$ since it counts the number of shot transitions that are different. □

While the l_p-based and Hamming based measures are resilient to precision and detection errors respectively, these measures are not resilient to the case we have both errors as we see in the following example.

EXAMPLE 6.4.2 Let $T(v) = [1, 41, 100]$ and $T(v') = [1, 40, 42, 100]$ where v and v' are similar sequences. However (say) due to a frame rate conversion, v' has two shot transitions at the 40^{th} and 42^{nd} seconds instead of the single shot transition at the 41^{st} second in v. Also let $T(w) = [1, 75, 100]$. The key difference betwee w and v is that w has a shot

[2] We are only conceptually constructing the characteristic vector for now.

transition at the 75^{th} second, and v has its transition around the 41^{st} second. Given this difference, v and w are unrelated.

We cannot use the l_p measure to compute similarity between w and v' since the two vectors are not the same length. (We will soon modify the l_p measure to handle this.) Also we see that Hamming distance between v and v' is 3 since the two videos differ in the 40^{th}, 41^{st} and 42^{nd} seconds. However, the Hamming distance between v and w is only 2 since these two videos differ only in the 41^{st} and 75^{th} seconds. This is not desirable since v and w are unrelated, and should have a larger distance value compared to v and v'. Therefore the Hamming distance is also not a desirable measure. □

The above example motivates the following distance measure between two video sequences v and w.

6.4.2 Fuzzy distance measure

We define the following fuzzy distance measure, which is a hybrid between the l_1 measure and Hamming distance notions, for some integer $a \geq 1$.

Definition 6.4.1 (Distance between video sequences)

$$dist_a(v,w) = \left(\sum_{t \in T(v)} \min_{t' \in T(w)} E_a(t,t') \right) + \left(\sum_{t' \in T(w)} \min_{t \in T(v)} E_a(t,t') \right), where$$

$$E_a(t,t') = \begin{cases} |t-t'|/a & \text{if } |t-t'| \leq a \\ 1 & \text{otherwise} \end{cases}$$

□

Conceptually, the above definition combines the l_1 measure and Hamming measure into one distance measure. We will show later in Section 6.7 that this is indeed a good measure. To understand the above definition, we now consider what each sub-term in the above equation represents.

- For transitions at t and t', the $E_a(t,t')$ term computes a "fuzzified" version of Hamming distance. Conceptually, this term handles precision error by allowing the t and t' to be slightly different, by decaying the distance in a linear fashion within distance a. In Figure 6.2 we illustrate this definition.

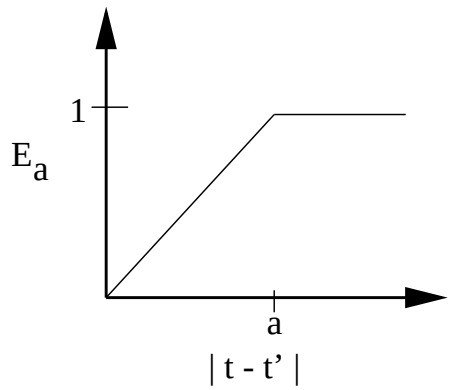


Figure 6.2: Contribution per matching shot transition.

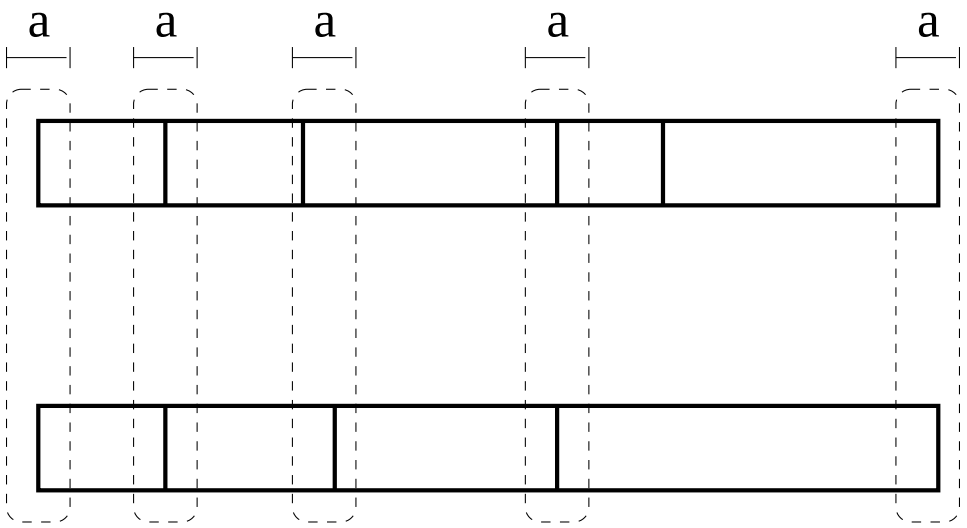


Figure 6.3: Matching shot transitions.

- Consider the term $\min_{t' \in T(w)} E_a(t, t')$. Conceptually, the min term *matches* the timing sequences $T(v)$ and $T(w)$ such that each the transition at t in v is matched with the closest shot transition in w. In Figure 6.3, we show two video sequences along with their matching shot transitions (each matching pair is marked by a dotted box).

  Conceptually, we match each shot transition to its closest corresponding transition (i.e., smallest E_a value) if the transitions are within distance a. We then compute the *contribution* of a shot transition at t based on the corresponding E_a value. That is, if for a shot transition at t the matching shot transition t' is within distance a, t contributes $|t-t'|/a$ towards the distance measure. If no matching transition is found, t contributes 1 towards the distance measure.

- The $\sum_{t \in T(v)} \ldots$ term is analogous to the l_1 measure. That is, it sums up the contribution of each $t \in T(v)$ towards the distance measure. The second term $\sum_{t' \in T(w)} \cdots$ is symmetric to the $\sum_{t \in T(v)} \ldots$ term and sums up contributions of transitions in $T(w)$ towards the distance measure.

We continue Example 6.4.2 now to understand the above definition.

EXAMPLE 6.4.3 We have $T(v) = [1, 41, 100]$ and $T(v') = [1, 40, 42, 100]$. For convenience, we refer to the i^{th} entry in the sequence at $T_i(v)$. For example, $T_2(v) = 40$ because the second entry in $T(v)$ is 40.

We notice that for $a = 2$, we have the following matchings $\{\langle T_1(v), T_1(v')\rangle, \langle T_2(v), T_2(v')\rangle, \langle T_2(v), T_3(v')\rangle, \langle T_3(v), T_4(v')\rangle\}$. That is, the 1^{st} transition in $T(v)$ is matched with the 1^{st} transition in $T(v')$ and so on. Note that both the 2^{nd} and 3^{rd} transitions in $T(v')$ are matched with the 2^{nd} transition in $T(v)$. Given the above matching, we see that

$$dist(v, v') = [(1-1) + \frac{|40-41|}{2} + \frac{|42-41|}{2} + (100-100)] + [(1-1) + \frac{|41-40|}{2} + (100-100)] = \frac{3}{2}$$

On the other hand, notice that $T(v, w) = |75-41|/2 + |41-71|/2 = 34$. That is, the measure identifies v and v' to be "close" and v and w to be "distant" and thereby unrelated. □

6.5 Implementing the FIND operation

We now consider how to identify registered video sequences similar to a given query video. Specifically, we discuss how to pre-process timing sequences of registered video sequences

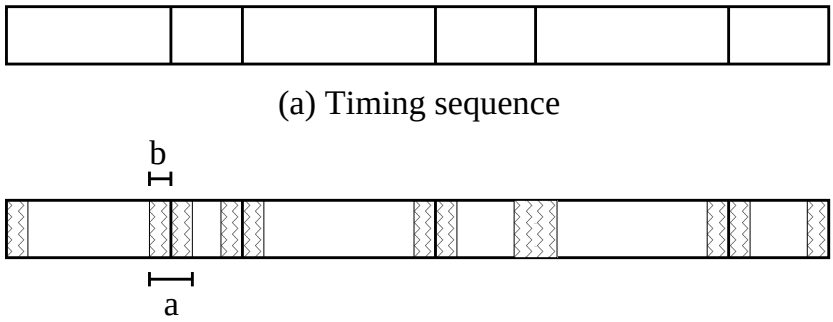


Figure 6.4: Fuzzified timing sequence.

so that given a query video's timing sequence, we can identify all similar registered timing sequences. Also to simplify exposition, we assume that a is odd. Recall that all time instants t_i are positive integers in the interval $\{0, 1, \ldots, M\}$.

Definition 6.5.1 (**Hamming embedding**) Assume $a = 2b + 1$. For a given timing sequence $T = [t_1, t_2, \ldots, t_n]$, we construct $H(T)$ of length M as follows. Initially fill $H(T)$ with all zeros. Then for each $t_i \in T$, set bits in interval $[\max\{t_i - t_1 - b, 0\}, \min\{t_i - t_1 + b, M\}]$ with ones. □

We show in Figure 6.4 a timing sequence and its corresponding *fuzzified* timing sequence as computed by the above Hamming embedding. Conceptually, in this procedure we replace each shot transition in a timing sequence by several shot transitions for a distance b to the left and right of the transition, i.e., all shot transitions are fuzzified.

In Figure 6.5 we focus on one pair of matching shot transitions (marked by dark vertical bars) in two videos. The lighter vertical bars correspond to shot transitions introduced by the above fuzzification procedure. Assume for now, there are no other shot transitions that intersect with the fuzzified shot transitions. Observe that if the two original shot transitions are distance d away, notice that the Hamming distance between the pair of fuzzified transitions is $2 * d$. This is because the two sequences differ in the left-most d fuzzy transitions in the first sequence, and in the right-most d fuzzy transitions in the second sequence. Also observe that if $d > a$, the Hamming distance will be $2 * a$. Based on the above observation, we can show the following lemma ($|t - t'|$ corresponds to d in the

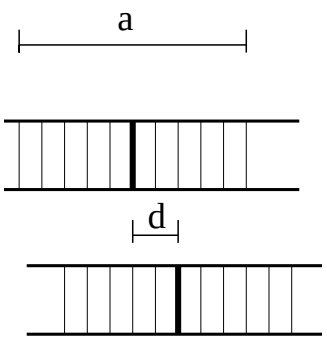


Figure 6.5: Hamming embedding of timing sequence.

discussion).

Lemma 6.5.1 *For shot transitions at t and t' in timing sequences v and w, and the corresponding fuzzified transitions $H(t)$ and $H(t')$ we have*

$$\frac{H(t) \oplus H(t')}{2} = |t - t'|, \text{ when } |t - t'| \leq a \quad (6.5.1)$$

$$= a, \text{ when } |t - t'| > a \quad (6.5.2)$$

□

Notice that $(H(t) \oplus H(t'))/2$ has the same form as $E_a(t, t')$ from the previous section. That is, we see that the contribution (E_a) of two matching shot transitions towards the distance measure, is twice the Hamming distance between the fuzzified shot transitions. Based on this crucial observation, we see the following theorem for $dist(v, w)$, for any videos v and w.

Theorem 6.5.1 *If the minimum separation assumption (below) holds, we have*

$$H(T(v)) \oplus H(T(w)) = a * dist(v, w).$$

□

Assumption 1 *Minimum separation: The minimum time between two transitions is at least $2a$. That is, for any $t, t' \in T(v)$ we have $|t - t'| \geq 2a$.* □

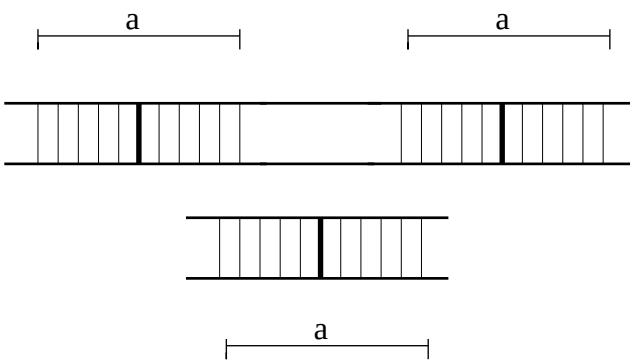


Figure 6.6: Overlapping fuzzified timing sequences.

Conceptually, this assumption ensures that each shot transition has at most one matching shot transition within distance a in the other timing sequence. For example, in Figure 6.6 we see that two sets of fuzzified shot transitions in the first sequence overlap with the fuzzified transition in the second sequence. Therefore, the Hamming distance for the two timing sequences will be smaller than $dist(v_i, w_j)$. However, the minimum separation assumption is reasonable for the following reason. The precision error is typically on the order of fractions of a second while the average inter-shot transition is typically on the order of seconds.

The following example illustrates the above theorem when the minimum separation assumption holds.

EXAMPLE 6.5.1 Consider $T(v) = [1, 5, 10]$ and $T(v') = [1, 6, 10]$. The corresponding characteristic vectors are $\chi = 1000100001$ and $\chi' = 1000010001$. We see that for $a = 3$, $dist(v, v') = |5 - 6|/3 + |6 - 5|/3 = 2/3$.

We have $H(T(v)) = 1101110011$ and $H(T(v')) = 1100111011$. The corresponding Hamming distance is 2. So we have $[H(T(v)) \oplus H(T(v'))]/a = dist(v, v')$. □

So far, we had assumed we explicitly computed the characteristic vector and the corresponding H vector. This would be expensive especially when the length of the timing sequence is large. However, we use the above mapping only to give the reader a conceptual understanding of our scheme. In practice, we can easily simulate such "intersections" using only the two timing sequences. Conceptually, we can "walk" through the two timing sequences, and count the number of positions at which the two sequences differ within window a.

In summary, we showed how to convert timing sequences $T(v)$ and $T(w)$ into an alternate representation, $H(T(v)$ and $H(T(w))$. Then we showed that the Hamming distance between $H(T(v))$ and $H(T(w))$, is equivalent to the $dist(v, w)$ as defined in the previous section. In the next section, we show how to exploit this theorem to efficiently index our registered timing sequences and implement the FIND and FIND-ALL operations efficiently.

6.6 Putting it all together

We summarize how to execute the FIND and FIND-ALL operations. First, we consider how to extend our similarity measure so we can handle video clipping. Then we show how to use a high-dimensional indexing data structure such as a Locality Sensitive Hashing (LSH) structure [IM98, GIM99]. The LSH structure indexes a bit string (i.e., strings of zeroes and ones) representing points in a high-dimensional space. Given a query bit string and some distance threshold m, it returns a list of stored bit strings within Hamming distance m of query bit-string.

6.6.1 Handling video clipping

For each video v, we extract the timing sequence $T(v)$. Since we need to identify partially copied video clips, we then chunk $T(v)$ into shorter timing sequences of k seconds each, similar to our chunking strategies in Chapter 2 as follows. (We discuss in our experiments the impact of k on the distance measure.) The i^{th} chunk of $T(v)$ is the timing sequence from $T_i(v)$ to $T_j(v)$ such that j is the lowest value at which $T_j(v) - T_i(v) \geq k$.

EXAMPLE 6.6.1 Say for video clip v, $T(v) = [1, 7, 13, 23, 37]$. Also let $k = 10$ seconds. The first chunk will be $[1, 7, 13]$ since the third entry in $T(v)$ is the lowest value at which the length of the timing chunk exceeds 10 seconds. Similarly, the second and third chunks will be $[7, 13, 23]$ and $[13, 23]$ respectively. □

As in our text CDS case, we then define the similarity between two videos to be the number of similar (according to our definition in the last section) timing chunks shared by the two videos.

6.6.2 Executing FIND and FIND-ALL operations

First, we pre-process all registered videos as follows. Compute the timing sequence from each video. Then chunk each video into timing segments of at least k seconds each, as discussed earlier. For each timing chunk, compute the fuzzified timing chunk. Add all fuzzified timing chunks (i.e., bit strings) into LSH structure along with the identifier (ID) of the video sequence.

Given a query video, compute its timing sequence. Then chunk the timing sequence into timing segments of at least k seconds each. For each timing chunk, probe the above LSH index structure and identify all videos with some closely matching timing chunk. That is, find timing chunks such that the distance between query bit-string and registered bit-string is less than threshold m. For each matching timing chunk for a registered video, increment a counter to keep track of the number of matching timing chunks. Registered videos with more than C (i.e., threshold) timing chunks in common, are then reported as videos similar to the query video. (In our experiments, we soon discuss the impact of m, k and C on our similarity measures.)

Given a set of query videos and a set of registered videos, we now consider how to implement the FIND-ALL operation. That is, we need to find all pairs v, r such that v is a query video, r is a registered video and they share more than C timing chunks in common. Notice that this is an instance of an *iceberg* query. That is, we cluster all fuzzified timing chunks. We then use our techniques from Chapter 3 to implement the FIND-ALL operation for videos.

6.7 Experiments

We designed our experiments in this section to evaluate (1) our feature extraction technique, (2) our similarity measure, and (3) our indexing strategies.

6.7.1 Data set

We used the following dataset for our experiments.

- We downloaded 2000 MPEG (MPEG1 and MPEG2) clips from the Internet primarily from news sites and from web sites that maintain clips of music videos and movie trailers. These clips were typically between 2 minutes and 5 minutes in length.

- We downloaded 5 movie trailers for "Star Wars: Episode 1" from a popular web site for movie fans [Cou99]. These clips were different versions of the same trailer. For instance, the official trailer of the movie has `www.starwars.com` tagged onto each frame. Also it starts with the Universal pictures logo after a pause for a few seconds. However, a slightly different video was shown in "Entertainment Tonite," a popular television program. This video is identical to the official trailer except it does not start with the Universal logo. Also the "E.T." logo replaced the Star Wars logo on this clip. The third video was taped by a fan using a video camera in a movie theater that was showing the trailer. This clip was more "jittery" since a person was holding the camera in his hand. The fourth and fifth videos were identical to the official trailer, except they were in AVI and Quicktime formats. We converted these into MPEG1 using `avi2mpeg`, Windows based software.

For our experiments, we assume that all the above videos are unrelated to all other videos, except for the 6 Star Wars videos. We examined some of these videos, and believe this is a valid assumption.

To evaluate how frame-rate conversions affected our data, we converted the above clips to lower frame rates as well (e.g., from 30 fps to 25 and 15 fps). In total, we had 5000 video clips in our collection some of which were related (i.e., identical videos at different frame rates or different formats), and the rest unrelated to each other. Notice that since we carefully designed our dataset, we know which clips are similar to others, and which clips are unrelated.

6.7.2 Computing shot transitions

Recall that one of our goals in building the CDS is to extract features as fast as possible. For this reason, we adopt the following two-pronged approach for detecting shot transitions:

1. **MPEG1:** We use an optimized version of the algorithm proposed in [YL95] to compute shot transitions for video sequences in MPEG1 format, faster than real-time. That is, we extract the shot transitions faster than it takes us to play the movie on a viewer. We chose this particular algorithm because of the quality of its performance compared to other similar algorithms [GKA98] and also due to its speed.

2. **non-MPEG1:** If the video sequence is not in MPEG1 format (e.g., the sequence is in another format such as MPEG2 or RealVideo), we adopt a computationally more

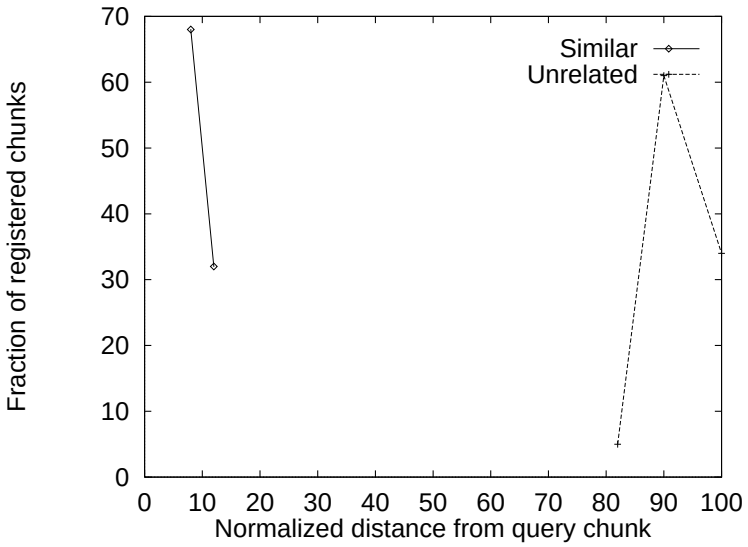


Figure 6.7: Histogram of distances ($k = 45$ secs).

expensive algorithm from [IIS99]. This process is slower since it works by uncompressing the video clip into a sequence of video frames, rather than working directly on the MPEG1 frames as in [YL95].

6.7.3 Quality of similarity measure

In our earlier discussions, our similarity measures were parameterized by k (minimum length of timing chunk), a (fuzzification window), m (threshold for distance between timing chunks), and C (threshold for number of shared timing chunks). We now discuss the impact of these parameters on how well our similarity measures perform on our data set.

We empirically observed that different versions of a video clip (i.e., after format or frame rate conversions), typically have precision error less than 50 msecs. That is, matching shot transitions are skewed up to 50 msecs to the left and right of a shot transition. Hence we observed that a good value for a is 101 msecs.

Before we discuss how to choose C and m, consider the following experiment that shows the impact of k on the similarity measure. First, consider the case where we set $k = 45$ and compute the first timing chunk for all 5000 videos. For each such timing chunk v, we compute $S(v)$, the set of timing chunks we know (from our data set) to be similar to v (i.e.,

same chunk, but at different frame rate or converted from a different format). Similarly, we compute $U(v)$, the set of unrelated timing chunks.

In Figure 6.7 we show a "typical" histogram that shows for a randomly chosen query chunk v, the distribution of the query chunk's distance to chunks in $S(v)$ and $R(v)$. The plot for "Similar" indicates the fraction of timing chunks similar ($S(v)$) to query chunk v as a function of the corresponding distance. (The distances are normalized to 100 for comparison.) That is, nearly 68% of chunks that are similar to the query chunk are within a normalized distance of 8 from the query chunk. And the other 32% are within a distance of 12 from the query chunk. That is, all similar timing chunks are within distance 12 from query chunk. We term this distance to be the *similar threshold* distance. Similarly for "Unrelated." That is, all unrelated timing chunks are at least distance 82 from query chunk. We term this distance to be the *unrelated threshold* distance. The two distances indicate good values of m. If we choose $12 < m < 82$ (where m is normalized), we can separate chunks that are similar and unrelated to a given query chunk.

Consider the following measure. For each query timing chunk, we define the *separation* to be the ratio between the unrelated threshold and similar threshold distances, Intuitively, when the separation is greater than one, we can always find a value for m such that there are no false positive and false negative errors. In Figure 6.8, we report the average and minimum separations as k is varied. Notice that as the length of k increases, both ratios increase. For $k \leq 25$, the minimum separation is less than one. That is, the similar and unrelated chunk portions of the histogram (Figure 6.7) intersect. In this case, we will need to choose m based on how many false positives and false negatives we can tolerate (as in Chapter 2).

From our experiments, we observe that k and C are inversely related. For instance, we observed that when $k = 45$ seconds, $C = 1$ is a good value. That is, when two videos have even one timing chunk with at least 45 seconds of overlap, they are similar. However, for smaller k (e.g., $k = 25$ seconds), C should be over 10. That is, the videos should share at least 10 timing chunks, when the chunks have only a minimum of 25 seconds overlap. This relationship follows the same intuition as in Chapter 2, when we noticed the inverse relationship between length of word chunks to the number of false positives. For our application, we believe $k = 25$ and $C = 10$ are good values.

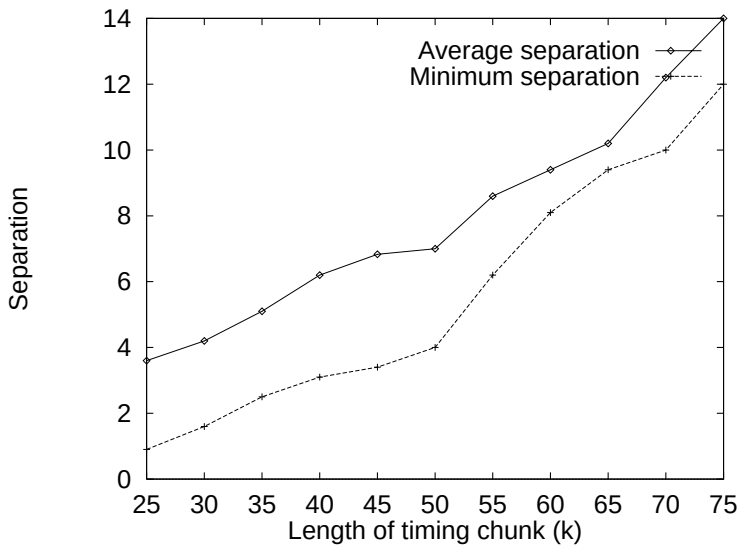


Figure 6.8: Separation as k varies.

6.7.4 Timing results

We mentioned earlier that it is crucial for the whole procedure from crawling and feature extraction through indexing to be fast. We now show some timing results (Table 6.1) to give the reader a flavor for which phases are the bottlenecks in the overall procedure. We ran these experiments on a SUN UltraSparc/II running SunOS 5.1 with dual processors, 256 MB RAM and a 100 Mbps link to the Stanford backbone.

In the registration phase, we download clips, compute shot transitions and add the fuzzified timing sequence to the LSH indices. The time to download clips varies primarily due to variance in the traffic at remote web sites and their uplink to the Internet (we ran the crawler on a Saturday nite, when we expected the network to be uncongested). The time to compute shot transitions varies dramatically as well depending on the video format. As we mentioned earlier, in case of MPEG1 clips, our shot transition algorithm uses the compressed format to compute shot transitions. This process takes roughly 10 seconds for a 2 minute clip. In case of non-MPEG1 format, recall that we use `mpeg2decode` to uncompress the data and provide us a sequence of JPEG images. While our algorithm to compute shot transitions is very fast, the underlying software to uncompress the video clip is very slow. In the future, we plan to develop shot transition algorithms for various

Phase	Time
Downloading	10 secs – 1 min
Shot transitions	10 secs – 25 mins
Computing timing sequence	< 250 msecs
Adding to LSH indices	< 100 msecs

Table 6.1: Timing results.

specific formats so we do not have to uncompress the underlying video data. Computing the fuzzified timing sequence and adding to LSH indices is very fast (typically less than 200 msecs in our implementation).

In the query phase, we face the cost of downloading the clip, computing shot transitions and probing the LSH indices for each of query's timing chunks. Again here the cost is dominated by the cost of downloading the clip and the index probing is typically fast (< 200 msecs in our implementation).

Notice that the main bottleneck currently is network latency. We can address this problem by downloading several video streams in parallel, thereby keeping the video comparator busy on the average. Also notice that our current feature extraction algorithms are another bottleneck. We believe that when shot transition algorithms are developed for all encoding formats, similar to MPEG1, the feature extraction phase will also be faster.

6.8 Conclusion

In this chapter, we discussed how to build a video comparator. The main contributions of this chapter are four-fold. Firstly, we discussed a variety of feature extraction techniques and identified one technique based on timing signatures that is resilient to a variety of attacks from a cyber-pirate. Secondly, we proposed a natural similarity measure to compare timing sequences so extracted. Thirdly, we proposed how to embed our measure into Hamming space. Finally, we evaluated the "discriminatory" characteristic of the features we extract, the quality of our similarity measure and finally the efficiency of our approach.

Chapter 7

Approximate Query processing

7.1 Introduction

The CDS should service a variety of user requirements in terms of tolerable expense and accuracy. For example, a singer may want the CDS to find songs similar to his song irrespective of the cost, but with no false negative errors (i.e., no similar songs are missed). On the other hand, Sony Music has a large collection of songs and may tolerate some errors for reduced cost in finding similar songs. In this chapter, we discuss how to build a CDS that can service varied user requirements.

Consider the case where the CDS has a set of media-specific predicates to test object similarity, each with specific cost and error characteristics (as discussed in earlier chapters). We can then build a CDS for a specific cost and error requirement if the CDS has a "matching" predicate with the same cost and error characteristics. However in general it is unlikely we can always find such a matching predicate since the spectrum of user requirements is large. Therefore, we adopt the following approach. As before, the CDS has a set of predicates designed by a domain expert. However the CDS automatically *composes* some of the given predicates in different ways to create new predicates when a user specifies his requirement. For example, if a singer requires the CDS to execute a FIND operation within two hours, the CDS automatically creates a new predicate as follows. The CDS *filters* the input (song pairs) to an expensive predicate with a cheaper but less accurate predicate so that the overall computation takes two hours to execute. Similarly, the CDS can create new predicates with smaller overall errors using several inaccurate predicates by streaming input to these predicates in parallel.

In general, we term predicates with either false positive and false negative errors as *approximate predicates*. Such approximate predicates approximate some *ideal predicate* (e.g., a human) that has zero false positive and false negative errors, but is computationally expensive to execute. Typically each ideal predicate has several approximate predicates each with different expense and accuracy characteristics. We now study the problem of selecting and composing approximate predicates in order to design new *composite predicates* that satisfy a given user requirement. More specifically, our contributions are as follows:

- We present a model for approximate predicates, including their selectivities, costs and errors. We derive formulae for the selectivities, costs and errors for logical combinations of approximate predicates.
- We present optimization strategies for automatically designing composite predicates. For some scenarios, our strategies yield provably optimal plans; for others the strategies are heuristic ones. For some of the heuristic strategies, we develop approximation ratios that bound how far a solution can be from the optimal one. In all scenarios, we discuss the complexity of our strategies.
- We note that our techniques are in fact very general, and can be used to improve performance of SQL queries in *extensible database* systems [CS96]. We discuss how our strategies can be easily incorporated into existing query optimizers for such systems, and evaluate these strategies empirically to show the potential performance gains.

The rest of the chapter is organized as follows. In Section 7.1.1 we present other motivating examples of complex, data-intensive applications where our techniques will be useful. In Section 7.2 we formally characterize predicates. In Section 7.3 we characterize the space of *query plans* [Ull88] we need to consider. In Section 7.4 we propose a brute-force optimizer, followed by more efficient ones in Sections 7.5, 7.6 and 7.7. In Section 7.8 we evaluate our techniques in constructing composite predicates.

7.1.1 Other motivating applications

A wide ranging class of applications are using database systems these days, and are evaluating complex predicates. Such predicates can, for example, compare images in the database to some reference image, can check for containment of points within regions, or can search for certain trading patterns in a stock market database. Because these predicates are often expensive to evaluate, application designers design cheaper approximate predicates to

cut down the number of data elements that must be analyzed by the ideal predicate. For example, to check if a point is contained inside a complex region, we can first check if the point is within the region's bounding rectangle. One expects that most database points will not be in the rectangle, so the ideal containment test need only be run on a much smaller subset of points.

In the QBIC image retrieval system [FSN95], color histogram matching is an important way of computing similarity measures between images. This matching is based on a 256-dimensional color histogram and requires a 256 matrix-vector multiplication. However, QBIC employs a much faster "pre-computation" in 3D space to filter input to the more expensive histogram matching phase. Only images that pass the fast test are given to the histogram test, and only the ones that pass both tests are shown to the end user. This filtering saves substantial computational effort [FSN95].

There are many similar motivating applications in data scrubbing [ME97] and search problems. For instance, approximation algorithms with bounded errors have been developed for many NP-hard problems such as the minimum-cost traveling sales-person problem (TSP) [Aro96], and for approximate searches in high-dimensional spaces [IM98]. Hence if a user can tolerate errors, these approximations and their composite versions can be used as a filter to complex ideal predicates, or to even replace the ideal predicates.

From the above applications, we see that automatically composing predicates to serve specific user requirements is a general problem. Rather than present our techniques for this problem in the specific context of building a CDS, we present our techniques for the general problem of adding approximate predicates to *extensible database systems* [Haas90]. These systems support complex, user-defined predicates and functions so these systems can be customized for different applications. After presenting our techniques in this general setting, we show how these techniques are applicable in building a flexible CDS.

7.2 Characterizing predicates

We now define different types of predicates, and how to characterize them in terms of expense, selectivity, and errors. We distinguish between two kinds of predicates based on how they can be evaluated.

1. **Access predicate:** These predicates select and stream out tuples in a given relation, using some index access method. For instance, consider a database that indexes

images based on their dominant color components. An access predicate to find all images with substantial yellow components can use the index, and stream out the corresponding "yellow" images.

2. **Restriction predicate:** These predicates are directly evaluated on a given tuple, rather than on a relation. For example, consider a predicate to check if a given image (tuple) has a substantial yellow component. We can implement a restriction predicate to compute the color histogram of the image, and check if the yellow component exceeds some threshold.

Let $I = \{I_1, I_2, \ldots, I_m\}$ be a set of ideal predicates. For each I_i, $1 \leq i \leq m$, we have a set of approximate predicates A_i ($= \{ A_{i,j} \}$) that can filter I_i. Let $A = \cup_{i=1}^{m} A_i$.

We now define the important characteristics of restriction predicate $A_{i,j}$ that approximates its corresponding ideal predicate I_i. Let *selectivity* $s_{i,j} = P(A_{i,j})$ be the probability that some given tuple satisfies $A_{i,j}$. It is then expected that for any input stream of t tuples, $t * s_{i,j}$ tuples satisfy $A_{i,j}$. Another important characteristic of $A_{i,j}$ is $e_{i,j}$, the expense of evaluating the predicate for each tuple, expressed in *units-per-tuple* (upts). We quantify $A_{i,j}$'s *false negative* error as $n_{i,j} = P(\neg A_{i,j}|I_i)$, which is the conditional probability that a tuple does not satisfy $A_{i,j}$, given that the tuple satisfies I_i. Similarly we quantify $A_{i,j}$'s *false positive* error as $p_{i,j} = P(A_{i,j}|\neg I_i)$, which is the conditional probability that a tuple satisfies $A_{i,j}$, given that the tuple does not satisfy I_i.

Access predicates have characteristics similar to restriction predicates. We define for access predicate $A_{i,j}$ the expense $e_{i,j}$ to be the expense of finding and streaming out tuples satisfying the predicate, normalized with respect to the number of tuples in the database. For instance, if an index on a database with 2000 images charges 1000 units to search and retrieve images with substantial yellow component, we define $e_{i,j} = 1000/2000 = 0.5$ upts. We define the selectivity $P(A_{i,j}) = s_{i,j}$ to be the fraction of output tuples to the total number of tuples in the relation. Similarly, we define $p_{i,j}$ to be $P(A_{i,j}|\neg I_i)$ and $n_{i,j}$ to be $P(\neg A_{i,j}|I_i)$.

Ideal predicates have expense and selectivity characteristics. In particular, we define s_i to be $P(I_i)$, and the expense of evaluating the predicate e_i. By definition, ideal predicates do not have false positive or negative errors.

EXAMPLE 7.2.1 Consider a relation with 1000 tuples. Out of all the tuples in the relation, 10 tuples satisfy ideal predicate I_1, i.e., $s_1 = 10/1000 = 0.01$. Say the expense of

Predicate	Type	Expense	Selectivity	False +ve	False -ve
I_1	Restriction	1000	0.1	0	0
$A_{1,1}$	Access	10	0.9	0.25	0.1
$A_{1,2}$	Access	50	0.2	0.1	0.1
$A_{1,3}$	Restriction	100	0.3	0.1	0.2
I_2	Restriction	2000	0.2	0	0
$A_{2,1}$	Restriction	500	0.5	0.2	0.1

Table 7.1: Characteristics of example predicates.

running I_1 on one tuple is 10,000 units.

Consider restriction predicate $A_{1,1}$ which has a per-tuple expense of 50 upts. Out of all the tuples in the relation, 107 tuples satisfy $A_{1,1}$. Out of these, 8 tuples also satisfy I_1. We can compute $s_{1,1} = 107/1000 = 0.107$, $p_{1,1} = P(A_{1,1}|\neg I_1) = (107 - 8)/(1000 - 10) = 0.1$, and $n_{1,1} = P(\neg A_{1,1}|I_1) = (10 - 8)/10 = 0.2$.

Next consider access predicate $A_{1,2}$ which costs 5000 units to execute using an index: 50 tuples satisfy $A_{1,2}$. Out of these, 9 tuples also satisfy I_1. We can compute $e_{i,j} = 5000/1000 = 5$, $s_{i,j} = 50/1000 = 0.05$, $p_{1,2} = P(A_{1,1}|\neg I_i) = (50 - 9)/(1000 - 10) = 0.04$, and $n_{1,2} = (10 - 9)/10 = 0.1$. □

7.3 Space of query plans

We now illustrate the space of plans possible in an extensible database that supports ideal and approximate predicates. Consider an example database with ten tuples. Consider the catalog information in Table 7.1, with meta-data about expensive predicates I_1 and I_2 along with their approximate predicates $A_{1,1}$, $A_{1,2}$, $A_{1,3}$ and $A_{2,1}$. The values listed in the table are "made-up" so as to make exposition clear, and should not be interpreted in any special way.

We use a standard *query tree* representation [Ull88] to show the logical query plans for our examples in this section. The tree has relations at its leaves; selections, joins, projections and cross-products are placed at the tree's internal nodes [Ull88]. In some cases, the trees may be annotated with other implementation details such as indices selected and interesting orders, but we will not use such annotations in our examples below for simplicity.

EXAMPLE 7.3.1 Consider the following simple `Select` query issued by the user: Find

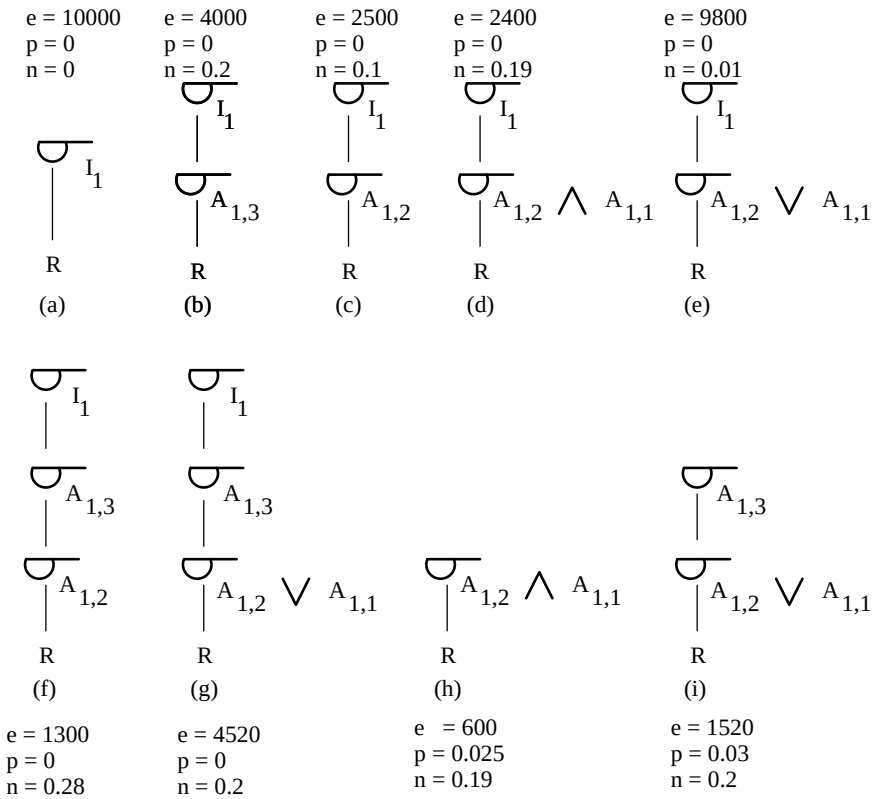


Figure 7.1: Some query plans for Example 7.3.1.

all tuples from table R satisfying predicate I_1. We present in Figure 7.1 query trees for alternative queries along with their expenses (e) in upts, and overall false positive (p) and negative (n) errors. (In the next section, we show how to compute or estimate these values for any given query plan.)

Plan (a) is the tree for the user query that applies predicate I_1 on all tuples in table R. Plans (b) and (c) are examples of filtering input to I_1 by checking $A_{1,3}$ or $A_{1,2}$ on tuples in R before checking for I_1. Plans (d) and (e) show how approximate predicates can be *composed* using conjuncts ($ANDs$ or "$\wedge$") and disjuncts (ORs or "$\vee$") to filter input to I_1. Observe that by composing approximate predicates, we managed to (1) reduce the execution expense from 2500 in Plan (c) to 2400 in Plan (d), and (2) reduce the false negative error from 0.1 in Plan (c) to 0.01 in Plan (e). Plans (f) and (g) show how a restriction predicate such as $A_{1,3}$ can be "sequenced" (**SQN** or $\rightarrow$) on top of composed access predicates, to reduce the expense of Plans (c) and (e), at the cost of increased n errors. The **SQN** operator is similar to AND operator in terms of errors and selectivity, but differs in terms of expense. For instance, $A_{1,1} \wedge A_{1,3}$ will have a higher expense than $A_{1,1} \rightarrow A_{1,3}$ since in the former, $A_{1,3}$ is applied on all tuples while in the latter it is applied only to tuples that satisfy $A_{1,1}$. Plans (h) and (i) are similar to Plans (d) and (g) except they do not check (expensive) I_1 on the tuples: these plans have lower expected expense at the cost of potential false positives ($p > 0$).

The set of query trees we present in Figure 7.1 is clearly not complete, but gives the reader a flavor for the space of plans for filtering simple predicates. □

EXAMPLE 7.3.2 Consider a `JOIN` query that performs an equi-join between two relations R_1 and R_2 as shown in Plan (a) of Figure 7.2.

We present some possible query trees in Figure 7.2 along with expenses and false positive and negative errors incurred in executing each plan.

Plan (b) reduces the execution expense by filtering I_1 with $A_{1,2}$. Plan (c) is a more complex alternative query that filters tuples in R_1 and R_2 using some approximate predicates and finally performing I_1 after the equi-join. Note that this plan never checks tuples with I_2 and therefore $p > 0$. Clearly we can see that this set of alternative queries is not complete: in fact any of the alternative queries in Example 7.3.1 can be used to filter I_1, and similarly for I_2. Also we can push ideal and approximate predicates to several places in the query tree, both with respect to the equality join predicate, as well as to each other (as in expensive predicate placement [CS96]. The example set of alternative queries we present

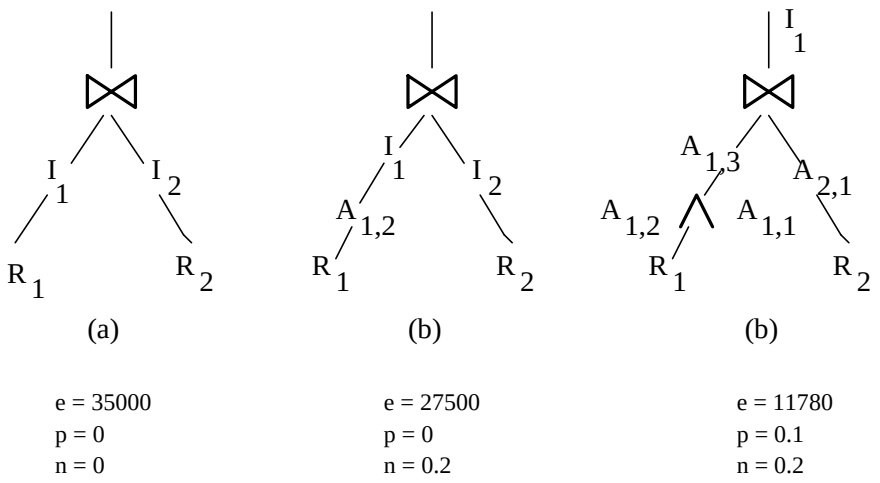


Figure 7.2: Some query plans for Example 7.3.2.

however do illustrate the increased set of plans to be considered by a query optimizer. □

From the examples we see that a query optimizer should consider alternative queries where approximate predicates are composed using operators such as $\vee$, $\wedge$, $\neg$ and $\rightarrow$. In our execution model, we define *well-formed* query plans to be ones where the following compositions *cannot* be executed:

1. **Negated access predicates:** These correspond to accessing an index to find tuples that satisfy a given predicate, and returning the rest of the tuples in the relation.

2. **Sequenced access predicates:** These correspond to accessing an index to find tuples that satisfy a given predicate, sequencing these tuples to another index and computing a smaller set of tuples that satisfy both predicates.

Note that the second assumption does not preclude "index intersection;" our execution model does allow tuples to be retrieved from two sets of indices independently, and subsequently intersected using the $\wedge$ operator.

A query optimizer that needs to consider the space of well-formed plans becomes more complex than traditional query optimizers due to the increased number of plans. However we also see from the same examples that the potential payoffs are huge (we show this using experiments in Section 7.8). Hence we believe the increased complexity in building a query optimizer is a worthwhile price to pay for the potential payoffs in query execution.

7.3.1 Minimization measures

In this subsection we define measures to evaluate query plans for alternative queries, so an optimizer can choose the best plan to execute the given user query. In classical query optimization, the goal is to choose query plans, for a given query, with minimal query execution expense. In extensible databases with approximate predicates, two natural measures to minimize when evaluating different query plans are:

1. **Expense (*MIN-EXP* measure):** This measure selects the query plan with the least execution cost, irrespective of the errors, among the possible alternative queries and their physical implementations. This measure is useful when approximate predicates, that make no false negative errors, stream candidate tuples to be checked by the ideal predicate. In this case, all query plans yield correct results, so cost is the way to compare plans. This measure could also be useful in other cases, for example, if we know that all approximate predicates have acceptably low false negative errors, or if ideal predicates are *replaced* by approximate ones with acceptably low false positive rates. In these cases, errors are assumed to be low enough, and we can select plans based on cost only.

2. **Expense subject to (p, n) constraint ((p, n)-*MIN-EXP* measure):** In many applications, the user would like to control the quality of results returned by specifying acceptable bounds for false positive and negative errors. In such scenarios, the minimization function is expense subject to the constraint that the query plan has error estimates tolerable to the user.

Of course, *MIN-EXP* is one instance of (p, n)-*MIN-EXP* with p and n set to infinity, but we will see that we can construct more efficient query optimizers for the *MIN-EXP* measure than for the (p, n)-*MIN-EXP* measure; hence we retain *MIN-EXP* as a minimization measure in its own right.

7.4 General query optimization

A query optimizer that supports approximate predicates has to choose from the space of plans illustrated in the previous section. There are a variety of approaches for this. We now present the approach we advocate in this chapter, and defer a discussion of its advantages and drawbacks until after we explain the scheme. Our approach incorporates

Algorithm 7.4.1 *NaiveWrapper for* (p, n)-MIN-EXP *measure*
Inp A: Approximate predicates; $[a_p, a_n]$: Acceptable errors
Procedure
[0] *BestPlan* := Dummy plan with infinite cost
[1] For each $U_1 \subseteq [A_1 \cup \{I_1\}]$, $U_2 \subseteq [A_2 \cup \{I_2\}]$, $U_3 \subseteq [A_3 \cup \{I_3\}]$
[2] For each composition C_1, C_2, C_3 of predicates in U_1, U_2 and U_3 with $\vee$, $\wedge$, and $\neg$ such that the compositions are *well-formed*
[3] Construct extended query Q as $\sigma_{C_1 \wedge C_2 \wedge C_3}(R_1 \times R_2)$.
[4] If false positive and negative errors of Q are less than a_p and a_n
[5] Q' := Q optimized using TQO
[6] If *cost*(Q') < *cost*(*BestPlan*)
[7] *BestPlan* := Q'
[8] Return *BestPlan*.

Figure 7.3: Naively wrapping any query optimizer for (p, n)-*MIN-EXP* measure.

a *traditional query optimizer* (TQO), as a component. Given a logical query, the TQO performs traditional query optimization tasks such as plan enumeration, evaluating join orderings, index selections and predicate placement [Ull88, CS96]. Current TQOs of course do not understand the special semantics of approximate predicates. If the TQO gets a query with approximate predicates, it treats these predicates as any other user-defined predicate. The TQO returns the best physical plan for implementing the given query, along with its estimate of the cost of executing that plan.

With our approach, we build a *wrapper* that understands approximate predicates, around the TQO. The wrapper is given a user query containing only ideal predicates, and information on the available approximate predicates. The wrapper *extends* the user query by composing some subset of approximate and ideal predicates into the query, so that the extended query has errors tolerable to the user. The wrapper then feeds this extended query to the TQO, which performs its tasks and returns the cost of executing the optimized version of the extended query. The wrapper iterates through alternative extended queries, and then chooses the alternative query whose optimized version costs the minimum to execute.

We now study this wrapper approach in more detail. The wrapper we present here is clearly not efficient, and hence we call it the *NaiveWrapper*. More efficient versions will be presented in later sections. To keep our discussion brief, we focus on the optimization of the query $[\sigma_{I_1}(R_1)] \bowtie_{I_3} [\sigma_{I_2}(R_2)]$, which is a join of two relations R_1 and R_2 with ideal predicates

I_1 and I_2 to be applied on R_1 and R_2, and I_3 being the join predicate. (Generalizing to more complex queries is straightforward.)

Figure 7.3 shows pseudo-code for the *NaiveWrapper* for this class of join queries. The wrapper first considers all subsets of predicates in $A_i \cup \{I_i\}$, $1 \leq i \leq 3$, (Step [1]), and constructs *alternative* extended queries using $\wedge, \vee, \neg$ (Steps [2], [3]). Next, the wrapper computes the expected false and positive errors for each alternative query; the techniques for this will be covered in Sections 7.5 and 7.6. If the alternative query so produced has a tolerable error (Step [4]), it is handed to the TQO (Step [5]). (If we are using a simple MIN-EXP measure, then all such queries are handed to the optimizer.) The optimizer computes the cost of each alternative query, and the wrapper selects the alternative query with the minimum overall cost of execution.

Notice that the TQO may rearrange the predicates in order to reduce costs. For example, if C_1 is $A_{1,1} \wedge A_{1,2} \wedge I_1$, the optimizer may decide to execute them in some sequence (i.e., introducing the **SQN** operator into the tree). It is important to note that such restructuring does not change the errors of an alternative query. Thus, a query that was deemed acceptable in Step [4] will continue to be acceptable after the optimizer restructures it to reduce costs. This property makes it possible to cleanly separate the wrapper from the optimizer, and we use this property again in the wrappers that we present in the following sections.

The naive wrapper in Figure 7.3 exhaustively enumerates all possible alternative queries, and hence is not practical: the number of Boolean functions on n variables is 2^{2^n} when these variables are composed using $\wedge$, $\vee$ and $\neg$. So, given $|A|$ approximate predicates, the number of alternative queries produced in Steps [1] and [2] is $O(2^{2^{|A|}})$. As we mentioned earlier, in later sections we present much more efficient wrappers, at least for certain classes of queries.

The main advantage of the wrapper approach is its modularity. One can build upon existing optimizers, that codify decades of experience. Thus, to optimize queries with approximate predicates, we do not have to re-invent well known techniques for access path selections, join ordering, hash joins, and so on.

On the other hand, modularity may be a *potential* problem because current TQOs assume that predicates are uncorrelated while making optimization decisions, especially during predicate placement [CS96]. Clearly this assumption is not valid in our case since approximate predicates are correlated with the ideal predicate, and may be correlated with each other.

Of course, we can still use TQOs even if predicates are correlated, except that the resulting plans may be sub-optimal. However, in our experiments (Section 7.8), we observed that incorporating approximate predicates with the wrapping approach leads to significant performance improvements, despite using sub-optimal TQOs. Notice that with the wrapper approach, predicate dependencies are still handled correctly when creating alternative queries and when estimating their false positive and negative errors. Also when new TQOs are developed to correctly handle predicate correlations by future research in traditional query optimization, we can easily incorporate them immediately using our approach.

The alternative to wrappers involves the tight coupling of the enumeration of alternative plans with the optimization phase, so that alternative queries can be automatically pruned when their costs exceed the cost of another candidate alternative query. This may lead to a more efficient optimization phase, but involves modifying an existing optimizer significantly, so that error computations are incorporated into the plan evaluation process. Tight integration does not solve the predicate-interdependence issue. Thus, we would still produce sub-optimal plans, unless the optimizer is also modified to take correlations into account, which has not yet been addressed by research in query optimization.

In summary, with the wrapper approach we can immediately incorporate approximate predicates into any current query optimizer that supports user-defined predicates. For this modularity, we pay the penalty of inefficient query optimizers that do not tightly couple alternative query generation with cost-based optimization. Also, the underlying query optimizer may produce sub-optimal physical plans due to assumptions of predicate independence. However, we have observed experimentally (Section 7.8) that the execution time for queries drops dramatically when we incorporate approximate predicates to filter expensive predicates, despite sub-optimal physical plans.

In the next few sections, we consider how to improve the alternative query generation process, so we do not evaluate a doubly exponential number of alternative queries. In Section 7.5 we focus on the *MIN-EXP* measure and simple `Select` queries. In Section 7.6 we extend the ideas from Section 7.5, and develop an efficient wrapper for SPJ queries under the *MIN-EXP* measure. We subsequently show in Section 7.7 why optimizing for the (p, n)-*MIN-EXP* measure is hard, and then present heuristics for the (p, n)-*MIN-EXP* measure (based on our provably good wrappers for *MIN-EXP* measure).

7.5 Optimizing `Select` queries for *MIN-EXP* measure

In this section we consider how to build a good filter for a simple `Select` query $\sigma_{I_i}(R_1)$ for the *MIN-EXP* measure. To do this, we first need to model how approximate predicates affect each other, i.e., what is the value of an approximate predicate checking a tuple that has already been checked by another approximate predicate. In Section 7.5.1 we propose two models of common predicate dependencies. We then show how to construct good filters in Section 7.5.3.

7.5.1 Modeling predicate dependencies

While there are many possible ways in which predicates can depend on each other, we now consider two cases we have found common among approximate predicates in SCAM and QBIC.

1. **Local Independence (LI):** We assume that all predicates in A_i are pairwise independent but dependent on I_i. That is, $P(\wedge_{A_{i,j} \in A_i} A_{i,j}) = \Pi_{A_{i,j} \in A_i} P(A_{i,j})$. This is a common assumption in extensible and relational databases. This models approximate predicates that consider different attributes of incoming tuples and therefore filter tuples independent of each other.

2. **Local Conditional Independence (LCI):** We assume all predicates in A_i are pairwise independent conditionally on I_i. That is, $P(\wedge_{A_{i,j} \in A_i} A_{i,j} | I_i) = \Pi_{A_{i,j} \in A_i} P(A_{i,j} | I_i)$. This assumption is strictly weaker than LI: if LI holds, LCI also holds, but the converse is not true. Under LCI, the selectivities of approximate predicates are not independent. For example, if we sequence $A_{1,1} \rightarrow A_{1,2}$, the selectivity of this filter is *not* $s_{1,1} * s_{1,2}$. This may be, for instance, because both $A_{1,1}$ and $A_{1,2}$ are approximating (using different techniques) the ideal predicate. So, if $A_{1,1}$ has already detected a document to be a potential copy in SCAM (using say sentence chunking from Chapter 2), that document is much more likely to be found to be a potential copy by $A_{1,2}$ (which may use the RFM measure from Chapter 2). However, under LCI we assume that approximate predicates make positive and negative errors independent of each other for any incoming tuple. That is, the probability that $A_{1,2}$ incorrectly identifies a document to be a copy does not depend on whether $A_{1,1}$ earlier correctly or incorrectly identified it as a potential copy. This is because the predicates are using different mechanisms that may fail in unrelated ways.

Char.	NOT	SQN	AND	OR
$e_{i,new}$	$e_{i,j}$	$\sum_{l=1}^{k}(e_{i,j_l} * \Pi_{q=1}^{l-1} s_{i,j_q})$	$\sum_{l=1}^{k} e_{i,j_l}$	$\sum_{l=1}^{k} e_{i,j_l}$
$s_{i,new}$	$1 - s_{i,j}$	$\pi_{l=1}^{k} s_{i,j_l}$	$\Pi_{l=1}^{k} s_{i,j_l}$	$1 - \Pi_{l=1}^{k}(1 - s_{i,j_l})$
$n_{i,new}$	$1 - n_{i,j}$	$1 - \Pi_{l=1}^{k}(1 - n_{i,j_l})$	$1 - \Pi_{l=1}^{k}(1 - n_{i,j_l})$	$\Pi_{l=1}^{k} n_{i,j_l}$
$p_{i,new}$	$1 - p_{i,j}$	$\Pi_{l=1}^{k} p_{i,j_l}$	$\Pi_{l=1}^{k} p_{i,j_l}$	$1 - \Pi_{l=1}^{k}(1 - p_{i,j_l})$

Table 7.2: Characteristics of filters constructed with LI approximate predicates.

Chars.	NOT	SQN	AND	OR
$e_{i,new}$	$e_{i,j}$	$\sum_{l=1}^{k} e_{i,j_l} *$ $[s_i * \Pi_{l=1}^{k}(1 - \mathrm{n}_{i,j_l}) +$ $(1 - s_i) * \Pi_{l=1}^{k} \mathrm{p}_{i,j_l}]$	$\sum_{l=1}^{k} e_{i,j_l}$	$\sum_{l=1}^{k} e_{i,j_l}$
$s_{i,new}$	$1 - s_{i,j}$	$s_i \Pi_{l=1}^{k}(1 - \mathrm{n}_{i,j_l}) +$ $(1 - s_i) \Pi_{l=1}^{k} \mathrm{p}_{i,j_l}$	$s_i \Pi_{l=1}^{k}(1 - \mathrm{n}_{i,j_l}) +$ $(1 - s_i) \Pi_{l=1}^{k} \mathrm{p}_{i,j_l}$	$s_i(1 - \Pi_{l=1}^{k} \mathrm{n}_{i,j_l}) +$ $(1 - s_i) \times$ $(1 - \Pi_{l=1}^{k}(1 - \mathrm{p}_{i,j_l}))$
$\mathrm{n}_{i,new}$	$1 - \mathrm{n}_{i,j}$	$1 - \Pi_{l=1}^{k}(1 - \mathrm{n}_{i,j_l})$	$1 - \Pi_{l=1}^{k}(1 - \mathrm{n}_{i,j_l})$	$\Pi_{l=1}^{k} \mathrm{n}_{i,j_l}$
$\mathrm{p}_{i,new}$	$1 - \mathrm{p}_{i,j}$	$\Pi_{l=1}^{k} \mathrm{p}_{i,j_l}$	$\Pi_{l=1}^{k} \mathrm{p}_{i,j_l}$	$1 - \Pi_{l=1}^{k}(1 - \mathrm{p}_{i,j_l})$

Table 7.3: Characteristics of filters constructed with LCI approximate predicates.

Under the above assumptions, we compute the characteristics for arbitrary predicate compositions in Section 7.5.2.

7.5.2 Propagating characteristics in a simple `Select` query

Consider the select query $\sigma_{I_i}[R_1]$. We consider alternate queries of the form $\sigma_{I_i}(\sigma_{F_i}[R_1])$, where F_i is some filter to I_1 composed only of predicates in A_i. (Including I_i in the alternate query ensures we make no positive errors, as discussed in Section 7.3.1.) If we trust the approximate predicates to make small false positive errors, we could leave out the ideal predicates, but this simple variation is not discussed here.) We now show how to compute the expense, selectivity and error characteristics of the filter F_i, which we call $e_{i,new}$, $s_{i,new}$, $p_{i,new}$ and $n_{i,new}$. Following that we compute the characteristics of the complete filtered ideal query. It is important to note that the following expressions are for both access predicates, as well as for restriction predicates.

Filter for an ideal predicate: We summarize in Table 7.2 the characteristics of filters assuming LI predicates. We derive similar entries for filters assuming LCI predicates in

Table 7.3. The **AND** and **OR** operators consider a set $\{A_{i,j_1}, A_{i,j_2}, \ldots, A_{i,j_k}\} \subseteq \mathrm{A}_i$, while the **SQN** operator considers an ordered list $A_{i,j_1} \rightarrow A_{i,j_2}, \ldots, \rightarrow A_{i,j_k}$ where $A_{i,j_l} \in \mathrm{A}_i, 1 \leq j \leq k$. The **NOT** operator considers a predicate $A_{i,j}$.

For example, selectivity of filter $\vee_{l=1}^{k} A_{i,j_l}$ (second row, last column of Table 7.2) is

$$\begin{aligned} s_{i,new} = P(\vee_{l=1}^{k} A_{i,j_l}) &= P(\neg \wedge_{l=1}^{k} \neg A_{i,j_l}) \qquad (DeMorgan's\ Law) \\ &= 1 - P(\wedge_{l=1}^{k} \neg A_{i,j_l}) \\ &= 1 - \Pi_{l=1}^{k}(1 - P(A_{i,j_l})) \quad (LI\ assumption) \\ &= 1 - \Pi_{l=1}^{k}(1 - s_{i,j_l}) \end{aligned}$$

Simialarly the selectivity of filter $\wedge_{l=1}^{k} A_{i,j_l}$ is $P(\wedge_{l=1}^{k} A_{i,j_l})$. But we know that $P(A_{i,j}) = P(I) * P(A_{i,j}|I_i) + P(\neg I_i) * P(A_{i,j}|\neg I_i)$. Hence selectivity of filter $\wedge_{l=1}^{k} A_{i,j_l}$ (second row, third column of Table 7.3) is

$$\begin{aligned} s_{i,new} = P(\wedge_{l=1}^{k} A_{i,j_l}) &= P(I_i) * P(\wedge_{l=1}^{k} A_{i,j_l}|I_i) + P(\neg I_i) * P(\wedge_{l=1}^{k} A_{i,j_l}|\neg I_i) \\ &= s_i * \Pi_{l=1}^{k} P(A_{i,j_l}|I_i) + \ (1 - s_i) * \Pi_{l=1}^{k} P(A_{i,j_l}|\neg I_i) \\ &= s_i * \Pi_{l=1}^{k}(1 - n_{i,j_l}) + \ (1 - s_i) * \Pi_{l=1}^{k} p_{i,j_l} \end{aligned}$$

The other table entries for selectivities and errors are similarly derived.

The entries for filter costs are derived as follows. The cost of a **SQN** filter is simply the cost of streaming the tuples through the predicates. The cost of the **NOT** filter is simply that of its restriction predicate $A_{i,j}$ because the same work must be performed (with reversed decisions).

The cost model for unioning or intersecting streams of tuples is trickier, since it depends on the application and how the data is stored. For this chapter, we assume the cost of unioning or intersecting streams to be negligible. This is the case, for example, if the access predicates generate sorted streams (sorted by say tuple-id). (The index used by $A_{i,j}$ may yield sorted tuples, or the tuples may be sorted dynamically, in which case the sort cost is included in $e_{i,j}$.) Note that information retrieval (IR) systems process unions and intersections in this fashion, and indeed the cost of the union and intersection of "inverted lists" (the sorted tuple ids) is typically negligible relative to the cost of accessing the index [SB88]. Other applications that use the same cost model include *mediators* that integrate a set of heterogeneous, remote databases – mediators typically assume the cost of performing unions

and intersections locally at the mediator is negligible, compared to the cost of accessing and streaming back tuples from the remote sites [CGMP96, LYGM99, VP97].

Of course, in some cases union and intersection costs may be significant. For example, if tuples are not sorted, we may have to use hashing to essentially execute a join. However for the rest of the chapter, we continue to assume the cost of performing unions and intersections is the cumulative cost of executing the access predicates. In Ref. [SGMC98] we present an alternate cost model based on hashing, and we discuss its impact on the algorithms presented in the chapter.

Filtered ideal query: Let e'_i, s'_i, n'_i and p'_i be the characteristics of the filtered ideal predicate. We then have $e'_i = e_{i,new} + s_{i,new} * e_i$, $n'_i = n_{i,new}$, $p'_i = 0$. Also, we have

$$\begin{aligned} s'_i &= P(F_{i,new} \wedge I_i) \\ &= P(F_{i,new}|I_i) * P(I_i) \\ &= (1 - n_{i,new}) * s_i \end{aligned}$$

7.5.3 Conjunctive filters

The number of alternate queries for a `Select` query is doubly exponential, since restriction predicates can be composed using $\vee$, $\wedge$, $\neg$ and $\rightarrow$, and access predicates can be composed using $\vee$ and $\wedge$. In this section we restrict the operators used to compose predicates without losing optimality for the *MIN-EXP* measure, using the following observations:

Observation 1 (Composing a set of access predicates)

1. *Access predicates should not be composed using* **OR**.

2. *Access predicates can be composed using the* **AND** *operator in case there is potential benefit, independent of predicate dependencies (such as LI or LCI).*

Proof: Recall from the definition of well-formed query plans that a set of access predicates can be composed only using $\vee$ and $\wedge$. Consider two access predicates $A_{i,j}$ and $A_{i,k}$. Now compare the cost of using $A_{i,j}$ as a filter to I_i, as opposed to using $A_{i,j} \vee A_{i,k}$. Note that the latter has a higher cost since $A_{i,j}$ and $A_{i,k}$ have to be executed, and duplicate tuples may have to be removed. Also the latter is less selective compared to the former. Hence the latter composition is not useful.

Now compare the cost of using $A_{i,j}$ as a filter to I_i, as opposed to using $A_{i,j} \wedge A_{i,k}$. Note that the latter is more expensive since $A_{i,j}$ and $A_{i,k}$ have to be executed, and the tuples satisfying the two predicates have to be intersected. However the latter is also more selective than the former, i.e., the latter feeds fewer tuples to subsequent predicates. Hence the latter filter is useful in cases when the cost of performing the intersection is offset by the reduced stream of tuples for subsequent processing. □

Observation 2 (Composing a set of restriction predicates) *Restriction predicates should be composed only using the* **SQN** *operator, independent of predicate dependencies.* **Proof:** Consider two restrictive predicates $A_{i,j}$ and $A_{i,k}$. While the expense of $A_{i,j} \vee A_{i,k}$ is the same as that of $A_{i,j} \wedge A_{i,k}$, the latter filter is more selective than the former, thereby streaming fewer tuples to subsequent predicates.

The **SQN** and **AND** operators are equivalent in their error and selectivity characteristics, and differ only in their expense characteristic, independent of predicate dependencies. Since composing restriction predicates using **SQN** leads to lower expenses, we compose restriction predicates using **SQN**. □

Based on Observations 1 and 2 we can safely restrict the set of query plans we need to consider to *conjunctive* filters.

Definition 7.5.1 (Conjunctive Filters) Given a set of access and restriction predicates, a conjunctive filter is produced by first composing the access predicates with the **AND** operator to form an **AND**-filter. Then the restriction predicates are composed using the **SQN** operator to form a **SQN**-filter. Finally, the *conjunctive* filter is formed by composing the **AND**-filter with the **SQN**-filter using a **SQN** operator. □

Figure 7.4 presents a *Conjunctive Wrapper* that considers only conjunctive filters. This wrapper is optimal for `Select` queries under the *MIN-EXP* measure. We only present the Steps that are modified from Figure 7.3. In Step [1] we consider all possible subsets of predicates that can be part of I_i's filter. In Step [2] we compose all the chosen predicates using **AND**. Notice that restriction predicates should be composed using **SQN**; however, they are composed with the **AND** for simplicity, since the underlying optimizer will anyway sequence the restriction predicates to minimize cost. Step [3] constructs the alternate query with the chosen predicates.

Algorithm 7.5.1 *ConjunctiveWrapper for* `Select` *query for* MIN-EXP *measure*
Inp I_i: Ideal predicate; A_i: Set of approximate predicates
Procedure
[0] $\cdots$
[1] For each $U_i \subseteq A_i$
[2] Compose $C_1 = \wedge_{a \in U_i}[a]$.
[3] Construct alternate query $\sigma_{I_i}(\sigma_{C_1}[R_1])$.
[4] – [8] $\cdots$

Figure 7.4: ConjunctiveWrapper for `Select` query for *MIN-EXP* measure.

From Step [2] we see that the number of conjunctive filters is still exponential since we can choose any subset of access and restriction predicates to build a conjunctive filter. While this is significantly more efficient than the doubly exponential naive algorithm of Section 7.4, this approach may still be unacceptable for some applications. In the next subsection, we propose heuristics for efficient filter construction with a polynomial number of calls to the optimizer, for the *MIN-EXP* measure. These heuristics will give us *provably* good filters for LI predicates, but may give us sub-optimal plans for LCI predicates.

7.5.4 Conjunctive filters computable in polynomial time

We construct the conjunctive filter in the following two steps.

1. *Compute* **SQN***-filter for restriction predicates:* We now choose the "right" subset of restriction predicates, and compose then with **SQN**. Assume L_i is an ordered list ($|L| = k$) of restriction predicates for ideal predicate I_i. Let $(e_{i,L_j}, s_{i,L_j}, n_{i,L_j}, p_{i,L_j})$ denote the characteristics of the j^{th} predicate in L. The ideal predicate I_i is the last one in the sequence, so we assume it is the $(k+1)^{st}$ entry in L_i. The cost of such a filter is shown in Table 7.4 (third column). To illustrate how the entries are obtained, consider the case of LI predicates. If t tuples are to be filtered, the cost of executing the first predicate A_{i,L_1}, is simply $t * e_{i,L_1}$. The second predicate receives only $s_{i,L_1} * t$ tuples, and will cost $(s_{i,L_1} * t) * e_{i,L_2}$ to execute. If we continue in this fashion, sum the costs, and divide by t to obtain the per tuple cost, we get the expression in Table 7.4. This is the expression that should be minimized for LI predicates, if we are to choose the right subset of restriction predicates.

Assumption/ Predicate	Access	Restriction
LI	$\sum_{A_{i,j} \in J_i} e_{i,j} + e_P * \Pi_{A_{i,j} \in J_i} s_{i,j}$	$\sum_{j=1}^{k+1} [e_{i,L_j} * \Pi_{l=1}^{j-1} s_{i,L_l}]$
LCI	$\sum_{A_{i,j} \in J_i} e_{i,j} +$ $e_P * [s * \Pi_{A_{i,j} \in J_i} (1 - n_{i,j}) +$ $(1 - s_i) * \Pi_{A_{i,j} \in J_i} p_{i,j}]$	$\sum_{j=1}^{k+1} [e_{i,L_j} *$ $(s * \Pi_{l=1}^{j-1} (1 - n_{i,L_l}) +$ $(1 - s_i) * \Pi_{l=1}^{j-1} p_{i,L_l})]$

Table 7.4: Minimization functions for *MIN-EXP* measure.

2. **AND**-*filter for access predicates:* We now choose the "right" subset of access predicates, and compose them with **AND**. Assume J_i ($\subseteq A_i$) is a set of access predicates on relation R, to be followed (composed using **SQN**) by some subsequent predicate P, with cost e_P.

   Table 7.4 (second column) gives the costs of the resulting filter, for the LI, LCI cases. To illustrate, consider the entry for LI predicates, $\sum_{A_{i,j} \in J_i} e_{i,j} + e_P * \Pi_{A_{i,j} \in J_i} s_{i,j}$ The first term ($\sum_{A_{i,j} \in J_i} e_{i,j}$) is the expense[1] incurred in executing the chosen access predicates, and ($e_P * \Pi_{A_{i,j} \in J_i} s_{i,j}$) is the expense of streaming the selected tuples ($\Pi_{A_{i,j} \in J_i} s_{i,j}$) through the subsequent predicate that costs e_P per tuple.

We first consider how to compute a good conjunctive filter by computing good **SQN** filters for an expensive predicate.

Theorem 7.5.1 (Choosing right subset of restriction predicates)
To choose the best subset of restriction predicates to filter ideal predicate I_i for the MIN-EXP *measure, it suffices to choose all restriction predicates with rank $\frac{e_{i,j}}{1-s_{i,j}} < e_i$.*

Proof: The quantity $e_i/(1 - s_i)$ is the cost per tuple filtered out by predicate T_i. Since an ideal predicate is the final predicate in a sequence, we can think of the selectivity of the ideal predicate s, as 0. Therefore in any optimal minimum expense filter we can use Smith's rule [Smi56] to argue that any predicate with $e_i/(1 - s_i) \geq e = e/(1 - 0)$ will be placed after the ideal predicate in the sequence. Since it is useless to have any predicate after the ideal predicate, no predicate with $e_i/(1 - s_i) \geq e$ will be used in any optimal solution.

Suppose there is a predicate T_l such that $e_l/(1-s_l) < e$ and is not in the optimal solution S. The expense of the optimal solution is $\Sigma_{i=1}^{k} [e_{L_i} * \Pi_{j=1}^{i-1} s_{L_j}] + s'e$ where $s' = \Pi_{i=1}^{k} s_{L_i}$. Now

[1] Recall from Section 7.2 that the expense of an access predicate is normalized with respect to the number of tuples in the database.

consider a modified solution where predicate T_l is introduced just before the ideal predicate in the sequence S. The expense of the new sequence is $\Sigma_{i=1}^{k}[e_{L_i} * \Pi_{j=1}^{i-1} s_{L_j}] + s' * e_l + s' * s_l * e$. Since $e_l/(1-s_l) < e$, it follows that $s' * e_l < s' * e - s' * s_l * e$ which implies that the expense of the modified solution is less than that of the optimal, a contradiction. □

Unfortunately, computing the right subset of access predicates is NP-Complete as we discuss in Ref. [SGMC98]. However, we are able to prove the following theorem for access predicates.

Theorem 7.5.2 (Choosing right subset of access predicates)
Suppose we construct an **AND***-filter as follows. First we rank the available access predicates by increasing value of (rank =)* $\frac{e_{i,j}}{\log(\frac{1}{s_{i,j}})}$*. Then, we construct the* **AND***-filter greedily: Let the* **AND***-filter contain the k-highest ranked predicates, k between 0 and the number of access predicates. If adding the* $(k+1)^{th}$*-highest ranked predicate into the* **AND***-filter reduces the expense of executing the alternate query, add the predicate to the* **AND***-filter. If the expense increases, the* **AND***-filter with the k-highest ranked predicates is the* **AND***-filter required. The filter so constructed is guaranteed to have expense no worse than twice that of optimal for the* MIN-EXP *measure.*

Proof: Assume without loss of generality that $e_{SF} = 1$ since we can divide all e_i and B by e_{SF} without changing the problem. For any subset S let E_S denote $\sum_{i \in S} e_i$ and P_S denote $\Pi_{i \in S} s_i$. Suppose we knew E_{S^*} where S^* is an optimal solution. Then we can reformulate the problem as a knapsack problem where the items have volume e_i and profits $\log \frac{1}{s_i}$ and the objective is to maximize the profits of the items picked with the constraint that the volume of the picked items is bounded by E_{S^*}. It is easy to see the equivalence. It is well known [GJ79] that the greedy algorithm which picks the items in increasing order of volume/profit while there is knapsack capacity left has an approximation ratio of 2.0. Notice that the greedy algorithm needs to know the knapsack capacity only to check when to stop and in fact the ordering of the items is independent of the knapsack capacity. In our reduction we do not know E_{S^*} but this is not a problem since we can pick the predicates greedily in increasing rank order and note the value of the total cost after the inclusion of each predicate. At the end of the process we will pick the point where the least total cost is achieved and we will be guaranteed to be within a factor of 2.0 of the optimal. □

Using the results of Theorems 7.5.1 and 7.5.2, we present in Figure 7.5 the *Linear-Wrapper* for a simple `Select` query (we only present the modifications over Figure 7.3).

Algorithm 7.5.2 *LinearWrapper for simple* `Select` *query for* MIN-EXP *measure*
Inp A_i: Set of approximate predicates
Procedure
[0] *BestPlan* := Dummy plan with infinite cost.
[1] Compute GR_i, a set of of all "good" approximate restriction predicates for I_i, such that $(rank =) \frac{e_{i,j}}{1-s_{i,j}} < e_i$.
Compute GA_i, an ordered list of approximate access predicates for I_i, sorted in increasing order of $(rank =) \frac{e_{i,j}}{\log(1/s_{i,j})}$.
[1a] $D_1 = \wedge_{gr \in GR_i}[gr]$
[2] For each $D_2 = \wedge_{ga_k \in GA_i}[ga_k]$, $1 \leq k \leq j$, $1 \leq j \leq |GA_i|$
[3] Construct alternate query $\sigma_{I_i}(\sigma_{D_1 \wedge D_2}[R_1])$.
[4] Q' := Q optimized using underlying query optimizer
[5] If $cost(Q') < cost(BestPlan)$
[6] $BestPlan := Q'$
[7] Else Return *BestPlan*
[8] Return *BestPlan*.

Figure 7.5: LinearWrapper for simple `Select` query for *MIN-EXP* measure.

LinearWrapper calls the underlying optimizer m times, where m is the number of access predicates. (If there are no access predicates, the optimizer is called once.)

Recall that *NaiveWrapper* and *ConjunctiveWrapper* assumed the optimizer returns estimates of the execution cost of the optimized version of any given alternate query, and also assumed they had access to system catalogs to lookup error estimates of various approximate predicates. *LinearWrapper* in addition to these assumptions assumes the wrapper has access to system catalogs to lookup expense and selectivity characteristics of predicates.

LinearWrapper is more efficient than the doubly-exponential *NaiveWrapper* and the singly-exponential *ConjunctiveWrapper* since it is linear in the number of approximate predicates. However it is only a heuristic: it is *provably good* for LI-dependent approximate predicates under our cost model, but may not produce optimal filters for LCI-dependent predicates.

7.6 Optimizing SPJ queries for *MIN-EXP* measure

Our *ConjunctiveWrapper* and *LinearWrapper* can easily be extended to deal with more general join queries, under the *MIN-EXP* measure. To illustrate, consider the query $Q :=$

$[\sigma_{I_1}(R_1)] \bowtie_{I_3} [\sigma_{I_2}(R_2)]$, where I_1, I_2 and I_3 are independent ideal predicates. The key idea is to treat this query, for selection of approximate predicates, as $Q := [\sigma_{I_1 \wedge I_2 \wedge I_3}(R_1 \times R_2)]$. This is equivalent to having the single predicate $I_1 \wedge I_2 \wedge I_3$ evaluated over relation $R_1 \times R_2$. (Note that this is just a conceptual way of looking at joins so we can compute the error characteristics – we are not physically implementing a cross-product!) Each approximate predicate for I_1, I_2 and I_3 can be considered approximates for $I_1 \wedge I_2 \wedge I_3$. Thus, our problem is again the selection of a good subset of approximate predicates for the single ideal predicate $I_1 \wedge I_2 \wedge I_3$.

Notice that the cost and selectivity for each approximate predicate still hold under the new ideal predicate $I_1 \wedge I_2 \wedge I_3$. For instance, consider an approximate predicate $A_{1,j}$ for I_1. Its selectivity over R_1, $s_{1,j}$, is the same as the selectivity over $R_1 \times R_2$. The false negative errors are also unchanged. For example, $n_{1,j} = P(\neg A_{1,j} | I_1)$ is identical to $P(\neg A_{1,j} | I_1 \wedge I_2 \wedge I_3)$ because $A_{1,j}$ is independent of I_2 and I_3. However, the false positive errors are changed, but can be computed from the parameters we already know. In particular, one can show that

$$p'_{1,j} = \frac{p_{1,j} * (1 - s_1) * s_2 * s_3 + s_{1,j} * (1 - s_2 * s_3)}{(1 - s_1 * s_2 * s_3)},$$

where $p'_{1,j}$ is the new error under $I_1 \wedge I_2 \wedge I_3$ and $p_{1,j}$ is the original error under I_1. We can compute false positive errors for approximate predicates of I_2, I_3 in a similar fashion. If two predicates for I_1 were LI (or LCI), they will continue to be LI (or LCI) under $I_1 \wedge I_2 \wedge I_3$. The I_1 predicates are all LI with respect to the I_2, I_3 predicates.

Once we have all the parameters for the approximate predicates, we can run *ConjunctiveWrapper* or *LinearWrapper* just as before. (Keep in mind that the wrappers only select approximate predicates to include; the underlying optimizer actually selects the join method and places the approximate and ideal predicates either before or after the join, in the order that minimizes costs, as in [CS96].) Similarly, if we wish to estimate the errors of a particular alternate query (to report to the user), we can use the expressions of Section 7.5.2.

7.7 Minimizing (p, n)-*MIN-EXP*

In this section we consider how to minimize the (p, n)-*MIN-EXP* measure. Unfortunately, the optimal plan may no longer be a conjunctive filter in this case. We present two examples

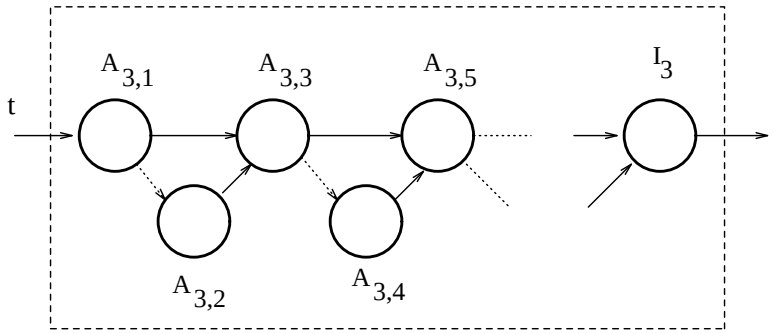


Figure 7.6: Example of a sawtooth filter.

to illustrate this.

EXAMPLE 7.7.1 (OR Filters): Reconsider Example 7.3.1 from Section 7.3. If the maximum allowable (p, n) is $(0, 1\%)$, a good filter (and the overall optimum) would be Plan (e). This plan is however not a conjunctive filter because it contains **OR** operators. In general, filters using **OR** operators reduce false negative errors and hence this space could contain good plans especially when the user can tolerate few false negatives. □

EXAMPLE 7.7.2 (Sawtooth Filters): In Figure 7.6 we present an example of what we call a sawtooth filter. That filter is for an ideal test I_3, and uses restriction predicates such as $A_{3,1}$ through $A_{3,5}$. In the figure we denote tuples that are rejected by a predicate using dotted outgoing links, while tuples that satisfy a predicate flow through the unbroken outgoing link. We see that tuples flowing through the dotted outgoing link of $A_{3,1}$ are sequenced through $A_{3,2}$. The tuples that satisfy $A_{3,2}$ are then combined with those that satisfy $A_{3,1}$ to be tested by $A_{3,3}$. Similarly for tests $A_{3,3}$ and $A_{3,4}$. Again, notice that these filters are not conjunctive.

A plan such as the one in Figure 7.6 may be useful in the following scenario. Say many tuples in a database are expected to satisfy I_3. Say $A_{3,1}$ is cheap, has high selectivity, few false positives but many false negatives, while $A_{3,2}$ is more expensive and has low false negative errors. In this case, $A_{3,1}$ acts as a fast filter to the ideal test and $A_{3,2}$ "protects" the filter by checking the $A_{3,1}$ rejects. Similarly for $A_{3,3}$ and $A_{3,4}$. □

Even though conjunctive filters may be suboptimal, it may still be useful to select the best conjunctive filter under the (p, n)-*MIN-EXP* measure. However, optimizing for

(p, n)-*MIN-EXP* for conjunctive filters is NP-Hard for both LI and LCI dependent predicates. We can easily see that the (p, n)-*MIN-EXP* **AND** filters for both LI and LCI predicates are NP-hard, since we can reduce the *MIN-EXP* **AND** filters to the (p, n)-*MIN-EXP* **AND** problems with maximum n set to 1. We show the reduction for the (p, n)-*MIN-EXP* **SQN** filter problem for LI predicates in Ref. [SGMC98].

Even though finding the best conjunctive filter is hard, we can still use the strategies of our previous wrappers as heuristics. For instance, the *LinearWrapper* can be modified as follows for the (p, n)-*MIN-EXP* measure: greedily insert restriction and access predicates based on their respective ranks, as long as the errors of the extended plan are tolerable. The extended plans that have tolerable errrors, will be passed to the underlying query optimizer for evaluation. The *ConjunctiveWrapper* can be similarly modified. These heuristic wrappers can be further extended so they consider a few promising **OR** filters, such as a simple **OR** of the predicates, or a sawtooth filter where high error predicates are protected.

7.8 Experiments

To understand the performance of our wrappers and the quality of the plans they generate, we conducted a variety of experiments.

First, we considered how our techniques will perform when we incorporate them into a text CDS. We assume we are given the approximate predicates in Chapter 2. For this experiment, we use the predicate characteristics as reported in Table 2.1 for the $50,000$ netnews articles. We assume that the ideal predicate is the hierarchical "diff" algorithm [CGM97], with an average cost of 20 milli-seconds per document comparison. (We believe that such a test could closely approximate what a person would consider a textual copy.)

Let us now consider the design of composite filters based on the above predicates. Since our goal is to evaluate our filtering algorithms independent of inter-predicate dependencies, we first assume that our predicates are LI. Let us consider four different maximum "n" values of 0, 0.05, 0.10, or 0.15. (We do not report results for $n > 0.15$ since the algorithms converge to the same composite filter as that for $n = 0.15$.) For each of these target n values, Table 7.5 gives (column "Linear") the cost of the best filter found by running the *Linear* algorithm. The table also shows the cost of the best filters produced by *Conjunctive* and *Naive*. The numbers in these two columns are found by exhaustive search. Notice that as expected, when the maximum allowable n increases, the expected expense (in seconds)

Max n	Linear	Conjunctive	Naive
0.0	1000	1000	1000
0.05	1000	1000	13.46
0.10	18.01	18.01	13.46
0.15	6.46	6.46	6.46

Table 7.5: Query plans for given n (LI Tests).

Max n	Linear	Conjunctive	Naive
0.0	1000	1000	1000
0.05	1000	1000	13.83
0.10	17.84	17.84	13.83
0.15	6.43	6.43	6.43

Table 7.6: Query plans for given n (LCI Tests).

decreases. We present the corresponding expenses of query plans if the SCAM tests are LCI in Table 7.6. In our scenario at least, the differences between LI and LCI predicates are not significant, and we observe the following. The *Linear* algorithm quickly identifies good composite filters without searching through an exponential or doubly exponential number of filters like the *Conjunctive* and *Naive* algorithms.

We also evaluated our techniques in the context of extensible databases. Some of the experiments used real approximate filters (approximating the location of an address by its zip code or area code, as opposed to a precise distance computation). Other experiments considered simulated queries and predicates, to evaluate performance over wider ranges of parameters. Here we only summarize one of the simulator experiments. The remaining results also confirm that our scheme works very well, yielding excellent plans with relatively little effort.

The goal of this one experiment was to understand how our wrappers perform when there are multiple ideal and approximate predicates. In this setup we consider a single SPJ query on randomly generated relations R_1, R_2. The query involves three ideal predicates, I_1 on R_1, I_2 on R_2, and a join predicate I_3. We assume that each ideal predicate has a number of LI approximate predicates; this number is varied in our experiments. The expense of our

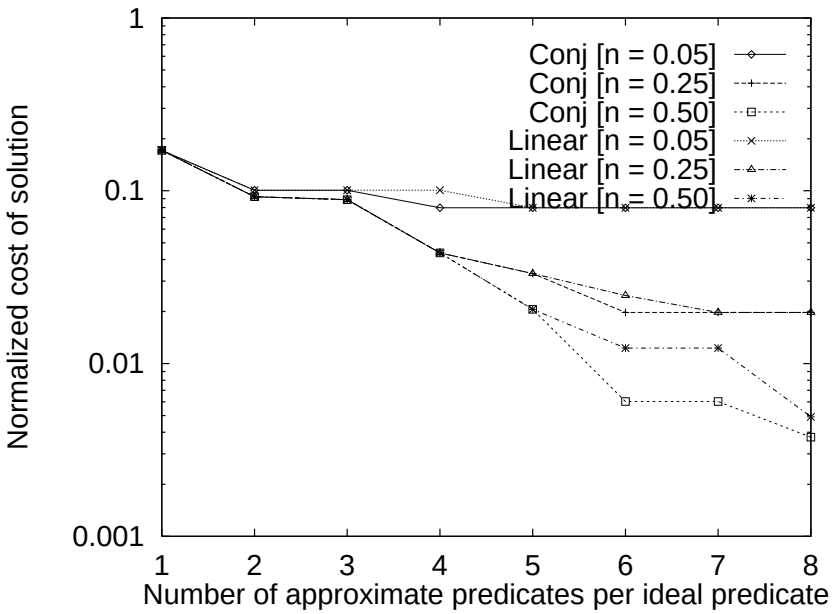


Figure 7.7: Quality of solutions with number of approximate predicates ($p = 0.0$).

ideal predicates was randomly chosen between $10,000$ and $20,000$ units. We represent this uniform distribution by $U(10000, 20000)$. If an approximate predicate is an access one, its cost follows the $U(100, 1000)$ distribution. If it is a restriction predicate, its distribution is $U(10, 50)$. The selectivity of all our predicates follows $U(0.01, 1.0)$. The false positive and negative errors of approximate predicates follow $U(0.01, 0.25)$.

In this experiment we evaluated the *normalized cost* of query plans generated by our wrappers. The normalized cost of a plan is defined as the execution cost of the plan divided by the cost of the plan that uses no approximate predicates. To compute the cost of a plan, we built a simple query optimizer (TQO) based on predicate placement [CS96] – our optimizer considered only sort-merge and hash-partitioned joins. We expect that as more approximate predicates become available, normalized costs will drop. In our experiments, we required solutions to have zero false positive errors (we performed experiments for other values of false positives, and observed similar results). We ran the following simulations 25 times, and report the average of our results.

In Figure 7.7 we show the normalized cost of solutions computed by our *ConjunctiveWrapper* (*Conj*), and by the greedy extension to *LinearWrapper* proposed in Section 7.7

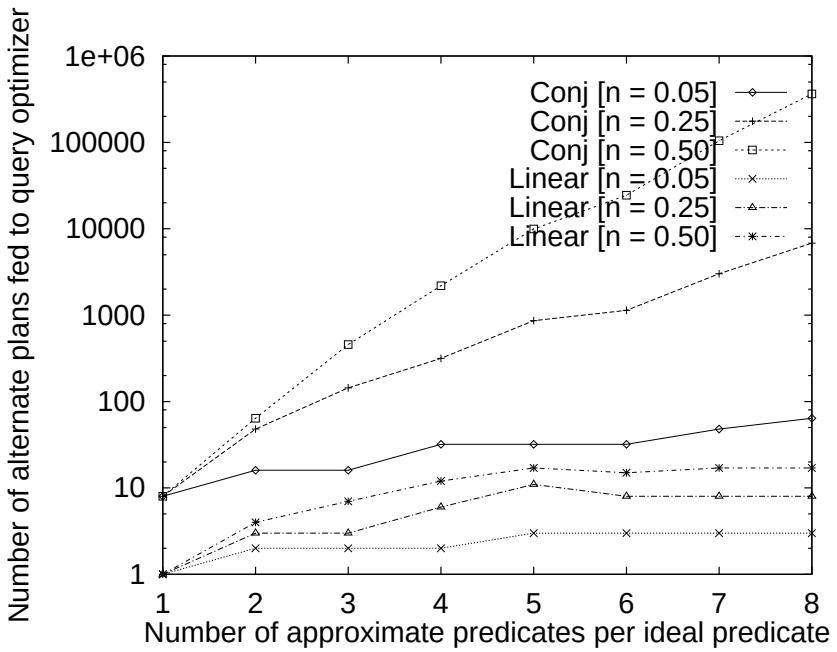


Figure 7.8: Number of plans with number of approximate predicates ($p = 0.0$).

(*Linear*), as the number of approximate predicates per ideal predicate varies. In Figure 7.8 we plot the number of alternate query plans fed by the wrapper to the underlying query optimizer. We see in Figure 7.7 that as we increase the number of approximate predicates (per ideal predicate), the normalized costs drops dramatically, especially as the user is willing to accept more errors. (Notice that the vertical axis is log scale.) Also observe that while the *Linear* wrapper yields higher cost solutions compared to *Conjunctive* wrapper, the difference is rather small. On the other hand, we see that the number of alternate query plans handed to the optimizer under *Conjunctive* is much larger than the equivalent number for *Linear*. Thus, we see that even though *LinearWrapper* was developed for *MIN-EXP* (which is not the measure used in this experiment), it still performed quite well under the (p, n)-*MIN-EXP* measure, with a much lower running time.

7.9 Conclusion

In this chapter, we discussed how to build a CDS to service a variety of user requirements in terms of tolerable expense and accuracy. We proposed the following *compositional* approach.

A domain expert designs a variety of predicates, each with different cost and accuracy characteristics. The CDS then automatically *composes* some of the given predicates and creates new predicates to service specific user requirements. In this chapter, we proposed and evaluated optimization techniques to construct "good" composite predicates. We noted that these techniques are very general and can be used to incorporate approximate predicates into extensible database systems.

Chapter 8

Indexing Temporal Repositories

8.1 Introduction

In a CDS system, new data is being added every day by publishers. For instance, a publisher such as ClariNews may add a few thousand articles each day to the CDS registry. However, they may also choose to *expire* articles more than a week old if they are not interested in protecting copyright of older articles. Index structures that we build to support our comparator operations should be reorganized as data is continually being added and expired. In this chapter, we consider the case when publishers require the CDS to maintain their registered objects for some *sliding* window of days, e.g., past week or month.

The problem of organizing index structures for a sliding time period is not unique to a CDS. For example, a Web search engine may provide an index for the past 30 days of Netnews articles, or a financial institution may keep an index of the stock market trades of the past 7 days. Each day, a batch of new data must be added to the index, and data older than the window should be removed.

In general, there are at least three (interrelated) reasons why such sliding window indices are useful. The first is that the application semantics require a sliding window. For example, if credit card bills can be contested for say up to 90 days, company agents may need to have fast access to the bills of exactly the past 90 days. A second reason is that user interest in data may wane over time. For instance, a stock market analyst may only want to look at recent trades, while a Netnews reader may not be interested in old data. So even if one could build an index for all the data, it would be less useful because it would give the user more information than he wants. A third reason is to reduce storage costs. For example,

until recently the Stanford University library maintained only the past 5 years of Inspec, a commercially available bibliography of technical papers. Clearly, at Stanford we were interested in older papers, but the library chose to provide fast index service for only the recent papers, and slower access (look through the stacks) for the rest. In this case, the sliding window index is a cache of what hopefully are the most frequently accessed papers.

Sliding window indices have been in use for many years, but the large volumes of data being generated today in some applications make it worthwhile to study these indices carefully. In particular, Internet search engines such as Altavista [Alt99], Infoseek [Inf99] and Dejanews [Dej99] are indexing ever-growing numbers of Web pages, Netnews articles, and other information. In Data Warehousing and On-line Analytical Processing (OLAP), huge volumes of sales, banking, and other transactions are being recorded and analyzed. In the rest of the high volume applications we have mentioned, there is often a similar need for indexing a window of days.

One obvious solution for indexing a window is to keep a single conventional index, and every day to delete the old data and insert the new batch of data into it. However, there are other interesting ways to maintain an index on a window of days, and we will see that they may have important advantages. To motivate, we now consider examples of a few such techniques in Tables 8.1, 8.2 and 8.3. In these examples, the techniques index a window of W days and partition the data across multiple indices. To service queries all indices will be accessed. The first row in each table is a "start" case where data of the first W days is indexed. On any subsequent day i, we need to index *new data* d_i into the required window. To do so, we execute the listed operations (under *Operation*). The columns labeled *Index* show the days that are covered by each index after the operations are executed. Some ways of maintaining an index of a window of days are:

1. *DEL:* We illustrate *DEL* in Table 8.1 with $W = 10$ and two indices, I_1 and I_2. On the tenth day, data of the first five days is indexed into I_1 and data of the next five days is indexed into I_2. When data d_{11} is available on the 11^{th} day, we first delete d_1 from I_1. We then index d_{11} into I_1. Similarly with subsequent days. *DEL* is similar to the obvious solution mentioned above, except that it uses multiple indices. Note that *DEL* maintains *hard* windows in that it indexes exactly the last W days (unlike *WATA*, one of the schemes we consider below).

2. *REINDEX:* We illustrate *REINDEX* in Table 8.2 with $W = 10$ and two indices, I_1

Day	New Data	Operation	Index (I_1)	Index (I_2)
10	$+ d_1, \ldots, d_{10}$	Start	$\{ d_1, d_2, d_3, d_4, d_5 \}$	$\{ d_6, d_7, d_8, d_9, d_{10}\}$
11	$+ d_{11}$	Delete d_1 from I_1	$\{ d_2, d_3, d_4, d_5 \}$	$\{ d_6, d_7, d_8, d_9, d_{10}\}$
		Add d_{11} to I_1	$\{ d_{11}, d_2, d_3, d_4, d_5 \}$	$\{ d_6, d_7, d_8, d_9, d_{10}\}$
12	$+ d_{12}$	Delete d_2 from I_1	$\{ d_{11}, d_3, d_4, d_5 \}$	$\{ d_6, d_7, d_8, d_9, d_{10}\}$
		Add d_{12} to I_1	$\{ d_{11}, d_{12}, d_3, d_4, d_5 \}$	$\{ d_6, d_7, d_8, d_9, d_{10}\}$
13	$+ d_{13}$	Delete d_3 from I_1	$\{ d_{11}, d_{12}, d_4, d_5 \}$	$\{ d_6, d_7, d_8, d_9, d_{10}\}$
		Add d_{13} to I_1	$\{ d_{11}, d_{12}, d_{13}, d_4, d_5 \}$	$\{ d_6, d_7, d_8, d_9, d_{10}\}$

Table 8.1: Deletion based index maintenance ($W = 10$).

Day	New Data	Operation	Index I_1	Index I_2
10	$+ d_1, \ldots, d_{10}$	Start	$\{ d_1, d_2, d_3, d_4, d_5\}$	$\{d_6, d_7, d_8, d_9, d_{10} \}$
11	$+d_{11}$	Reindex d_2, d_3, d_4, d_5, d_{11}	$\{ d_2, d_3, d_4, d_5, d_{11}\}$	$\{d_6, d_7, d_8, d_9, d_{10} \}$
12	$+d_{12}$	Reindex d_3, d_4, d_5, d_{11}, d_{12}	$\{ d_3, d_4, d_5, d_{11}, d_{12}\}$	$\{d_6, d_7, d_8, d_9, d_{10} \}$
13	$+d_{13}$	Reindex d_4, d_5, d_{11}, d_{12}, d_{13}	$\{ d_4, d_5, d_{11}, d_{12}, d_{13}\}$	$\{d_6, d_7, d_8, d_9, d_{10} \}$
14	$+d_{14}$	Reindex d_5, d_{11}, d_{12}, d_{13}, d_{14}	$\{ d_5, d_{11}, d_{12}, d_{13}, d_{14}\}$	$\{d_6, d_7, d_8, d_9, d_{10} \}$
15	$+d_{15}$	Reindex d_{11}, d_{12}, d_{13} d_{14} d_{15}	$\{ d_{11}, d_{12}, d_{13}, d_{14}, d_{15}\}$	$\{d_6, d_7, d_8, d_9, d_{10} \}$
16	$+d_{16}$	Reindex d_7, d_8, d_9, d_{10}, d_6	$\{ d_{11}, d_{12}, d_{13}, d_{14}, d_{15}\}$	$\{d_7, d_8, d_9, d_{10}d_{16} \}$

Table 8.2: Reindexing based index maintenance ($W = 10, n = 2$).

and I_2. On the tenth day, data of the first 10 days is indexed into I_1 and I_2 as in *DEL*. When data d_{11} is available on the 11^{th} day, we replace the expired d_1 in I_1 with d_{11}. We perform this by rebuilding index I_1 with data d_2, d_3, d_4, d_5 and d_{11}. Similarly with subsequent days. *REINDEX* also maintains hard windows.

3. *Wait and Throw Away (WATA):* We illustrate *WATA* in Figure 8.3 with $W = 10$ and four indices. On the tenth day, we index data of the first three days into I_1, data of the next three days into I_2, data of the subsequent three days into I_3 and data of the tenth day into I_4. When data d_{11} is available on the 11^{th} day, we add it to I_4. Similarly for d_{12}. When data d_{13} is available on the 13^{th} day, we first throw away I_1.

Day	New Data	Operation	Index I_1	Index I_2	Index I_3	Index I_4
10	$+d_1, ..d_{10}$	Start	$\{d_1, d_2, d_3\}$	$\{d_4, d_5, d_6\}$	$\{ d_7, d_8, d_9\}$	$\{d_{10}\}$
11	$+ d_{11}$	Add d_{11} to I_4	$\{d_1, d_2, d_3 \}$	$\{d_4, d_5, d_6\}$	$\{ d_7, d_8, d_9\}$	$\{d_{10}, d_{11}\}$
12	$+ d_{12}$	Add d_{12} to I_4	$\{d_1, d_2, d_3\}$	$\{d_4, d_5, d_6\}$	$\{ d_7, d_8, d_9\}$	$\{d_{10}, d_{11}, d_{12}\}$
13	$+ d_{13}$	Drop I_1	-	$\{d_4, d_5, d_6\}$	$\{d_7, d_8, d_9\}$	$\{d_{10}, d_{11}, d_{12}\}$
		Make $I_1 = \phi$	$\{ \}$	$\{d_4, d_5, d_6\}$	$\{d_7, d_8, d_9\}$	$\{d_{10}, d_{11}, d_{12}\}$
		Add d_{13} to I_1	$\{ d_{13} \}$	$\{d_4, d_5, d_6\}$	$\{d_7, d_8, d_9\}$	$\{d_{10}, d_{11}, d_{12}\}$
14	$+ d_{14}$	Add d_{14} to I_1	$\{d_{13}, d_{14}\}$	$\{d_4, d_5, d_6\}$	$\{d_7, d_8, d_9\}$	$\{d_{10}, d_{11}, d_{12}\}$

Table 8.3: WATA based index transitions ($W = 10, n = 4$).

We then create a new index I_1, and finally add d_{13} to it. The next day we add d_{14} to I_1, and so on.

Notice that in *WATA* we occasionally maintain data older than the required window. For example on days 11 and 12, data of d_1 is still indexed in I_1 even though it is no longer required as part of the window. *WATA* maintains *soft* windows. Such soft windows may be acceptable in certain applications. For instance, in case of Altavista it is probably acceptable to maintain a soft window of up to 35 days while the required window is only 30 days. Such soft windows may also be acceptable for statistical or trend analyses.

We now briefly consider some of the advantages of the schemes, as presented in the examples. Note however that in this chapter we will propose enhancements to these sample schemes, as well as additional schemes, so our comments should be taken as first indication of what might be good or bad about a scheme. We will have a more detailed and formal analysis of the schemes in Sections 8.5 and 8.6. Some of the advantages of the schemes in the example are:

- **Bulk insert/delete:** In *WATA*, deletions are performed in bulk by throwing away a whole index. If there are several deletes, bulk deletes may be more efficient than deleting an entry at a time (as in *DEL*). For instance in a commercial relational database such as Sybase, it takes a few milliseconds to throw away an index, irrespective of the index size. On the other hand, deleting an entry at a time takes time proportional to the number of deletes. Similarly, it may be efficient to reindex data, like *REINDEX*

does, if there are a lot of inserts and the index does not cover too many other days, because *incremental* indexing schemes [FJ92, TGMS94] may be expensive.

- **Better structured index:** Even though *REINDEX* may sometimes be more costly because it rebuilds indices from scratch, this rebuilding can often lead to a better structured index (e.g., less fragmentation and contiguous layout on disk). Such an index could lead to more efficient query processing. Thus, we can trade off more index build time for better query performance. This may be another reason to prefer *REINDEX* over *DEL* or *WATA*.

- **Simpler code:** With *REINDEX* and *WATA*, we do not need complex index deletion code [Jan95]. Also *REINDEX* does not require complex concurrency control, since updates and queries are operating on a different set of indices. We will later consider the case when *shadow* indices are used to avoid concurrency control code in all the schemes.

- **Legacy systems:** Some IR indexing packages (e.g., WAIS [KM91] and SMART [SB]) do not implement deletes at all. If we need to use some such package or a legacy system to maintain a window of days, we may have to use one of the new schemes such as *REINDEX* or *WATA*.

- **Query performance:** Clearly, having multiple indices creates more work for queries, as they must perform several searches. However in "data analysis" scenarios where query volume may be relatively low and data volumes may be high, the high query costs may be amortized by the savings under some of the categories listed above. Furthermore, if multiple disks and computers are available, the queries across indices can be easily parallelized. Also in some queries may be constrained to search over a subset of the indexed days, in which case fewer indices may be searched.

In this chapter we use the term *wave index* to refer to a collection of n "conventional" indices that provide access to a window of W consecutive time intervals ($1 \leq n \leq W$). We use the term "day" to refer to each time interval, although in general time intervals need not be 24 hours.

In the first part of this chapter (Sections 2 and 3), we propose six different wave indexing algorithms and three ways for performing updates within each algorithm. In particular, we formalize *DEL*, *REINDEX* and *WATA*, propose *REINDEX*$^+$, *REINDEX*$^{++}$ that improve

REINDEX, and finally describe *RATA*, a hybrid of *REINDEX* and *WATA*. Each of the above algorithms differ in (1) how the first W days are initially split across the n indices, (2) how the wave index is modified when a new day's data is available, and (3) whether they maintain "hard" or "soft" windows.

In the second part of this chapter (Section 8.5), we evaluate each of our proposed schemes for a variety of system performance measures. Through our evaluations we attempt to answer questions such as the following: (1) Given a new day's worth of data, how fast can a scheme index the data and make it available for querying? (2) How does the scheme perform as the query/update mix changes? (3) How much overall disk activity is required for maintaining a window and for servicing queries during a day? (4) How much disk space is required to index the data? (5) Does a scheme require complex code for deletion, or for concurrency control? (6) Can the scheme be implemented on top of "widely available" index structures, or is special code required?

In the final part of the chapter we consider three "case studies" and show how different wave indices may be appropriate in each scenario. First, we consider our own text CDS service. In addition, we also evaluate our techniques in the context of a generic Web search engine such as Altavista that indexes the same articles for general user queries, and a representative TPC-D benchmark [TPC99] query in a warehousing context. For each scenario we measure realistic parameters whenever possible (e.g., the volume of Netnews articles in a day), and make educated guesses when it is not possible (e.g., how many queries are likely in a web search engine). We believe that our results provide useful insights into the tradeoffs between the wave index schemes, and can help an application designer in selecting a wave index.

8.2 Preliminaries

In this section we outline the basic index structures used in this chapter, we describe how these are updated, and we define the operations to manage wave indices. Note that most of the ideas in this chapter are applicable to all classes of index structures, but for concreteness here we will focus on one specific class we now describe.

Figure 8.1 illustrates the basic index structures. The data we need to index consists of *records*. For instance, r_1 and r_2 in the figure are records. Each of the records has a *search field*, F, upon which an index is being built. Each record may have multiple values for F,

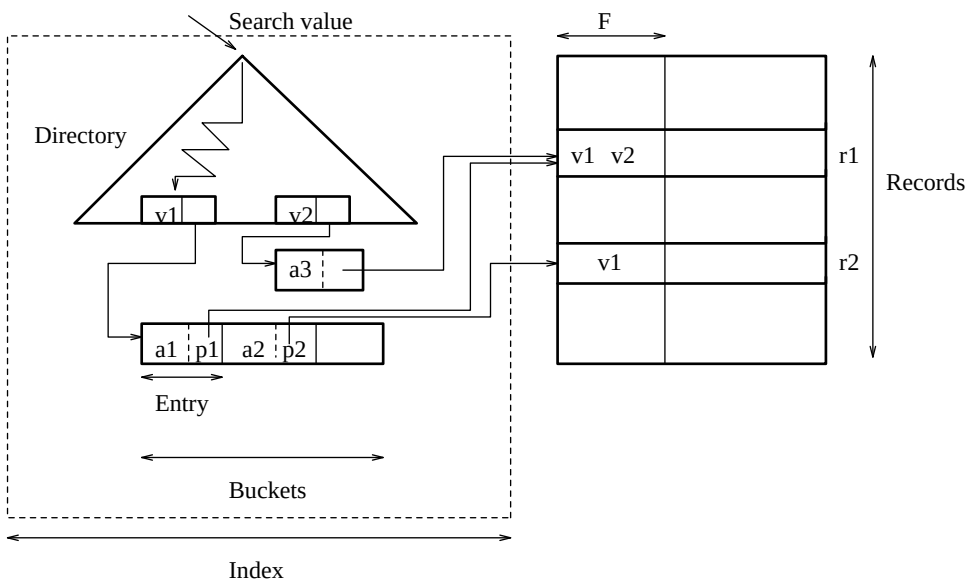


Figure 8.1: Basic index structures.

for example a record may have values "War" and "Peace" for its title field. Similarly, an employee record may have values $D55$ and $D57$ in its "department" field. The index consists of a *directory* and associated *buckets*. The directory is a search structure (e.g., a B+Tree or a hash table) that given a search value, v, identifies a bucket b. Bucket b contains a pointer p_i for each record r_i having search value v. In b, each pointer p_i may have additional associated information a_i. For example, in an Information Retrieval context, with each p_i we can store the byte offset of value v in field F of r_i. In a relational database context, with each p_i we may store additional attributes of r_i to speed up searches. For some of the indexing schemes we use here, we require a *timestamp* for each a_i, which denotes the day r_i was inserted. We refer to p_i and its associated information as an *entry*.

For simplicity we assume that the directory is in memory, and the buckets are on disk. We define an index to be *packed* if each of its buckets uses a minimal amount of space to store entries (without room for growth), and all its buckets are allocated contiguously on disk. If an index is not packed, we still assume that entries within a given bucket are contiguous. In some applications, packed indices may be preferable since they save space, and are efficient for queries that scan the whole index. For example, queries that compute some aggregate such as *sum, min* or *max* typically scan the whole index. If the index entries

are packed contiguously on disk, the query can be efficiently executed by scanning (with a single disk seek) the entries from the first bucket until the last bucket, and computing the aggregate.

A *wave index* on a search field F is used to search a collection of *days*, where each day contains the records generated during a particular time period (typically 24 hours). The time periods covered by the wave index should be contiguous. The days are partitioned into disjoint *clusters* and an index on F is built for each. Each individual index is termed a *constituent* index. The set of constituent indices is termed the wave index, Θ. In the rest of the chapter, when we use "index" we refer to a constituent of a wave index.

8.2.1 Update techniques

Suppose that we have an index on a set of records and the records change, or records are added to or deleted from the set. To update the index to reflect this batch of updates, we can use one of the following three techniques. (In this chapter we assume updates for a day are performed as a batch. This usually leads to better performance, mainly due to memory caching.)

1. **In-Place Updating:** For each update the directory and/or buckets are modified in-place. If there is not enough space in the bucket, then the bucket can be copied to a new location and allocated more space. To decide how much space to add, we could use techniques proposed in [FJ92]. This updating technique requires concurrency control to prevent queries from reading inconsistent data. Typically the resulting index is not packed even if the original one was packed.

2. **Simple Shadow Updating:** First make a copy of the index, and then for each update modify the new copy of the index in-place. Finally, the new index replaces the old version in the wave index. The main advantage of this technique is that queries can be serviced using the old index, while the new index is being updated. Hence no concurrency control is required. The corresponding disadvantage is that more space is required than with in-place updating while the new day is being indexed.

3. **Packed Shadow Updating:** This technique is similar to the simple shadow technique except that the resulting index is packed. Although this technique works in general, here we describe it when the updates consist of a set of inserted records, and records to be deleted are those with an expired timestamp. First we build a temporary

index for the new records to be inserted. We then scan the buckets of the index to be updated, copying them to a new contiguous location, but in the process deleting entries with expired timestamps, and leaving enough space in each new bucket copy to accommodate entries for the inserted records. Then we scan the temporary index, and append each bucket to the appropriate bucket in the new index, if one exists. If not, that bucket represents a new search value not present in the old index. We append such buckets after the last bucket in the new index. Finally we update the directory to reflect the new search values, and the new index replaces the old version in the wave index.

8.2.2 Operations on a Wave Index

In describing our wave index algorithms, we use the following primitive functions. For simplicity we use integers to refer to days. Thus the days indexed by I in a wave index Θ can be represented by a set of integers, referred to as the *time-set* of the index.

1. **Wave index update operations:**

   (a) **AddIndex(I, Θ):** Given a wave index Θ and an index I, this operation adds I to the set of constituent indices in Θ.

   (b) **DropIndex(I, Θ):** Given a wave index Θ and an index I, this operation first removes I from Θ. It then deletes all index entries in I (i.e., reclaims space).

2. **Constituent index update operations:**

   (a) **BldIdx(*Days*):** Given *Days*, a set of integers, this operation builds a packed index for the batch of records in those days. i.e., for the cluster identified by *Days*. We assume here that a packed index is achieved by scanning the *Days* records and counting the number of entries needed in each bucket. Then contiguous buckets of the appropriate size are allocated on disk.

   (b) **AddToIdx(*Days*, I):** Given *Days*, a set of integers, and an index I, this operation incrementally adds the batch of entries for *Days* records to I. This can be achieved using any one of techniques in Section 8.2.1. Thus if in-place or simple shadow updating is used, the resulting I will not be packed. If packed shadows are used, then I is replaced in the wave index by a new packed index.

(c) **DeleteFromIndex($Days$, I):** Given $Days$, a set of integers, and an index I this operation incrementally deletes entries for $Days$ records from I. Like $AddToIdx$, this can also be performed using any of the three techniques in Section 8.2.1. Again if in-place or simple shadow updating are used, I will not be packed. If packed shadow updating is used, I will be packed.

Note that *BldIdx* and *AddToIdx* can often be used to achieve the same goal. However the performance can be very different. For instance, let a cluster have five days worth of data and suppose that we already have an index for the first four days. We can construct an index for the 5-day cluster either by adding the the fifth day to the existing index, or by building the index from scratch for the 5 days. The former option is typically less expensive than the latter. However, unless packed shadowing was used in the former, the latter will be more efficient for scan queries since the resulting index is packed. On the other hand, if we do not have the initial 4-day index, it is typically more efficient to do a *BldIdx* rather than a series of *AddToIdx* operations.

3. **Access operations:**

We expect four kinds of queries to access the wave index. They are *IndexProbe*, *SegmentScan*, *TimedIndexProbe* and *TimedSegmentScan*. To illustrate, consider a set of daily sales records for the past year, indexed by the sales person. Let us assume that each index entry contains, in addition to a pointer to full sales record, the amount and date of sale (i.e., when the record was inserted.) A query that looks at all sales entries for a given salesperson, $S1$, will be executed as an *IndexProbe*, which probes the index with search value $S1$. A query that looks at sales entries of $S1$ for the past month will be executed as a *TimedIndexProbe*, which is an *IndexProbe* restricted to entries with a date in the past month. A query to compute aggregate yearly sales by sales person for the store will be executed as a *SegmentScan*, which scans all buckets of the index. A query to compute aggregate sales for the past month will be executed as a *TimedSegmentScan*, which is a *SegmentScan* restricted to entries inserted in the past month. As we shall see now, *IndexProbe* and *SegmentScan* can be expressed as *TimedIndexProbe* and *TimedSegmentScan* respectively.

(a) **TimedIndexProbe(Θ, T_1, T_2, s):** Given a wave index Θ, times T_1 and T_2 and search value s, this operation retrieves buckets of entries for v inserted between

day T_1 and T_2. Specifically, this operation probes a subset of constituent indices in Θ whose clusters have days more recent than T_1 and older than time T_2. For each such index, buckets for s are retrieved, and entries with insert time in the desired range are selected. Note that if we restrict timed queries to only refer to time intervals that correspond to the cluster intervals, then bucket entries do not need insertion times. That is, all entries for s in the indices in the T_1, T_2 range will be relevant. When $T_1 = -\infty$ and $T_2 = \infty$, this operation is equivalent to an *IndexProbe* that probes all indices.

(b) **TimedSegmentScan(Θ, T_1, T_2):** Given a wave index Θ and times T_1 and T_2, this operation retrieves all entries inserted between day T_1 and T_2. It does this by scanning buckets of all constituent indices in Θ whose clusters have days more recent than time T_1 and older than time T_2. When $T_1 = -\infty$ and $T_2 = \infty$, this operation is equivalent to a *SegmentScan* that scans all buckets in all indices.

8.3 Building simple wave indices

In this section, we review the simple algorithms to build wave indices that we presented in Section 8.1. Let d_i refer to the i^{th} days' data, and d_{new} refer to a new day's data. Let Θ be the wave index being maintained.

8.3.1 Deletion *(DEL)*

We briefly motivated *DEL* in the Introduction with Table 8.1. In *DEL*, we initially index W/n days of data[1] each in indices $I_1, I_2, \ldots, I_n$. We then make $I_1, I_2, \ldots, I_n$ constituent indices of Θ. Every day when d_{new} is available, we delete entries of d_{new-W} from I_j that indexed d_{new-W}. Then we insert entries for d_{new} to I_j. The deletion and insertion can be performed using one of the update techniques proposed in Section 8.2.1. We present the formal *DEL* algorithm in Appendix A as Figure 10.1.

DEL maintains hard windows. If in-place or simple shadow updating are used, *DEL* requires code to implement incremental deletion in both the directory and the buckets. Also the resulting index is not packed. If packed shadow updating is used, the resulting index is however packed.

[1] In the formal algorithm in Appendix A, we handle the case when W/n is not an integer.

8.3.2 Reindexing (*REINDEX*)

We briefly motivated *REINDEX* in the Introduction with Table 8.2. The operations performed at each step in the example is actually a *BldIdx*. We formally present the *REINDEX* algorithm in Appendix A as Figure 10.2.

REINDEX maintains hard windows, and the resulting index is packed. However this technique requires reindexing W/n days worth of data every day. In Section 8.4 we propose several schemes that reduce the work done while building the index.

8.3.3 Wait and Throw Away (WATA)

We briefly motivated the WATA approach in the Introduction with Table 8.3. Recall that this algorithm uses a lazy form of deletion by throwing away an entire index only when all its entries have expired. Clearly there are several ways to implement this type of lazy deletion. For example, Table 8.4 presents a scheme that is slightly different from the one in Table 8.3, for the same $W = 10$, $n = 4$. We see that on the 10^{th} day the example in Table 8.4 forms different clusters for the four indices than we had earlier. While there are a variety of measures we can use to evaluate different WATA-based schemes, we concentrate on the following measures for the purposes of this chapter:

- **Length of index:**

  For some applications, we may prefer a WATA scheme that gives us the "tightest" soft window. For instance, if we are computing the average revenue and standard deviation for the past week from the sales relation of the company, we may prefer a WATA scheme with small soft windows for more accuracy.

  In Table 8.4, we see that the total number of days indexed on days 11, 12, 13 is 11, 12 and 13, respectively. We define the *length* of the wave index so constructed to be 13, the maximum number of days stored in the index at any time. Similarly the length of the index constructed in Table 8.3 is 12, since the total number of days indexed on days 11, 12, 13 is 11, 12 and 10. Since the example in Table 8.3 has a smaller length, it indexes fewer extra days thereby providing a "tighter" window.

- **Size of index:**

  In many applications, we may prefer to use a WATA scheme that incurs the least space overhead due to lazy deletion. That is, we need to minimize the total *index size*, where

index size is the maximum storage required for maintaining the wave index.

When the size of data to be indexed is the same from day-to-day, minimizing index size corresponds to minimizing index length. However, the size of data to be indexed can vary dramatically across days. For example, the number of daily Usenet postings in popular newsgroups varies dramatically depending on the day of the week. In Figure 8.2 we report the total number of daily postings in September 1997 across about 10,000 popular newsgroups subscribed to by the Stanford Computer Science department's NNTP server. We see that the number of postings on the second Wednesday is about 110,000, while on Sundays, the number of postings falls to around 30000. We will discuss algorithms to minimize the index size in case of non-uniform data sizes later in this section.

In Figure 10.5 of Appendix A we propose one instance of WATA termed WATA*. For this algorithm, we can show the following (proof in Appendix B).

Theorem 8.3.1 *(Index length)*
WATA* *is an optimal algorithm to construct a* WATA *wave index with the smallest index length.* □

It is easy to see that to construct an optimal WATA index for index size, we need complete information of data sizes of all future days. Since such information is typically not available, we need to design an *online* algorithm that adds a new day's data to the wave index based only on the current day's data and the currently indexed days. However we can use the *WATA** algorithm for minimizing index size as well, due to the following property.

Theorem 8.3.2 *(Index size)*
WATA* *is an online algorithm to construct wave indices with index size no more than twice the index size of any optimal* WATA *algorithm that has complete knowledge of future data sizes. That is, the* competitive ratio *[MR95] of* WATA* *is* 2.0 *for the index size measure.* □

Kleinberg et al [KMRV97] recently extended our work in WATA index construction in the following directions. They proposed an optimal WATA algorithm, for the case when they have complete knowledge of data sizes in the future. For the online problem, they improved the competitive ratio of our WATA algorithm to $\frac{n}{n-1}$, for n indices by assuming

Day	Operation	Index I_1	Index I_2	Index I_3	Index I_4
10	$I_1 \leftarrow$ BldIdx($\{1, 2, 3, 4\}$) $I_2 \leftarrow$ BldIdx($\{5, 6, 7\}$) $I_3 \leftarrow$ BldIdx($\{8, 9, 10\}$) $I_4 \leftarrow \{\ \}$	$\{d_1, d_2, d_3, d_4\}$	$\{d_5, d_6, d_7\}$	$\{d_8, d_9, d_{10}\}$	$\{\ \}$
11	AddToIdx($\{11\}$, I_4)	$\{d_1, d_2, d_3, d_4\}$	$\{d_5, d_6, d_7\}$	$\{d_8, d_9, d_{10}\}$	$\{d_{11}\}$
12	AddToIdx($\{12\}$, I_4)	$\{d_1, d_2, d_3, d_4\}$	$\{d_5, d_6, d_7\}$	$\{d_8, d_9, d_{10}\}$	$\{d_{11}, d_{12}\}$
13	AddToIdx($\{13\}$, I_4)	$\{d_1, d_2, d_3, d_4\}$	$\{d_5, d_6, d_7\}$	$\{d_8, d_9, d_{10}\}$	$\{d_{11}, d_{12}, d_{13}\}$
14	$I_1 \leftarrow \phi$ AddToIdx($\{14\}$, I_1)	$\{\ \}$ $\{d_{14}\}$	$\{d_5, d_6, d_7\}$ $\{d_5, d_6, d_7\}$	$\{d_8, d_9, d_{10}\}$ $\{d_8, d_9, d_{10}\}$	$\{d_{11}, d_{12}, d_{13}\}$ $\{d_{11}, d_{12}, d_{13}\}$

Table 8.4: Another example of index transitions based on WATA (W = 10, n = 4).

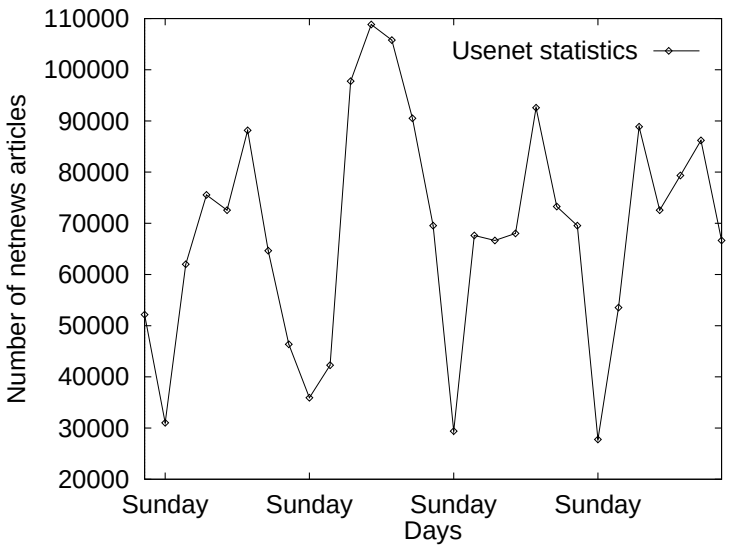


Figure 8.2: Number of Usenet postings per day in September 1997.

they know the total maximum index size ever possible in the future, ahead of time. Recall that our WATA* is "purely" online in that it assumes no such information, while still providing a competitive ratio of 2.0.

All WATA algorithms maintain soft windows and thereby use more space to store the extra days of data. However, they do relatively little work each day, and index deletion code is not needed. Also, once a new day's data is available, it takes only the time of one *AddToIdx* before the new data is available for querying. However, *TimedSegmentScans* may be less efficient due to the entries of days older than the window, but that are part of the soft window. Another potential disadvantage of WATA is that it requires at least two constituent indices to be efficient. To see this, consider the case when there is only one constituent index. In that case, each new day has to be added to the single index, and at no point will all data in the index expire to allow the removal of the index. Hence, the constituent index will then keep growing forever. For this reason, we require at least two constituent indices for WATA.

8.4 Enhancing simple wave indices with temporary indices

In this section, we enhance the simple wave-indexing algorithms of Section 8.3 by constructing temporary indices. These enhanced schemes improve important performance measures such as average maintenance work, and time to add new data, at the cost of using more disk space.

8.4.1 Improved reindexing ($REINDEX^+$)

This scheme enhances *REINDEX* by reducing the average work required in maintaining a wave index. To motivate $REINDEX^+$, we reconsider the example for *REINDEX* in Table 8.2. Note that index entries for d_{11} are recomputed every day from day 11 to day 15. Similarly, index entries for d_{12} are recomputed every day from day 12 to day 15. Similarly for d_{13} and d_{14}. Instead $REINDEX^+$ maintains a temporary index, *Temp*, to avoid recomputing these index entries every day.

In Table 8.5, we present an example of how $REINDEX^+$ works with $W = 10$ and $n = 2$. In this table (and in subsequent tables) we drop column *New Data* and assume that on day i ($i > W$), data d_i is available to be indexed. We add column *Temp* to show the current entries in *Temp*. On the 10^{th} day, the first five days are indexed in I_1 and the next five days

are indexed in I_2 (as in *DEL* and *REINDEX*). In addition, an empty index *Temp* is created. On the 11^{th} day when new data d_{11} is available, the cluster of I_1 should contain days d_{11}, d_2, d_3, d_4, and d_5. For this, we first index d_{11} into *Temp*. We then copy *Temp* into I_1 so I_1 contains entries for d_{11}. Then we incrementally add d_2, d_3, d_4 and d_5 into I_1. On the 12^{th} day after new data d_{12} is available, the cluster of I_1 should contain days d_{11}, d_{12}, d_3, d_4, and d_5. For this, we first add new data d_{12} to *Temp*. We then copy *Temp* into I_1 so I_1 contains entries for d_{11} and d_{12}. Finally we incrementally add d_3, d_4 and d_5 to I_1. Similarly for subsequent days. Observe that between days d_{11} and d_{15} we are incrementally indexing progressively fewer days. This reoccurs between days d_{16} and days d_{20} and so on. We can see that the average number of days indexed per transition by *REINDEX*$^+$ during index build is about half that of *REINDEX*. The *REINDEX*$^+$ algorithm is formally described in Appendix A as Figure 10.3.

REINDEX$^+$ maintains hard windows. If we use in-place or simple shadow updating to update the constituent indices, the resulting index is not packed. If we use packed shadow updating instead, the resulting index is packed. Every day, this scheme on the average reindexes about half the number of days that *REINDEX* does, by using additional space to store a temporary index, *Temp*. Also like *REINDEX*, it does not require code for deleting from an index.

8.4.2 Further improved reindexing (*REINDEX*$^{++}$)

This scheme improves *REINDEX*$^+$ by reducing the time to index new data and making new data available sooner for querying. We achieve this by performing most of the work required in maintaining the wave index before the data is available. For this, we use a few temporary indices ($T_1, T_2, \ldots$) and increase our storage requirements.

We explain how *REINDEX*$^{++}$ works using the example in Table 8.6 with $W = 10$ and two indices, I_1 and I_2. On the 10^{th} day, we index the first five days in I_1 and the next 5 days in I_2. Then we build temporary indices $T_0, T_1, \ldots, T_4$ as follows. We initialize T_0 to an empty index, and we create T_1 with day 5. Then we copy T_1 to index T_2, and incrementally add day 4 to it, so T_2 contains days 4 and 5. Similarly for T_3 and T_4, so that T_3 contains days $3, 4, 5$ and T_4 contains days $2, 3, 4, 5$, as shown in column *Temp*. On the 11^{th} day, add d_{11} to T_4. Then rename T_4 as I_1 so that I_1 now contains days $2, 3, 4, 5$ as well as 11. Queries can start accessing data of d_{11} at this point much faster than if *REINDEX* were used. We then add d_{11} to T_3 so T_3 now contains days $3, 4, 5$ as well as day 11. Indexes T_2, T_1 and T_0

Day	Operation	Index I_1	Index I_2	Temp
10	Temp $\leftarrow \phi$ $I_1 \leftarrow$ BldIdx($\{1,2,3,4,5\}$) $I_2 \leftarrow$ BldIdx($\{6,7,8,9,10\}$)	$\{d_1, d_2, d_3, d_4, d_5\}$	$\{d_6, d_7, d_8, d_9, d_{10}\}$	ϕ
11	*Temp*, $I_1 \leftarrow$ BldIdx($\{11\}$) AddToIdx($\{2,3,4,5\}, I_1$)	$\{d_{11}, d_2, d_3, d_4, d_5\}$	$\{d_6, d_7, d_8, d_9, d_{10}\}$	$\{d_{11}\}$
12	AddToIdx($\{12\}$, Temp) $I_1 \leftarrow$ Temp AddToIdx($\{3,4,5\}, I_1$)	$\{d_{11}, d_{12}, d_3, d_4, d_5\}$	$\{d_6, d_7, d_8, d_9, d_{10}\}$	$\{d_{11}, d_{12}\}$
13	AddToIdx($\{13\}$, Temp) $I_1 \leftarrow$ Temp AddToIdx($\{4,5\}, I_1$)	$\{d_{11}, d_{12}, d_{13}, d_4, d_5\}$	$\{d_6, d_7, d_8, d_9, d_{10}\}$	$\{d_{11}, d_{12}, d_{13}\}$
14	AddToIdx($\{14\}$, Temp) $I_1 \leftarrow$ Temp AddToIdx($\{5\}, I_1$)	$\{d_{11}, d_{12}, d_{13}, d_{14}, d_5\}$	$\{d_6, d_7, d_8, d_9, d_{10}\}$	$\{d_{11}, d_{12}, d_{13}, d_{14}\}$
15	$I_1 \leftarrow$ Temp AddToIdx($\{15\}, I_1$) Temp $\leftarrow \phi$	$\{d_{11}, d_{12}, d_{13}, d_{14}, d_{15}\}$	$\{d_6, d_7, d_8, d_9, d_{10}\}$	ϕ
16	*Temp*, $I_2 \leftarrow$ BldIdx($\{16\}$) AddToIdx($\{7,8,9,10\}, I_2$)	$\{d_{11}, d_{12}, d_{13}, d_{14}, d_{15}\}$	$\{d_{16}, d_7, d_8, d_9, d_{10}\}$	$\{ d_{16} \}$

Table 8.5: Example of index transitions in *REINDEX*$^+$ (W = 10, n = 2).

remain unchanged (they are not shown in order to reduce the size of the table.) On day 12, we add d_{12} to T_3 so it contains days $3, 4, 5, 11$ as well as day 12. As earlier, we rename T_3 as I_1 and queries can start accessing data of d_{12} at this point. We then add d_{11} and d_{12} to T_2 to be used the next day. Indexes T_1 and T_0 remain unchanged. Similary for days 13 and 14. On day 15 we reinitialize T_0, T_1, ..., T_4 for the next set of days. We formally present the algorithm for *REINDEX*$^{++}$ in Appendix A as Figure 10.4.

REINDEX^{++} maintains hard windows. Like *REINDEX*$^+$, the constituent indices are packed only if packed shadow updating is used. Notice that in *REINDEX*$^{++}$ we are doing marginally additional amount of work compared to *REINDEX*$^+$. On any given day, we are adding the new day's data to about half the indices which is the work done in *REINDEX*$^+$. In addition on days 10, 15, ..., we incrementally index 4 days of data. In general, we would incrementally index W/n days of data every W/n days. Clearly this work can be spread across the W/n days. Hence *REINDEX*$^{++}$ performs about the same amount of work as *REINDEX*$^+$, but reduces the time to index a new day's data.

Operation	Index I_1	Index I_2	Temp
$I_1 \leftarrow$ BldIdx($\{12345\}$)			
$I_2 \leftarrow$ BldIdx($\{678910\}$)	$\{d_1 d_2 d_3 d_4 d_5\}$	$\{\ d_6 d_7 d_8 d_9 d_{10}\ \}$	
$T_0 \leftarrow \phi$, $T_1 \leftarrow$ BldIdx($\{5\}$)			$T_0 = \phi$, $T_1 = \{d_5\}$
$T_2 \leftarrow T_1$, AddToIdx($\{4\}, T_2$)			$T_2 = \{d_5 d_4\}$
$T_3 \leftarrow T_2$, AddToIdx($\{3\}, T_3$)			$T_3 = \{d_5 d_4 d_3\}$
$T_4 \leftarrow T_3$, AddToIdx($\{2\}, T_4$)			$T_4 = \{d_5 d_4 d_3 d_2\}$
AddToIdx($\{11\}, T_4$)			$T_4 = \{d_2 \ldots d_5 d_{11}\}$
Rename T_4 as I_1	$\{d_2 \ldots d_5 d_{11}\}$	$\{d_6 d_7 d_8 d_9 d_{10}\}$	
AddToIdx($\{11\}, T_3$)			$T_3 = \{d_5 d_4 d_3 d_{11}\}$
AddToIdx($\{12\}, T_3$)			$T_3 = \{d_5..d_3 d_{11} d_{12}\}$
Rename T_3 as I_1	$\{d_5..d_3 d_{11} d_{12}\}$	$\{d_6 d_7 d_8 d_9 d_{10}\}$	
AddToIdx($\{11, 12\}, T_2$)			$T_2 = \{d_5 d_4 d_{11} d_{12}\}$
AddToIdx($\{13\}, T_2$)			$T_2 = \{d_5 d_4 d_{11} d_{12} d_{13}\}$
Rename T_2 as I_1	$\{d_5 d_4 d_{11} d_{12} d_{13}\}$	$\{d_6 d_7 d_8 d_9 d_{10}\}$	
AddToIdx($\{11, 12, 13\}, T_1$)			$T_1 = \{d_5 d_{11} d_{12} d_{13}\}$
AddToIdx($\{14\}, T_1$)			$T_1 = \{d_5 d_{11} d_{12} d_{13} d_{14}\}$
Rename T_1 as I_1	$\{d_5 d_{11} d_{12} d_{13} d_{14}\}$	$\{d_6 d_7 d_8 d_9 d_{10}\}$	
AddToIdx($\{11, 12, 13, 14\}, T_0$)			$T_0 = \{d_{11} d_{12} d_{13} d_{14}\}$
AddToIdx($\{15\}, T_0$)			$T_0 = \{d_{11} d_{12} \ldots d_{15}\}$
Rename T_0 as I_1	$\{d_{11} d_{12} d_{13} d_{14} d_{15}\}$	$\{d_6 d_7 d_8 d_9 d_{10}\}$	
$T_0 \leftarrow \phi$, $T_1 \leftarrow$ BldIdx($\{10\}$)			$T_0 = \phi$, $T_1 = \{d_{10}\}$
$T_2 \leftarrow T_1$, AddToIdx($\{9\}, T_2$)			$T_2 = \{d_{10} d_9\}$
$T_3 \leftarrow T_2$, AddToIdx($\{8\}, T_3$)			$T_3 = \{d_{10} d_9 d_8\}$
$T_4 \leftarrow T_3$, AddToIdx($\{7\}, T_4$)			$T_4 = \{d_{10} d_9 d_8 d_7\}$
AddToIdx($\{16\}, T_4$)			$T_4 = \{d_{11} d_{12} \ldots d_{15}\}$
Rename T_4 as I_2	$\{d_{11} d_{12} d_{13} d_{14} d_{15}\}$	$\{d_{10} d_9 d_8 d_7 d_{16}\}$	
AddToIdx($\{16\}, T_3$)			$T_3 = \{d_{10} d_9 d_8 d_{16}\}$

Table 8.6: Example of index transitions in *REINDEX*$^{++}$ (W = 10, n = 2).

Operation	Index I_1	Index I_2	Index I_3	Index I_4	Temp
$I_1 \leftarrow$ BldIdx($\{1,2,3\}$) $I_2 \leftarrow$ BldIdx($\{4,5,6\}$) $I_3 \leftarrow$ BldIdx($\{7,8,9\}$) $I_4 \leftarrow$ BldIdx($\{10\}$) $T_0 \leftarrow$ BldIdx($\{3\}$) $T_1 \leftarrow T_0$ AddToIdx($\{2\}, T_0$)	$\{d_1, d_2, d_3\}$	$\{d_4, d_5, d_6\}$	$\{d_7, d_8, d_9\}$	$\{d_{10}\}$	$T_0 = \{d_3\}$ $T_1 = \{d_3, d_2\}$
AddToIdx($\{11\}, I_4$) Drop I_1 Rename T_0 as I_1	$\{d_3, d_2\}$	$\{d_4, d_5, d_6\}$	$\{d_7, d_8, d_9\}$	$\{d_{10}, d_{11}\}$	Same: T_1
AddToIdx($\{12\}, I_4$) Drop I_1 Rename T_1 as I_1	$\{d_3\}$	$\{d_4, d_5, d_6\}$	$\{d_7, d_8, d_9\}$	$\{d_{10}, d_{11}, d_{12}\}$	
$I_1 \leftarrow$ BldIdx($\{13\}$) $T_0 \leftarrow$ BldIdx($\{6\}$) $T_1 \leftarrow T_0$ AddToIdx($\{5\}, T_1$)	$\{d_{13}\}$	$\{d_4, d_5, d_6\}$	$\{d_7, d_8, d_9\}$	$\{d_{10}, d_{11}, d_{12}\}$	$T_0 = \{d_6\}$ $T_1 = \{d_6, d_5\}$
AddToIdx($\{14\}, I_1$) Drop I_2 Rename T_0 as I_2	$\{d_{13}, d_{14}\}$	$\{d_6, d_5\}$	$\{d_7, d_8, d_9\}$	$\{d_{10}, d_{11}, d_{12}\}$	Same: T_1

Table 8.7: Example of index transitions in RATA

8.4.3 Reindex and Throw Away (RATA)

We now propose a variant of WATA to maintain hard windows. RATA is similar to WATA except that it uses additional temporary indices to simulate deleting old entries. We explain RATA with the example in Table 8.7. In the example, we use the notation T_i for temporary indices that replace some constituent index I_j on day i. On the 10^{th} day, RATA indexes the first ten days in the same way as WATA. In addition, it also builds additional temporary indices, T_{11} and T_{12} so that T_{11} indexes d_3 and d_2, and T_{12} indexes d_3. On day 11, RATA indexes d_{11} in I_4 like WATA. Then it drops I_1 and replaces I_1 with T_{11} which contains entries for d_3 and d_2. The wave index thereby indexes d_2 through d_{11}. Similarly for subsequent days. We present the formal RATA* algorithm based on *WATA** in Appendix A as Figure 10.6. It it easy to see that we can extend RATA to enhance any WATA-based algorithm.

RATA performs more work than WATA but maintains hard windows. However it takes the same time as WATA to index a new day's worth of data after it is available. For instance, on day 13 additional work is done to build temporary indices T_0 and T_1 to be used

on subsequent days. However the operation $T_1 \leftarrow$ `BldIdx({ 6 })` can be performed on day 11 since it depends only on d_6 which is already available on day 11. Similarly, $T_{15} \leftarrow T_{14}$ and `AddToIdx({5},` T_{14}`)` can be performed on day 12 since they depend on T_{14} and d_5, which are available after day 11 and 5 respectively. Hence if we use the above optimization, we would never need to index more than two days of data on any given day. The formal algorithm in Figure 10.6 does not show this optimization to keep the exposition simple.

8.5 Analytic comparison of wave indexing schemes

In the last few sections we proposed six algorithms to build wave indices and three different ways for performing updates with each algorithm. We now present a simple analysis of the schemes. For our analysis, we assume that our n constituent indices are stored on one disk. In case of multiple disks, our analysis can be extended in a similar fashion, but is not shown here. We consider in Section 8.7 a few trends we expect in case of multiple disks.

Since our goal in this section is to identify general trends rather than to predict accurate performance numbers, we now propose some "coarse" parameters to compare our wave indexing schemes. The parameters we propose below are of three types (and sometimes of more than one type): (1) parameters that depend on the *hardware* used (such as disks used), (2) parameters that depend on the specific *application* (such as the average number of *TimedIndexProbes*), and (3) *implementation* parameters that depend on the how certain algorithms are implemented (such as which incremental indexing scheme is used).

1. **Disk Parameters:** Let *seek* be the time to perform one seek. Let $Trans$ be the transfer speed in blocks per second to transfer disk blocks from disk to memory. These are both hardware parameters.

2. **Space Parameters:** For ease of analysis, we assume that the data size of all days is the same. It is easy to extend our analysis for the case of non-uniform data sizes. Let S be the space required to store a packed index of one day. Let S' be the space required to store a non-packed index of one day. We assume that the space required to store a packed index for d days is $S * d$, and the space to store a non-packed index for d days is $S' * d$.

   The parameter S is an application parameter since it depends on the size of data. The

parameter S' depends on the application as well as on the implementation of incremental indexing. In this chapter for concreteness, we assume we index incrementally using the *CONTIGUOUS* scheme of Faloutsos and Jagadish [FJ92]. Essentially, the *CONTIGUOUS* scheme allocates contiguous space for each search value. Each new index entry for a value is appended into the corresponding allocated space. When the allocated space is consumed, the scheme allocates a larger space which is g (growth factor) times larger than the previous space. It then copies over the index entries to the new space, and releases the old space. Similarly for deletion. Different implementations may use different g values and this clearly affects the value of S'.

3. **Constituent Index Operation Parameters:** Let *Add* be the time to incrementally index one day's data. Let *Del* be the time to incrementally delete one day's data from an index. Let *Build* be the time to build an index of one day's data.

   All three depend on the application. Clearly the larger the amount of data in an application, the more expensive is each operation. All three depend on the implementation as well. For instance in *CONTIGUOUS*, if the initial space allocated for a new bucket is small, the time to add and delete is large because a lot of time is spent in copying the old bucket to a new location to allow for future growth.

4. **Update Technique Parameters:** Given an unpacked index for one day, let CP be the time to copy all buckets of that index into memory, and then flush them to another location on disk. Given a packed index for one day, let $SMCP$ be the time to copy all buckets of the index into memory, delete entries with expired timestamps, and then flush packed buckets to another location on disk. Both CP and $SMCP$ depend on the size of the data to be copied, and hence are application parameters.

5. **IndexProbe Parameters:** Given an index for one day, let c be the average size of a bucket (in disk blocks) for some random search value. We assume that the size of the bucket for d days is $d * c$. Let $Probe_{num}$ be the number of *TimedIndexProbes* and $Scan_{num}$ be the number of *TimedSegmentScans* in a day. Recall that *TimedIndexProbes* and *TimedSegmentScans* access between 1 and n constituent indices depending on the specified time ranges. Let $Probe_{idx}$ and $Scan_{idx}$ be the average number of indices a *TimedIndexProbe* and *TimedSegmentScan* access. All the above parameters are application parameters.

Some of the important performance measures we consider for each scheme are:

1. **Space Utilization:** First, we consider how much space is required to store the required window of days, i.e., during system *operation.* We also consider how much additional space is required when a new day is being indexed, i.e., during index *transitions.* This measure helps system administrators in deciding how many disks to buy, for instance.

2. **Query Response Time:** We consider how long it takes to execute *TimedIndexProbes* and *TimedSegmentScans.* In cases where users are sitting at a terminal waiting for a response, it is important to keep this measure low.

3. **Transition Time:** We consider how soon after a new day's data is available it is part of the wave index and ready for querying. In cases like the stock market where decisions may be made based on the new data, it may be critical to keep the transition time low. This measure may not be quite as important in data mining queries which look at general trends, for instance.

4. **Pre-Transition Time:** We consider how much time is spent each day as precomputation in preparing temporary indices. This measure shows how long this precomputation will interfere with user queries.

5. **Total Work:** During the course of the day, we need to index new data, maintain indices and answer a stream of queries. We try to capture the work done by the system during the day into a single number by estimating resources consumed. We believe one good estimate of work done is the time to index a given volume of new data, precompute new indices, and in answering a set of user queries as if they were performed one after the other, without parallelism. For this, we first add the transition time and the pre-transition time. We then add the time to perform $Probe_{num}$ timed probes that access $Probe_{idx}$ indices each, and the time to perform $Scan_{num}$ timed scans that access $Scan_{idx}$ indices each.

In Table 8.8 we show the space utilization of the six algorithms if they are implemented with simple shadow updating. To simplify the equations in the table, we define $X = \frac{W}{n}$ and $Y = \frac{W-1}{n-1}$.

We now consider in detail the first two columns that show maximum and approximate average space required during system operation. We estimate the maximum space as follows:

we compute the maximum number of days indexed in the constituent indices as well as in the temporary indices. We then multiply that number of days by S' (or S in case of *REINDEX*) to obtain the maximum space required. We estimate the average space averaged over the number of transitions in a similar fashion. For instance, we see that *REINDEX+* requires an average of $(W + \frac{X}{2}) * S'$ and a maximum of $(W + \lceil X \rceil - 1) * S'$ space while the system is in operation. This is because the constituent indices in *REINDEX*$^+$ index W days. In addition, in *REINDEX*$^+$ *Temp* indexes at most $\lceil X - 1 \rceil$ days (Figure 10.3). However when averaged over time, *Temp* indexes about $\frac{X}{2}$ days.

REINDEX^{++} also stores W days in its constituent indices. In addition it maintains $\lceil X \rceil$ temporary indices, and each temporary index T_i $(0 \leq i \leq \lceil X \rceil - 1)$ stores i days. That is, the temporary indices will store a maximum of $\frac{1}{2} * \lceil X \rceil * \lceil X - 1 \rceil$ days. The average space for *REINDEX*$^{++}$ can be similary calculated. Similarly, WATA* stores a maximum of $W + \lceil Y - 1 \rceil$ (proved in Appendix B) days since it maintains soft windows. Hence it requires the maximum space indicated in the table. *RATA* maintains temporary indices similar to *REINDEX*$^{++}$ (See Figures 10.4 and 10.6). *RATA* however stores a maximum of $\lceil Y \rceil$ days in a temporary index rather than $\lceil X \rceil$ days like *REINDEX*$^{++}$. Hence the difference between the formulae for *RATA* and*REINDEX*$^{++}$.

We now consider the third and fourth columns in Table 8.8 which report the additional space required during transitions. We estimate the additional space required as follows: if some constituent index needs to be shadowed for updating, we need space to store the shadow index. If some temporary index needs to be updated, we require no additional space since queries are executed only on constituent indices. Hence the maximum additional space required during transitions is the size of the largest constituent index: this can be computed by multiplying the maximum number of days in a constituent index by S' (S in case of *REINDEX*). We estimate average additional space by averaging additional space requirements over the number of transitions. For instance, *REINDEX*$^+$ requires $\lceil X \rceil * S'$ space during transitions since the maximum number of days in an index is $\lceil X \rceil$. *REINDEX*$^{++}$ requires no additional space during a transition since it updates only temporary indices.

The corresponding space utilization table for the algorithms when we use in-place updating (not shown) will look similar to Table 8.8 except that the space required by the six algorithms during index transitions will be zero. This is because no space is required for a shadow index. Similarly, the corresponding table for the algorithms when we use packed

Measure/ Scheme	Max Space [Operation]	Avg Space [Operation]	Max Space [Transition]	Avg Space [Transition]
DEL	$W * S'$	$W * S'$	$\lceil X \rceil * S'$	$X * S'$
REINDEX	$W * S$	$W * S$	$\lceil X \rceil * S$	$X * S$
REINDEX$^+$	$(W + \lceil X \rceil) * S'$	$(W + \frac{X}{2}) * S'$	$\lceil X \rceil * S'$	$\frac{X}{2} * S'$
REINDEX^{++}	$(W + \frac{\lceil X \rceil \lceil X-1 \rceil}{2}) * S'$	$(W + \frac{X^2}{6} + \frac{X}{2}) * S'$	0	0
WATA*	$(W + \lceil Y - 1 \rceil) * S'$	$(W + \frac{Y}{2}) * S'$	$\lceil Y \rceil * S'$	$\frac{Y}{2} * S'$
RATA*	$(W + \frac{\lceil Y \rceil \lceil Y-1 \rceil}{2}) * S'$	$(W + \frac{Y^2}{6} + \frac{Y}{2}) * S'$	$\lceil Y \rceil * S'$	$\frac{Y}{2} * S'$

Table 8.8: Space utilization of wave indices that use simple shadow updating ($X = \frac{W}{n}, Y = \frac{W-1}{n-1}$).

Measure/ Scheme	TimedIndexProbe	TimedSegmentScan
DEL	$Probe_{idx} * (seek + \frac{W}{n} * \frac{c}{Trans})$	$Scan_{idx} * (seek + \frac{W}{n} * \frac{S'}{Trans})$
REINDEX	$Probe_{idx} * (seek + \frac{W}{n} * \frac{c}{Trans})$	$Scan_{idx} * (seek + \frac{W}{n} * \frac{S}{Trans})$
REINDEX$^+$	$Probe_{idx} * (seek + \frac{W}{n} * \frac{c}{Trans})$	$Scan_{idx} * (seek + \frac{W}{n} * \frac{S'}{Trans})$
REINDEX^{++}	$Probe_{idx} * (seek + \frac{W}{n} * \frac{c}{Trans})$	$Scan_{idx} * (seek + \frac{W}{n} * \frac{S'}{Trans})$
WATA*	$Probe_{idx} * (seek + \frac{W}{n} * \frac{c}{Trans})$	$Scan_{idx} * (seek + \frac{W-1+\frac{W-1}{2n-2}}{n} * \frac{S'}{Trans})$
RATA*	$Probe_{idx} * (seek + \frac{W}{n} * \frac{c}{Trans})$	$Scan_{idx} * (seek + \frac{W}{n} * \frac{S'}{Trans})$

Table 8.9: Query performance of wave indices that use simple shadow updating.

shadow updating (not shown) will look similar to Table 8.8, except that all S' will be replaced by S. The only other difference is that *REINDEX*$^{++}$ will now require an average of $X * S$ space since packed shadowing requires temporary indices to be copied to a new location as well.

In Table 8.9, we present the time to perform one *TimedIndexProbe* and one *TimedSegmentScan* for the techniques implemented with simple shadow updating. We estimate the time to perform *TimedIndexProbe* as follows: we first compute the time to perform a probe on one index. This we compute by assuming each probe requires one seek followed by a transfer of the corresponding bucket from disk to memory. We then multiply the time to probe one index by $[1, n]$ to indicate the range of possible times a *TimedIndexProbe* can take depending on the specified time ranges. For instance, *REINDEX*$^{++}$ takes time

Measure/ Scheme	Precomputation	Transition
DEL	$\frac{W}{n} * CP + Del$	Add
REINDEX	0	$\frac{W}{n} * Build$
REINDEX$^+$	0	$\frac{W}{n} * CP + \frac{1}{2} * \frac{W}{n} * Add$
REINDEX^{++}	$\frac{W}{2*n} * Add$	Add
WATA*	0	$\frac{1}{2} * \frac{W-1}{n-1} * CP + Add$
RATA*	$\frac{1}{2} * \frac{W-1}{n-1} * CP + Add$	$\frac{1}{2} * \frac{W-1}{n-1} * CP + Add$

Table 8.10: Maintenance performance of wave indices that use simple shadow updating.

$(seek + \frac{W}{n} * \frac{c}{Trans})$ for probing one index, and therefore $Probe_{idx} * (seek + \frac{W}{n} * \frac{c}{Trans})$ overall.

We estimate *TimedSegmentScan* as follows: we first compute the time to perform a scan of one index. This we compute by assuming each scan requires one seek followed by retrieving buckets of k days, where k is the number of days indexed in the index. Similar to *TimedIndexProbe* we then multiply the time to scan one index by $Scan_{idx}$ to indicate that the actual time depends on the specified time ranges. For instace *REINDEX*$^{++}$ takes time $(seek + \frac{S'}{Trans} * \frac{W}{n})$ to scan one index, and therefore $Scan_{idx} * (seek + \frac{S'}{Trans} * \frac{W}{n})$ overall.

The table for *TimedSegmentScan* and *TimedIndexProbe* for the algorithms implemented with in-place updating looks identical to Table 8.9. The corresponding table for the algorithms implemented with shadow updating also looks similar to Table 8.9 except that all S' are replaced with S.

In Table 8.10 we present the time it takes each day for adding a new day's data (Transition) and to index data in temporary indices for future use (Pre-computation). We first consider transition time. In the table we see that, for example, every day *RATA* copies a temporary index with an average (averaged across time) of $\frac{1}{2} * \frac{W-1}{n-1}$ days to a new location, and adds a new day to the index. This is performed as pre-computation to prepare simulating hard windows for the next few days. Similarly we see for Transition time that every day *RATA* copies a constituent index with an average (averaged across time) of $\frac{1}{2} * \frac{W-1}{n-1}$ days to a shadow location, and adds the new day to the shadow.

The corresponding table for in-place updating looks similar except for fewer copy operations since additions and deletions are in-place. The corresponding table for packed shadow updating is presented in Table 8.11. We see that the time taken by the different algorithms

Measure/ Scheme	Precomputation	Transition
DEL	0	$\frac{W}{n} * SMCP + Build$
REINDEX	0	$\frac{W}{n} * Build$
REINDEX$^+$	0	$\frac{W}{n} * (SMCP + CP) + \frac{1}{2} * \frac{W}{n} * Build$
REINDEX^{++}	$\frac{W}{2*n} * SMCP + \frac{W}{2*n} * Build$	$\frac{1}{2} * \frac{W}{n} * SMCP + Build$
WATA*	0	$\frac{1}{2} * \frac{W-1}{n-1} * CP + Build$
RATA*	$\frac{1}{2} * \frac{W-1}{n-1} * SMCP + Build$	$\frac{1}{2} * \frac{W-1}{n-1} * CP + Build$

Table 8.11: Maintenance performance of wave indexing techniques that use packed shadow updating.

is typically less than in simple shadow updating. This is because operations such as deletion are handled as part of the smart copy operation. Also we can show that with packed shadow updating, the incremental insert operations take time *Build* rather than *Add*.

8.6 Case studies

Given the relatively large number of implementation options, parameters, and performance metrics, it is difficult to draw concrete conclusions without looking at particular applications scenarios. In this section we present three application areas (copy detection, web engines, and warehousing), and within those we instantiate particular scenarios (e.g., data size, hardware speeds). For each scenario there are parameters we could directly measure, for example, how many Netnews articles need to be indexed each day for copy detection. Other parameters could be measured via experiments. For example, we evaluated S' by actually implementing the index algorithms and loading data into an index. (The number we obtain is realistic yet specific to our implementation.) However, other parameters values were "educated guesses," for instance, exactly how many queries to expect each day. Hence, the reader should not interpret the results of this section as absolute predictions, but rather as illustrations of performance trends and of the process to follow in selecting a particular wave index scheme. The scenarios we consider are:

1. **Text CDS:** One of the services we provide in our text CDS is to index articles of a set of newsgroups for a week to allow authors to search for recent illegal copies of

Parameter Type	Parameter	Text CDS	WSE	TPC-D
Hardware	*seek*	14 msec	14 msec	14 msec
	$Trans$	10 MBps	10 MBps	10 MBps
Application	S	56 MB	75* MB	600 MB
	c	100 bytes	100* bytes	100* bytes
	$Probe_{num}$	100,000*	340,000*	0*
	$Probe_{idx}$	n	n	–
	$Scan_{num}$	10*	0*	10*
	$Scan_{idx}$	1	–	n
Implementation	g	2.0	2.0	1.08
(*CONTIGUOUS*)	$Build$	1686 secs	2276 secs	8406 secs
	Add	3341 secs	4678 secs	11431 secs
	Del	3341 secs	4678 secs	11431 secs
	S'	78.4 MB	105* MB	627 MB

Table 8.12: Parameter values chosen in case study.

their articles. In the following experiments, we report results only for the case we implement wave indices using simple shadowing (due to our space constraints).

2. **Web search engine (WSE):** Several WSEs (e.g., Altavista [Alt99], SIFT [YGM95], Infoseek [Inf99] and Dejanews [Dej99]) index Netnews articles in addition to a subset of the World-Wide-Web. We consider how a WSE should index articles for a sliding window of 35 days. In the study of a generic WSE, we report results for the case the indices are implemented with simple shadowing as well as packed shadowing. (In-place updating is similar to simple shadowing.)

3. **TPC-D:** TPC-D is a benchmark from the Transaction Processing Council [TPC99]. The benchmark models a decision support environment in which complex business-oriented queries are submitted against a large database. The queries may access large portions of the database and typically involve various operations such as joins, sorting and aggregation that may implemented with sequential scans and index probes. The benchmark defines two large relations *LINEITEM* and *ORDER*, and six other smaller relations. Similarly 17 queries have been prescribed.

   To simplify our experiments, we consider the following specific scenario. Say we build a wave index on relation *LINEITEM* on the *SUPPKEY* attribute for a window of the

past 100 days. Every day the new additions to *LINEITEM* arrive as a batch based on the sales of the day. Let query $Q1$ (specified in the TPC-D benchmark as the "Pricing Summary Report") be the only query that is executed. In our experiments we used the data characteristics (in terms of distribution of tuples, sizes of tables, etc.) prescribed by the TPC-D benchmark. In the following experiments for TPC-D, we report results for the case the indices are implemented with simple shadowing.

In Table 8.12 we report specific values we used for different parameters in our case study. The hardware parameters were chosen based on current technology. The application parameters we report are for data of one day. As stated earlier, we chose specific values for application parameters either based on experience, or based on educated guesses (denoted in the table with a $*$). As an example of the former, we computed S for our text CDS by building a packed index on about $70,000$ text articles (in a day) and computed the space required. As an example of a guess, we estimated that commercial WSEs index about 100,000 articles per day. (Our text CDS indexes fewer since our NNTP server subscribes to fewer newsgroups).

We chose implementation parameters for our text CDS as follows. First we implemented the *BldIdx* scheme (as specified in Section 8.2.2) in C, and measured its running time on a DEC 3000 with an Alpha processor running OSF/1.0 and 96 MB of RAM. We then implemented and measured *AddToIdx* using the *CONTIGUOUS* [FJ92] incremental indexing scheme. To choose a good value for g in *CONTIGUOUS*, we executed *AddToIdx* to index words of one day's Netnews articles for several values of g. Based on the trade off between space consumption, S', and the time spent in copying buckets to new locations, we chose $g = 2$. For $g = 2$, we report S' and *Add* in Table 8.12. Since *DeleteFromIndex* is symmetric to *AddToIdx*, we assume that *Del* takes the same time as *Add*. The time to execute *BldIdx* on the Netnews data is reported as $Build$.

In our text CDS we expect to service about 100 user queries each day from authors and publishers to check if a given document was available as a Netnews article in the past week. Since for each query we expect to perform 100 *TimedIndexProbes* [SGM96] on the data of the last week, $Probe_{num} = 100,000$ and $Probe_{idx} = n$ ($W = 7$). Our CDS also offers a *registration* service in which authors submit documents so they can be checked on a daily basis against the current day's Netnews articles. We can check the submitted documents against the current day's articles efficiently with a scan on the current day's index. We estimate (based on expected size of registration database) that we will need to perform

about 10 segment scans each day on the current day's index (stored in one index). Hence $Scan_{num} = 10$ and $Scan_{idx} = 1$.

For the WSE, we estimated application and implementation values by scaling the corresponding values in the CDS by $100,000/70,000$ (based on relative number of articles). In a WSE, we expect about $170,000$ queries in a day for Netnews articles. This is roughly 1% of the number of queries per day in Altavista for the more popular web data [Alt99]. Since each user query performs an average of two index probes (average length of a query is two words [Alt99]) over all data in the window, we estimate $Probe_{num} = 340,000$ and $Probe_{idx} = n$.

For TPC-D, we repeated the experiments we did for the CDS and chose $g = 1.08$. This is because values for *SUPPKEY* in TPC-D are uniformly distributed, while words in the CDS Netnews articles exhibit skewed Zipfian [Zip49] behavior. We assume about 10 complex analytical queries are run every day over data of the entire window to analyze trends. We assume these queries are executed using a scan over all the indices, and therefore $Scan_{num} = 10$ and $Scan_{idx} = n$.

We now present a few select graphs to indicate how the wave indices perform in our text CDS, WSE and TPC-D. As we describe these graphs, keep in mind that they illustrate performance metrics (e.g., space, work) and not qualitative measures such as ease of implementation. Recall that even if a scheme outperforms the other others in a given scenario, it may not be advisable either because (1) it requires complex code, or (2) it cannot be implemented with our favorite index package.

In Figure 8.3 we report the overall space required (averaged across transitions) by the CDS during system operation and transition (sum of column 2 and 4 in Table 8.3). We see that *REINDEX* requires the minimal amount of space. This is because (1) *REINDEX* maintains packed indices that consume minimal space, and (2) *REINDEX* does not have any additional temporary indices like $REINDEX^+$, $REINDEX^{++}$ or *RATA*. We also see that all schemes require less space as n increases. This is because each constituent index stores fewer days as n increases. Hence shadow indices are smaller during transitions. Also in schemes like $REINDEX^+$, $REINDEX^{++}$ and *RATA*, there are fewer days in each temporary index as n increases. In schemes like *WATA* and *RATA*, the number of days in the soft window also decreases as n increases.

In Figure 8.4 we report the transition time to index new data in our CDS (column 2 of Table 8.4). There are two main factors that influence transition time: (1) does the

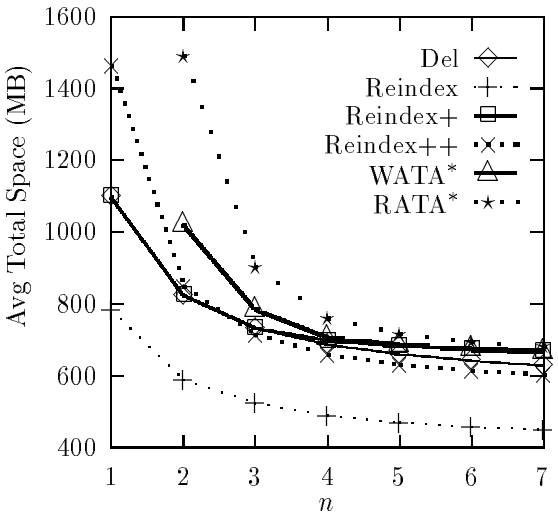


Figure 8.3: Average space required during CDS' operation and transition (W = 7).

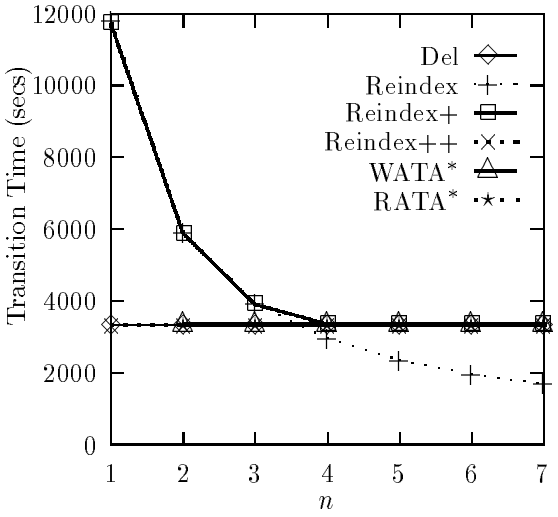


Figure 8.4: Average transition time for CDS (W = 7).

scheme use *BldIdx* or *AddToIdx* to add the new data? (2) for each scheme, how many days are reindexed using *BldIdx* or incrementally indexed using *AddToIdx*? For instance, from Table 8.12 we see that if a scheme executes *BldIdx* for one day, its transition time (1686 secs) is lower than another scheme that indexes *AddToIdx* (3341 secs) for the same day. However if the first scheme executes *BldIdx* for 5 days, its transition time ($1686 * 5$ secs) is higher than the second scheme (3341 secs). Since *DEL*, *WATA*, *RATA* and *REINDEX*$^{++}$ execute *AddToIdx* during transitions and always incrementally index one day, we see that their transition times do not depend on n. However recall that *REINDEX* executes *BldIdx* on $\frac{W}{n}$ days each day, which clearly depends on n. Hence we see that initially ($n \leq 3$) *REINDEX* performs poorly due to the cost of reindexing $\frac{W}{n}$ days each day. But for $n \geq 4$, the cost savings of executing a *BldIdx* rather than an *AddToIdx* compensates for the cost of reindexing 1 or 2 days each day. *REINDEX*$^{+}$ performs the worst since it executes *AddToIdx* on an average of $\frac{1}{2} * \frac{7}{n}$ days each day.

In Figure 8.5, we report the total work done during the day by the different schemes in our CDS. The total work is very sensitive to the mix of queries and updates. For example, if we have many queries in a day, it is best to perform more work at update time in order to obtain an index that is better for queries (e.g., packed, small n). In the CDS scenario, the opposite is true: the number of copy detection queries is relatively small compared to the number of documents indexed.

In Figure 8.5 again we see that *REINDEX* performs poorly for small n but is very efficient for large n. This is because of the relative cost of reindexing some constituent index each day versus the savings due to using *BldIdx*, and faster scans due to packed indices. We see from the figure that the reindexing cost in *REINDEX* dominates for small n, while for large n the savings dominate. We also see that *DEL*, *WATA* and *RATA* are relatively stable since they incrementally add and delete a small constant number of days each day. They increase slowly with n since *TimedIndexProbes* need to probe an increasing number of indices.

From Figures 8.3, 8.4 and 8.5, we recommend using *REINDEX* for the CDS with $n = 4$ indices. We recommend $n = 4$ as a compromise value between the following two conflicting factors: (1) as n increases, *REINDEX* performs better than the other schemes and (2) as n increases, the response time of *TimedIndexProbes* increases since more constituent indices need to be probed. We choose $n = 4$ since we would like to keep the user response time low, and since we see from the graphs that we obtain diminishing returns for our performance

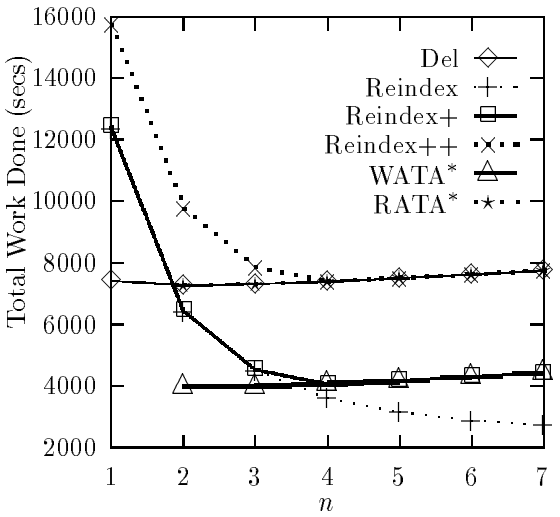


Figure 8.5: Average work done by CDS during day (W = 7).

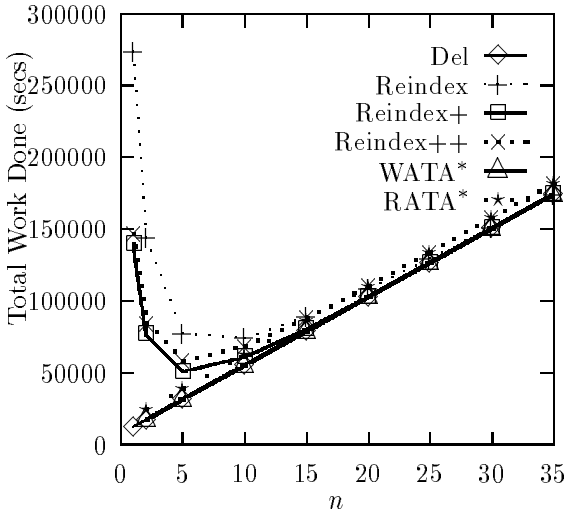


Figure 8.6: Average work done by WSE during day (W = 35).

measures for $n \geq 4$.

We now consider the performance of our wave indices for WSE. We observed trends similar to Figure 8.3 and 8.4 for the average space during transitions and average transition time for WSE as well as TPC-D (not reported). In Figure 8.6, we report the total work done by WSE with packed shadowing for $W = 35$. We see that due to significantly higher query volume and window size, *REINDEX* that performed best in the CDS, now in fact performs the worst. *REINDEX* does poorly for small n for the reasons described earlier. But *REINDEX* continues to do poorly even as n increases since the cost savings of reindexing fewer days in a constituent index is offset by the increased cost of more probes executed for a *TimedIndexProbe*. In this case *DEL*, *WATA* and *RATA* perform the minimal amount of work when $n \leq 2$. This is because they always perform minimal amount of work in indexing new data, and also because n is small enough to service *TimedIndexProbes* cheaply.

From Figure 8.6, we recommend using *DEL* ($n = 1$) with packed shadow updating for a WSE. This is because for $n = 1$, the response time for user queries is low. Also, *DEL* performs minimal total work.

Similarly in Figure 8.7 we report the total work done by the different algorithms in the TPC-D case when packed shadowing is used. (We resized the graph since *REINDEX* performs very poorly.) In this example we see again that *DEL* ($n = 1$) and *WATA** ($n = 2$) perform the best, while *REINDEX* performs the worst. In Figure 8.8, we report the total work done by the different algorithms in the TPC-D case when simple shadowing is used. While we see similar trends to Figure 8.7, we see how the work done is significantly less in case of packed shadowing. This is of course because packed shadowing does deletion while copying, and because segment scans are efficient due to the packed constituent indices. For simple shadowing, we see that *WATA* performs the minimal amount of work among the schemes, and performs less work as n increases. This is because the number of expired days stored in the constituent indices decreases as n increases, and segment scans are more efficient. Also we would like to point out that *WATA* performs significantly better than *DEL* and *RATA*: *WATA* requires upto $10,000$ seconds (about 3 hours) less time than both *DEL* and *RATA*. This is not clear from the graph due to the ranges displayed in the vertical axis. In-place updating of course performs like simple shadowing in all measures except it uses less space during index transitions, and is more complex to implement.

From Figures 8.7 and 8.8 we recommend the following schemes (in order of preference) to be used for TPC-D. If packed shadowing can be implemented, use *DEL* ($n = 1$) since it has

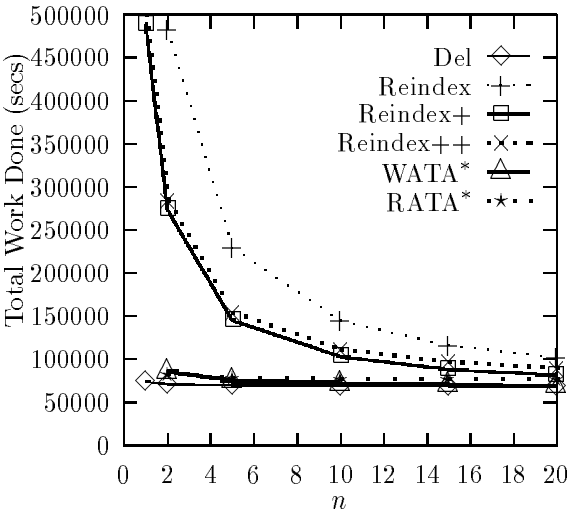


Figure 8.7: Work done in TPC-D (packed shadowing) during day (W = 100).

the best user response time and since it performs minimal work. If packed shadowing cannot be implemented (since some legacy system needs to be used), implement *WATA* ($n = 10$). This is because it performs significantly less work (about $9,000$ seconds worth) than *DEL*. Beyond $n \geq 10$ the savings in *WATA* are marginal while increasing query response time. If hard windows are required, we recommend *RATA* ($n = 10$) since it performs the same work as *DEL*, and is not as complex to implement as *DEL*.

In Figure 8.9 we consider the question of how the schemes scale when the required window size increases from 4 days to 6 weeks. Recall that the reindexing schemes index $O(\frac{W}{n})$ days each day, while *DEL*, *WATA* and *RATA* index a small constant number of days each day. Hence we see that, for a given n, as W increases the three reindexing based schemes do not scale while *DEL*, *WATA* and *RATA* scale very well. So if in the CDS we expect to index (say) a window of 14 days some time in the future, it may be worth the effort now to implement *WATA** rather than *REINDEX*.

If we did expect to index a window of 14 days in the future, we need to consider how much data may have increased by then. In Figure 8.10 we consider the case in the CDS when the number of netnews articles per day increases from $70,000$ to $70,000 * SF$, where $0.5 \leq SF \leq 5$ is the scale factor. We see that *REINDEX* scales the best for this measure since it does not use expensive incremental indexing schemes like *CONTIGUOUS*. However, *WATA** still performs best when $SF \leq 3$. So if we expect the data in the future to increase

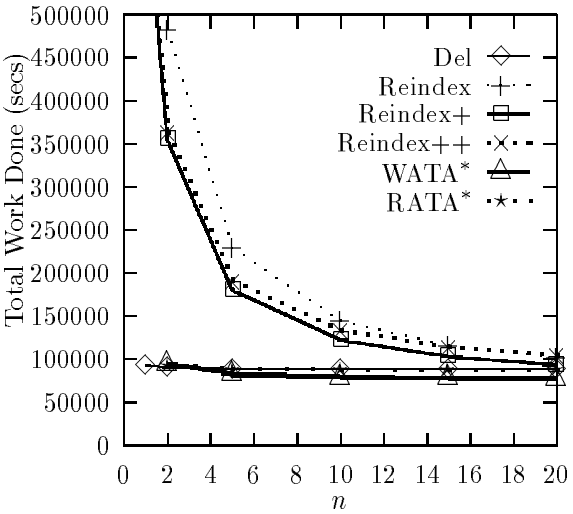


Figure 8.8: Work done in TPC-D (simple shadowing) during day (W = 100).

significantly (i.e., the number of Netnews articles per day becomes $\geq 70,000 * 3$), it may be actually be best to implement *REINDEX* rather than *WATA**! This shows us that before choosing a particular scheme to implement we should consider carefully both (1) whether we may ever want a larger window size, and (2) if so, how much do we expect data to increase by.

Finally we consider how our WATA* scheme performs when we index 200 days worth of Usenet data, collected between June and December 1997. The purpose of this experiment is to understand how much space overhead the WATA* scheme incurs to support lazy deletion. Specifically we are interested in the *index size* ratio, which we define to be the maximum index size ever required by the lazy WATA* scheme divided by the maximum index size ever required if we use eager deletion strategies such as REINDEX. We report this ratio in Figure 8.11 for the 200 days of data, as n varies for $W = 7$. For instance, when $n = 4$ the index size ratio is 1.24 which indicates the WATA* scheme costs 24% of overhead in storage. We note that the space overhead for WATA* appears tolerable (≤ 1.6), and decreases as n increases – we believe this makes the case stronger for WATA based indexing.

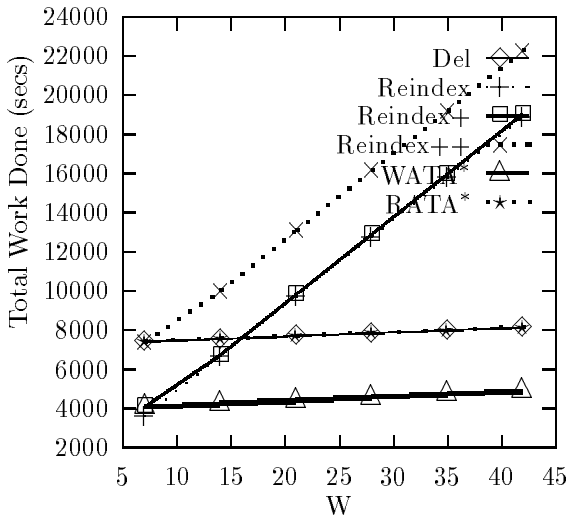


Figure 8.9: Work done during day by CDS with W (n = 4).

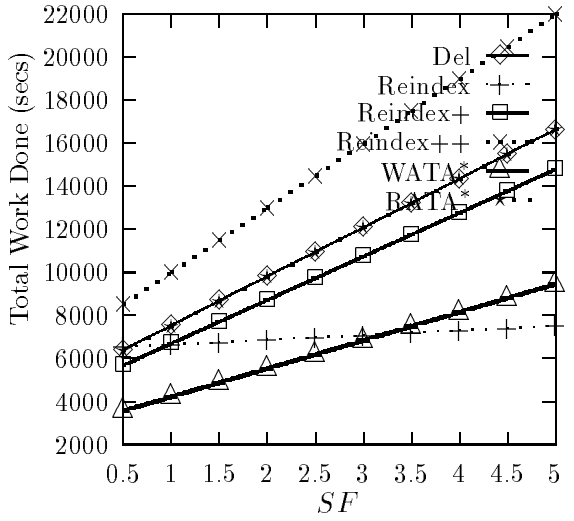


Figure 8.10: Work done during day by CDS with SF (W = 14, n = 4).

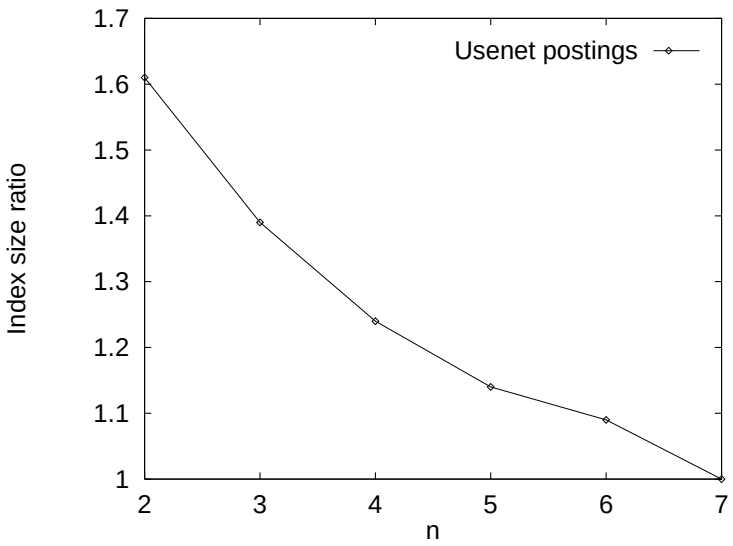


Figure 8.11: Index size ratio with n (W = 7).

8.7 Conclusion

We consider the problem of organizing index structures when content publishers register their content for a fixed period of time, e.g., a week. We proposed several techniques to build *wave indices*. We then analyzed these schemes and showed experimentally under a variety of scenarios how the schemes perform for different volumes of input data and query patterns. We also note that our techniques are applicable in a wide range of other applications, and evaluate our techniques in some of these scenarios.

Chapter 9

Conclusions and Future Work

Digital content publishers expect to lose significant revenues due to cyber-piracy. One approach to combat cyber-piracy is to employ a copy detection system (CDS) that *polices* the web and finds potential copies of a publisher's content. In this thesis, we discussed how to build CDS systems that *scale* as the web grows, are *accurate* in finding potential copies, are *resilient* to attacks from a cyber-pirate, and are *flexible* enough to service a variety of different user requirements.

Some of the contributions of this thesis include:

1. **Algorithms:** We designed several algorithms that are crucial in building CDS systems. In some cases, we also observed that our algorithms solve more general problems, and are not limited only to copy detection. For example, the iceberg strategies from Chapter 3 are applicable in a variety of data-intensive applications, including data mining and online analytical processing (OLAP). Also, our filtering strategies (Chapter 7) and our wave indexing algorithms (Chapter 8) are applicable in several domains as discussed in these chapters.

2. **Features:** For specific domains such as text (Chapter 2) and video (Chapter 6), we discussed what are good features to extract from content, and how to compare content.

3. **Evaluation:** In all cases, we showed empirically or analytically that our algorithms scale for large amounts of data and our similarity functions perform well in practice. For instance, in Chapter 4 we showed that our text CDS scales to over 60 million

pages and several hundreds of gigabytes of text data. Also in Chapter 5, we discussed a real life plagiarism case where our techniques helped catch a plagiarist.

4. **Prototypes:** As proofs-of-concept, we have built SCAM (Stanford Copy Analysis Mechanism) and DECEIVE (Detecting Copies of Internet Video) for finding text and video copies. These prototypes implement the algorithms we introduced in this dissertation. In addition, we discussed in Chapter 5 the PROBE-CDS prototype that automates the plagiarism detection process. These prototypes were used as a testbed to evaluate our algorithms and features in a realistic environment.

9.1 Future Work

There are several directions for future work. Earlier, we briefly discussed one way to build a content delivery system that uses copy protection, watermarking, and copy detection to protect digital content. In general, a content delivery system should be designed and implemented based on the level of security the content provider requires. For instance, a person with little valuable content may offer the content on a unprotected web site. On the other hand, Knight-Ridder may offer limited access to its customers for its valuable Dialog databases. For content of intermediate value, we can use one or more of Cryptolope-like solutions, watermarking or simple crypto-secure links depending on the security desired by the content provider. One promising area of research is to design content delivery systems based on a variety of considerations such as (1) level of security offered to provider, (2) ease of use to customers, (3) *terms-and-conditions* supported by the infrastructure, as well as (4) the cost of building the system and how it scales.

In this dissertation, we designed media-specific features and comparison functions so that the CDS is resilient to attacks from a cyber-pirate. However, our claims of resiliency were backed by experiments rather than by formal proofs. A useful research direction is to formalize *attack models* so that we can formally prove or disprove if a CDS is resilient to a specific attack. For example, we addressed the problem of building a video CDS in a two-step process. We considered potential attacks on video content and observed that all these attacks induced precision and detection errors. Then we proposed index structures that handled such errors, thereby making the CDS resilient to attacks. We believe that a formal attack madel for specific media would be useful.

In Chapter 7, we discussed query optimization algorithms for automatically composing

approximate predicates. As we mentioned earlier, a promising research area is to carefully evaluate additional classes of filters, such as saw-tooth filters and OR-based filters.

In Chapter 8, we discussed how to construct temporal indices. A useful direction for further research is to consider how the different wave indices perform when multiple disks are used. In particular, if n matches the number of disks, indexing can be parallelized easily. Also building new constituent indices on separate disks avoids contention. Since there are several interesting ways in which a given number of disks can be allocated to the constituent indices, we are planning to evaluate some of the tradeoffs.

Chapter 10

Appendix A: Wave Indexing Algorithms

Each of the following algorithms have two important states, **Start** and **Transition**. Initially when the wave index is to be built for the 1^{st} W days, the operations in **Start** will be executed. Each subsequent day, the new day's data is indexed using the operations in **Transition**.

Algorithm for *DEL*$[W, n$**: Integer]**

- **Globals:**
  - $I_1, I_2, \ldots, I_n$: Index// `Constituent indices`
  - *Days* : Array$[1..n]$ of Set of Integers // `Time-sets of indices`
- **Start**$[d_1, d_2, \ldots, d_W$: Data]
  Local Variables: low, i, j: Integer
  1. $low = 1$
  2. For all $i = 1, 2, \ldots, W \ mod \ n$
     // 1^{st} $W \ mod \ n$ `time-sets have` $\lceil \frac{W}{n} \rceil$ `days`
     (a) $Days[i] \leftarrow \cup_{j=low}^{low+\lceil W/n \rceil} \{j\}$
     (b) $I_i =$*BldIdx*$(Days[i])$
     (c) $low = low + \lceil W/n \rceil + 1$
  3. For all $i = W \ mod \ n + 1, W \ mod \ n + 2, \ldots, n$
     // `Other time-sets have` $\lfloor \frac{W}{n} \rfloor$ `days`
     (a) $Days[i] \leftarrow \cup_{j=low}^{low+\lfloor W/n \rfloor} \{j\}$
     (b) $I_i =$*BldIdx*$(Days[i])$
     (c) $low = low + \lfloor W/n \rfloor + 1$
- **Transition**$[d_{new}$: Data]
  1. Let I_j be the index containing data of d_{new-W}.
  2. *DeleteFromIndex*(d_{new-W}, I_j)
  3. *AddToIdx*(d_{new}, I_j)

Figure 10.1: Algorithm for *DEL*

Algorithm for *REINDEX*$[W, n$**: Integer]**

- **Globals:**
  - $I_1, I_2, \ldots, I_n$: Index // `Constituent indices`
  - $Days$: Array$[1..n]$ of Set of Integers // `Time-sets of indices`
- **Start**$[d_1, d_2, \ldots, d_W$: Data]
  1. Same as Start for *DEL*
- **Transition**$[d_{new}$: Data]
  Local Variables: j: Integer
  1. Let I_j be the index containing data of d_{new-W}.
  2. $Days[j] = Days[j] - \{new - W\} \cup \{new\}$ // `Updating` j^{th} `time-set`
  3. $I_j \leftarrow BldIdx(Days[j])$ // `Rebuilding index`

Figure 10.2: Algorithm for *REINDEX*

Algorithm for $REINDEX^+[W, n$: **Integer**]

- **Globals**
  - $I_1, I_2, \ldots, I_n$: Index `// Constituent indices`
  - $Temp$: Index `// Temporary index`
  - $Days$: Array[1..n] of Set of Integers `// Time-sets of indices`
  - $DaysToAdd$: Set of Integers `// Tracks days to add to temp index`
- **Start**[$d_1, d_2, \ldots, d_W$: Data]
  1. Same as DEL
  2. $Temp \leftarrow \phi$
- **Transition**[d_{new}: Data]
  Local Variables: j: Integer
  1. Let I_j be the index containing data of d_{new-W}
  2. If $Temp = \phi$
     `// As in days` 11 `and` 16 `in Table 8.5`
     (a) $DaysToAdd \leftarrow Days[j] - \{new - W\}$
     (b) $Temp$, $I_j \leftarrow BldIdx(d_{new})$
     (c) $AddToIdx(DaysToAdd, I_j)$
  3. Else If $DaysToAdd = \phi$
     `// As in day` 15 `in Table 8.5`
     (a) $I_j \leftarrow Temp$
     (b) $AddToIdx(d_{new}, I_j)$
     (c) $Temp \leftarrow \phi$
  4. Else
     `// As in days` 12, 13, 14 `in Table 8.5`
     (a) $AddToIdx(d_{new}, Temp)$
     (b) $I_j \leftarrow Temp$
     (c) $AddToIdx(DaysToAdd, I_j)$
  5. $Days[j] \leftarrow Days[j] - \{new - W\} \cup \{new\}$ `// Updating` j^{th} `time-set`
  6. $DaysToAdd \leftarrow DaysToAdd - \{new - W + 1\}$

Figure 10.3: Algorithm for REINDEX$^+$

Algorithm for $REINDEX^{++}[W, n$: **Integer**]

- **Globals**
  - $I_1, I_2, \ldots, I_n$: Index `// Constituent indices`
  - $T_1, T_2, \ldots, T_{\lceil W/n \rceil}$: Index `// Temporary indices`
  - $Days$: Array$[1..n]$ of Set of Integers `// Time-sets of indices`
  - $DaysToAdd$: Set of Integers `// Tracks days to be added to temp indices`
  - $TempUsed$: Integer `// Tracks next temp index that replaces a constituent index`
- **Initialize**$[\cup_{l=j}^{k}\{d_l\}$: Set of Integers]

  Local Variables: i: Integer
  1. $T_0 \leftarrow \phi$, $T_1 \leftarrow BldIdx(d_k)$
  2. For i from $j+1$ to k
     (a) $T_{i-j+1} \leftarrow T_{i-j}$
     (b) $AddToIdx(d_{k-(i-j)}, T_{i-j+1})$
  3. $TempUsed = k - j + 1$
  4. $DaysToAdd \rightarrow \phi$
- **Start**$[d_1, d_2, \ldots, d_W$: Data]
  1. Same as *DEL*
  2. *Initialize(Days[1] - {1})*
- **Transition**$[d_{new}$: Data]

  Local Variables: j, j': Integer
  1. Let I_j be the index containing data of d_{new-W}
  2. If $TempUsed = 0$ `// As in day 10 and 15 in Table 8.6`
     (a) $AddToIdx(d_{new}, T_0)$
     (b) Rename T_0 as I_j
     (c) Let $I_{j'}$ be the index containing data of $d_{new-W+1}$
     (d) $Initialize(\text{Days}[j'] - \{new - W + 1\})$
  3. Else `// As in days 11, 12, 13, 14 in Table 8.6`
     (a) $DaysToAdd \leftarrow DaysToAdd \cup \{new\}$
     (b) $AddToIdx(d_{new}, T_{TempUsed})$
     (c) Rename $T_{TempUsed}$ as I_j
     (d) $TempUsed = TempUsed$ - 1
     (e) $AddToIdx(DaysToAdd, T_{TempUsed})$
  4. $Days[j] = Days[j] - \{new - W\} \cup \{new\}$

Figure 10.4: Algorithm for $REINDEX^{++}$

Algorithm for $WATA^*[W, n$: **Integer]**

- **Globals:**
  - $I_1, I_2, \ldots, I_n$: Index // `Constituent indices`
  - $Days$: Array[1..n] of Set of Integers // `Time-sets of indices`
  - Z: Array[1..n] of Integers // `Sizes of indices`
  - $last$: Integer // `Tracks last modified index`
- **Start**[$d_1, d_2, \ldots, d_W$: Data]
  Local Variables: low, i, j: Integer
  1. $low = 1$
  2. For all $i = 1, 2, \ldots, (W-1)\ mod\ (n-1)$,
     // 1^{st} $(W-1) mod (n-1)$ `time-sets have` $\lceil \frac{W-1}{n-1} \rceil$ `days`
     (a) $Days[i] \leftarrow \cup_{j=low}^{low+\lceil (W-1)/(n-1) \rceil} \{j\}$
     (b) I_i =$BldIdx(Days[i])$
     (c) $low = low + \lceil (W-1)/(n-1) \rceil + 1$
     (d) $Z_i = \lceil (W-1)/(n-1) \rceil$
  3. For all $i = (W-1)\ mod\ (n-1)+1, (W-1)\ mod\ (n-1)+2, \ldots, n-1$,
     // `Other time-sets till` $n-1$ `have` $\lfloor \frac{W-1}{n-1} \rfloor$ `days`
     (a) $Days[i] \leftarrow \cup_{j=low}^{low+\lfloor (W-1)/(n-1) \rfloor} \{j\}$
     (b) I_i =$BldIdx(Days[i])$
     (c) $low = low + \lfloor (W-1)/(n-1) \rfloor + 1$
     (d) $Z_i = \lfloor (W-1)/(n-1) \rfloor$
  4. $Days[i] \leftarrow \{W\}$ // `Last time-set has` W^{th} `day`
  5. $I_n \leftarrow$BldIdx(b_W)
  6. $last = n$
- **Transition**[d_{new}: Data]
  1. Let I_j be the index containing data of d_{new-W}
  2. If $\sum_{i=1, i \neq j}^{n} Z_i = W-1$, perform *ThrowAway* else perform *Wait*.
     (a) *ThrowAway:* // `Throw away` j^{th} `index`
        i. $DropIndex(I_j)$
        ii. $I_j \leftarrow \phi$
        iii. $I_j \leftarrow BldIdx(new)$
        iv. $Days[j] \leftarrow \{new\}$, $Z_j = 1$
        v. $last = j$
     (b) *Wait:* // `Add new day to last modified index`
        i. $AddToIdx(d_{new}, I_{last})$
        ii. $Z_{last} = Z_{last} + 1$
        iii. $Days[last] = Days[last] \cup \{new\}$

Figure 10.5: Algorithm for $WATA^*$

Algorithm for $RATA^*[W, n$: **Integer]**

- **Globals:**
  - $I_1, I_2, \ldots, I_n$: Index // `Constituent indices`
  - $Days$: Array[1..n] of Set of Integers // `Time-sets of indices`
  - Z[1..n]: Integer // `Sizes of indices`
  - $TempUsed$: Integer // `Tracks next temp index that will replace a constituent index`
  - $last$: Integer // `Tracks last modified index`
- **Initialize**[$\cup_{l=j}^{k}\{d_l\}$: Set of Integers]
  Local Variables: i: Integer
  1. $T_1 \leftarrow BldIdx(d_k)$
  2. For i from $j+1$ to k
     (a) $T_{i-j+1} \leftarrow T_{i-j}$
     (b) $AddToIdx(d_{k-(i-j)}, T_{i-j+1})$
  3. $TempUsed = k - j + 1$
- **Start**[$d_1, d_2, \ldots, d_W$: Data]
  Local Variables: low, i, j: Integer
  1. Same as $WATA^*$
  2. $Initialize(Days[1] - \{1\})$
- **Transition**[d_{new}: Data]
  Local Variables: j': Integer
  1. Let I_j be the index containing data of d_{new-W}
  2. If $\sum_{i=1, i\neq j}^{n} Z_i = W - 1$, perform *ThrowAway* else perform *Wait*.
     (a) *ThrowAway:*
        i. Same steps as *ThrowAway* in $WATA^*$
        ii. Let $I_{j'}$ be the index containing data of $d_{new-W+1}$
        iii. Initialize($Days[j'] - \{new - W + 1\}$) // `Preparing temporary indices for next cycle`
     (b) *Wait:*
        i. $AddToIdx(d_{new}, I_{last})$
        ii. $Days[last] = Days[last] \cup \{new\}$
        iii. Drop I_1
        iv. Rename $T_{TempUsed}$ as I_j // `Using temporary index to simulate hard window`
        v. $Days[j] = Days[j] - \{new - W\}$
        vi. $TempUsed =$ $TempUsed$ - 1

Figure 10.6: Algorithm for $RATA^*$

Chapter 11

Appendix B: Efficacy of WATA*

We define family $\mathcal{F}$ to be the set of *WATA*-based algorithms that construct and maintain a wave index, Θ, for W days. Recall that these algorithms use only the following operations: (1) *AddToIdx*, to add a day to a constituent index, (2) *DropIndex*, to remove a constituent index from Θ, and (3) *AddIndex*, to add a constituent index to Θ.

Since constituent indices change each day, we use $I_j^{(i)}$ to refer to index I_j on day i. We define $|I_j^{(i)}|$ to be the number of days indexed in I_j, $j = 1, 2, \ldots, n$ on day i. We use $s(I_J^{(i)})$ to denote the corresponding storage required by the index on day i.

Index length measure

We define *length(T)* of Θ on day i to be the total number of days indexed in the constituent indices on day i, i.e., $\sum_{j=1}^{n} |I_j^{(i)}|$. We define *max length* of Θ to be the maximum length of Θ during its life-time, i.e., $max_{i=0}^{\infty} length(i)$. We define *residual length* of Θ to be *max length* – W. We define *waste*, $w(I_j^{(i)})$, for each constituent index I_j, $j = 1, 2, \ldots, n$, to be the number of days indexed in I_j that are older than the required window on day i.

We now prove that *WATA** minimizes max length of Θ for a given W and n, among the set of algorithms in $\mathcal{F}$. We do this in two steps. First we consider an abstract algorithm, *OPT*, in $\mathcal{F}$ that minimizes the max length of Θ: we compute the lower bound on max length for such an optimal algorithm in Theorem 1. Then we show in Theorem 2 that *WATA** achieves the same lower bound, thereby making it optimal.

Theorem 1: Let *OPT* be an algorithm in $\mathcal{F}$ that minimizes the max length of Θ. Algorithm *OPT* cannot build a Θ for a given W and n with maximum length less than

$W + \lceil \frac{W-1}{n-1} \rceil - 1$.

Proof of Theorem 1: Let C be the residual length of OPT for Θ. By definition, on some day, k, the number of days indexed in Θ by OPT will be $W + C$. Since these $W + C$ days need to be split across the n constituent indices, there must be some index I_{big}, $1 \leq big \leq n$ with

$$|I_{big}^{(k)}| \geq \frac{W + C}{n} \tag{11.0.1}$$

Also, by definition

$$\begin{aligned} C &= max_{t=1}^{\infty} \sum_{j=1}^{n} w(I_j^{(t)}) && (11.0.2) \\ &\geq w(I_{big}^{(t)}), t \geq 1 && (11.0.3) \end{aligned}$$

Observe that every index I_j has to be dropped at some point in time. This is clear since C will otherwise be ∞. Now consider the earliest day, l, ($l \geq k$) I_{big} is ready to be dropped. On the previous day, $l-1$, $w(I_{big}^{(l-1)})$ is $|I_{big}^{(l)}| - 1$ since there is exactly one day in $I_{big}^{(l-1)}$ that does not expire until the next day.

Since I_{big} grows monotonically between day k and day $l-1$, we see that

$$\begin{aligned} C &\geq w(I_{big}^{(l-1)}) && (11.0.4) \\ &= |I_{big}^{(l-1)}| - 1 && (11.0.5) \\ &\geq |I_{big}^{(k)}| - 1 && (11.0.6) \\ &\geq \frac{W + C}{n} - 1 && (11.0.7) \\ n * (C + 1) &\geq W + C && (11.0.8) \\ C &\geq \frac{W - n}{n - 1} && (11.0.9) \\ &= \frac{(W - 1) - (n - 1)}{n - 1} && (11.0.10) \\ &= \frac{W - 1}{n - 1} - 1 && (11.0.11) \end{aligned}$$

That is, no algorithm in $\mathcal{F}$ can have a wave index with max length below $W + \frac{W-1}{n-1} - 1$. Since the max length of Θ should be an integer (number of days is an integer), no algorithm

in $\mathcal{F}$ can have a max length below $W + \lceil \frac{W-1}{n-1} - 1 \rceil$. □

Theorem 2: The maximum length of *WATA** (Figure 10.5) is $W + \lceil \frac{W-1}{n-1} - 1 \rceil$.

Proof of Theorem 2:

The time-sets constructed by *OPT* have consecutive days. That is, if days i and $i+2$ are assigned to some I_j, then $i+1$ is also assigned to I_j. Also *WATA** drops a constituent index when all days in its time-set has expired. Hence if we maintain a window of W consecutive days, there can be at most one I_j, such that $w(I_j) > 0$.

Also when the wave index is constructed initially, the number of days in each constituent index is at most $\lceil \frac{W-1}{n-1} \rceil$. Observe that *WATA** maintains this throughout the life-time of the wave-index.

Consider the day before some I_j is to be thrown away. At that point, only one day is alive and the other $|I_j| - 1$ have expired. Hence the maximum $w(I_j)$ is $\lceil \frac{W-1}{n-1} \rceil - 1$. Therefore the maximum length of any wave index constructed using *WATA** is $W + \lceil \frac{W-1}{n-1} \rceil - 1$. □

Index size measure

Theorem 3 *WATA** has a competitive ratio of 2.0 with respect to optimal among the algorithms in $\mathcal{F}$.

Proof of Theorem 3: Let M be the maximum value of index size required to store W consecutive days of data, across the entire duration of indexing. Clearly, the optimal algorithm *OPT* requires at least M storage space to handle the largest window. During this period, WATA* requires M storage as well to store the W days. In addition, WATA* may require an additional S storage to store the residual days that may have expired but are in the same index as some day that has not yet expired. However, this space S clearly cannot exceed M, since M is the maximum storage required for any time window W. Hence $S \leq M$. Therefore, WATA* has a competitive ratio of 2.0.

Bibliography

[ACM96] Edoardo Ardizzone, Marco La Cascia, and D. Molinelli. Motion and color based video indexing and retrieval. In *International Conference on Pattern Recognition (ICPR'*, pages 135 – 139, August 1996.

[Ale99] Alexa Corporation. http://www.alexa.com

[Alt99] AltaVista search engine. http://altavista.digital.com

[Aro96] Sanjeev Arora. Polynomial time approximation schemes for Euclidean TSP and other geometric problems. In *Proceedings of 37th Annual Symposium on Foundations of Computer Science (FOCS'96)*, pages 2 – 11, Burlington, Vermont, October 1996.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of International Conference on Very Large Databases (VLDB '94)*, pages 487 – 499, September 1994.

[BB99] Krishna Bharat and Andrei Z. Broder. Mirror, Mirror, on the Web: A study of host pairs with replicated content. In *Proceedings of 8th International Conference on World Wide Web (WWW'99)*, May 1999.

[BCC94] Eric W. Brown, James P. Callan, and William B. Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference of Very Large Databases (VLDB'94)*, pages 192 – 202, Santiago, Chile, September 1994.

[BD83] D. Bitton and D. J. DeWitt. Duplicate record elimination in large data files. *ACM Transactions in Database Systems (TODS)*, 8(2):255 – 265, 1983.

[BDGM95] Sergey Brin, James Davis, and Hector Garcia-Molina. Copy detection mechanisms for digital documents. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 398 – 409, San Francisco, CA, May 1995.

[BGM97] Andrei Broder, Steve C. Glassman, and Mark S. Manasse. Syntactic clustering of the web. In *Sixth International World Wide Web Conference*, pages 391 – 404, April 1997.

[BMS97] Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*, pages 265 – 276, May 1997.

[BMUT97] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for masket basket data. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*, pages 255 – 264, May 1997.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *7th International Conference on World Wide Web (WWW'98)*, April 1998.

[BP99] Sergey Brin and Lawrence Page. Google search. http://www.google.com

[Bro97] Andrei Broder. On the resemblance and containment of documents. In *Compression and complexity of Sequences (SEQUENCES'97)*, pages 21 – 29, 1997.

[BS98] Dan Boneh and James Shaw. Collusion secure fingerprinting for digital data. *IEEE Transactions on Information Theory*, 44(5):1897 – 1905, 1998.

[Bul98] William M. Bulkeley. Cyber cops try to get Internet music pirates to change their tune. *Wall Street Journal*, June 15 1998.

[CGK89] Danette Chimenti, Ruben Gamboa, and Ravi Krishnamurthy. Towards an open architecture for LDL. In *Proceedings of the 15th International Conference on Very Large Databases (VLDB'89)*, pages 195 – 203, August 1989.

[CGM97] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–37, Tuscon, Arizona, May 1997.

[CGMP96] Chen-Chuan K. Chang, Hector Garcia-Molina, and Andreas Paepcke. Boolean query mapping across heterogeneous information sources. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 8(4), August 1996.

[CL99] Ingemar Cox and Jean-Paul Linnartz. Some general methods for tampering with watermarks. *To appear IEEE Journal on Selected Areas of Communication (JSAC'99)*, 1999.

[CLR91] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. The MIT Press, 1991.

[Cou99] Countingdown.com. Popular film clips. http://www.countingdown.com.

[CP90] Douglas Cutting and Jan Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of the ACM SIGIR International Conference on Information Retrieval (SIGIR'90)*, pages 405 – 411, Minneapolis, Minnesota, January 1990.

[CS93a] Rakesh Chandra and Aries Segev. Managing temporal financial data in an extensible database. In *Proceedings of the 19th International Conference of Very Large Databases (VLDB'93)*, pages 302 – 313, Dublin, Ireland, August 1993.

[CS93b] Surajit Chaudhuri and Kyuseok Shim. Query optimization in the presence of foreign functions. In *Proceedings of 19th International Conference on Very Large Databases (VLDB'93)*, pages 529 – 542, Dublin, Ireland, August 1993.

[CS96] Surajit Chaudhuri and Kyuseok Shim. Optimization of predicates with user-defined predicates. In *Proceedings of 23rd International Conference on Very Large Databases (VLDB'96)*, pages 87 – 98, Mumbai, India, August 1996.

[CS97] Surajit Chaudhuri and Kyuseok Shim. Optimization of predicates with user-defined predicates. *Microsoft Research Tech. Report: MSR-TR-97-03*, March 1997.

[CSGM99] Junghoo Cho, Narayanan Shivakumar, and Hector Garcia-Molina. Finding replicated web documents and collections. Technical report, Stanford Infolab, February 1999.

[CTL90] R. Muntz Calvin T.Y Leung. Query processing for temporal databases. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 200 – 208, February 1990.

[Dej99] DejaNews News Research Service. http://www.dejanews.com.

[Den95] Peter J. Denning. Editorial: Plagiarism in the web. *Communications of the ACM*, 38(12), December 1995.

[Dic99] Dice Corp. Watermarking audio. http://www.digital-watermark.com.

[Dig99] Digimarc Corp. Watermarking images. http://www.digimarc.com.

[DM97] Yining Deng and B. S. Manjunath. Content-based search of video using color, texture and motion. In *IEEE International Conference on Image Processing*, pages 534–537, October 1997.

[Dmy84] Edward Dmytryk. *On Film Editing.* Focal Press, Boston, MA, USA, 1984.

[Haas90] Laura Haas et al. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, pages 143–160, March 1990.

[EKW91] Ramez Elmasri, Yeong-Jim Kim, and Gene T.J. Wuu. Efficient implementation techniques for the time index. In *Proceedings of 7th IEEE International Conference of Data Engineering (ICDE'91)*, pages 102 – 111, 1991.

[FJ92] Christos Faloutsos and H.V. Jagadish. On b-tree indices for skewed distributions. In *Proceedings of the 18th International Conference of Very Large Databases (VLDB'92)*, pages 363 – 374, Vancouver, British Columbia, Canada, September 1992.

[FM85] Philippe Flajolet and Nigel Martin. Probabilistic counting algorithms for database applications. *Journal of Computer System Sciences*, 31(2):182 – 209, 1985.

[For98] Forrester. Entertainment and technology strategies. *The Forrester Report*, 10(1), January 1998.

[FSGM+98] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries effciently. In *Proceedings of International Conference on Very Large Databases (VLDB '98)*, pages 299 – 310, August 1998.

[FSN95] Myron Flickner, Harpreet S. Sawhney, and Wayne Niblack. Query by image and video content: The QBIC system. *IEEE Computer Magazine*, 28(9):23–32, September 1995.

[Gev95] Theo Gevers. Color image invariant segmentation and retrieval. In *PhD thesis, Wiskunde Informatica Natuurkunde and Sterrenkunde, Amsterdam, The Netherlands*, 1995.

[GGM95] Luis Gravano and Héctor García-Molina. Generalizing *GlOSS* for vector-space databases and broker hierarchies. In *Proceedings of the Twenty-first International Conference on Very Large Databases (VLDB'95)*, pages 78–89, September 1995.

[GGMT94] Luis Gravano, Hector Garcia-Molina, and Anthony Tomasic. The effectiveness of *GlOSS* for the text database discovery problem. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*, pages 126 – 137, May 1994.

[GIM99] Aris Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of Conferernce on Very Large Databases VLDB'99)*, September 1999.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, New York, 1979.

[GKA98] Ullas Gargi, Rangachar Kasturi, and S. Antani. Performance characterization and comparison of video indexing algorithms. In *International Conference on Computer Vision and Pattern Recognition (CVPR'98)*, June 1998.

[GMGS96] Hector Garcia-Molina, Luis Gravano, and Narayanan Shivakumar. dSCAM: Finding document copies across multiple databases. In *Proceedings of International Conference on Parallel and Distributed Information Systems (PDIS'96)*, Miami Beach, Florida, December 1996.

[Gol96] Paul Goldstein. *Copyright's Highway*. Hill and Wang, 1996.

[Goo97] Dan Goodin. A market waiting to happen. *The Recorder*, July 1997.

[GS93] Himawan Gunadhi and Arie Segev. Efficient indexing methods for temporal relations. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 5(3):496–509, 1993.

[Har99] Hartung. Links to watermarking sites. http://www-nt.e-technik.uni-erlangen.de/~hartung/watermarkinglinks.html.

[Hei96] Nevin Heintze. Scalable document fingerprinting. In *Proceedings of Second USENIX Workshop on Electronic Commerce*, pages 191 – 200, November 1996.

[HHW97] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*, Tuscon, Arizona, June 1997.

[HNSS96] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Arun N. Swami. Selectivity and cost estimation for joins based on random sampling. *Journal of Computer and System Sciences*, 52(3):550 – 569, June 1996.

[HS93] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: optimizing queries with expensive predicates. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*, pages 267 – 276, Washington, D.C., May 1993.

[IIS99] Piotr Indyk, Giridharan Iyengar, and Narayanan Shivakumar. Finding pirated video sequences on the internet. Technical report, Stanford Infolab, February 1999.

[IL98] Giridharan Iyengar and Andy B. Lippman. Models for automatic classification of video sequences. In *Storage and Retrieval from Image and Video Databases*, pages 3312 – 3331, January 1998.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Symposium on Theory of Computing (STOC'98)*, pages 604–613, 1998.

[Inf99] Infoseek search engine. http://www.infoseek.com.

[INSS92] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos .K. Sellis. Parametric query optimisation. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 103 – 114, Vancouver, August 1992.

[Jan95] Jan Jannink. Implementing deletion in B+trees. *SIGMOD Record*, 24:33 – 38, March 1995.

[JV96] Anil K. Jain and Aditya Vailaya. Image retrieval using color and shape. *Pattern Recognition Journal*, 29:1233–1244, August 1996.

[KLK97] Ulrich Kohl, Jeffrey Lotspiech, and Marc A. Kaplan. Safeguarding digital library contents and users. *D-Lib Magazine*, September 1997.

[KM91] Brewster Kahle and Art Medlar. An information system for corporate users: Wide Area Information Servers. Technical Report TMC199, Thinking Machines Corporation, April 1991.

[KMRV97] Jon Kleinberg, Rajeev Motwani, Prabhakar Raghavan, and S. Venkatasubramanian. Storage management for evolving databases. In *Proceedings of 38th IEEE Symposium on Foundations of Computer Science (FOCS'97)*, pages 353 – 362, June 1997.

[KR95] Burt Kaliski and Matt Robshaw. Message authentication with MD5. http://www.rsa.com/rsalabs/pubs/cryptobytes/spring95/md5.htm, 1995.

[KS91] Curtis Kolovson and Michael Stonebraker. Segment Indexes: Dynamic indexing techniques for multi-dimensional interval data. In *Proceedings of 1991 ACM SIGMOD International Conference on Management of Data (SIGMOD'91)*, pages 138–147, May 1991.

[LYGM99] Wilburt Labio, Ramana Yerneni, and Hector Garcia-Molina. Capability sensitive query processing on Internet sources. In *Proceedings of the International Conference on Data Engineering (ICDE'99)*, pages 50 – 59, March 1999.

[Man94] Udi Manber. Finding similar files in a large file system. In *Proceedings of the winter USENIX Conference*, pages 1 – 10, 1994.

[ME97] Alvaro Monge and Charles Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Proceedings of SIGMOD 1997 Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'97)*, May 1997.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[RIAA99] Recording Industry Association of America (RIAA). Personal communication.

[Olk93] Frank Olken. *Random sampling from databases.* Ph.D. dissertation, UC Berkeley, April 1993.

[OS95] Virginia E. Ogle and Michael Stonebraker. Chabot: Retrieval from a relational database of images. *IEEE Computer Magazine*, 28(9):40–48, September 1995.

[Osa99] Yoshitomo Osawa. Sony corporation. http://www.sony.co.jp/soj/CorporateInfo/CHANCENavigator/f06.html, 1999.

[PCY95] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash based algorithm for mining association rules. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*, pages 175 – 186, May 1995.

[PK79] Gerald J. Popek and Charles S. Kline. Encryption and secure computer networks. *ACM Computing Surveys*, 11(4):331–356, December 1979.

[PP97] James Pitkow and Peter Pirolli. Life, death, and lawfulness on the electronic frontier. In *International conference on Computer and Human Interaction (CHI'97)*, 1997.

[RM82] Karel Reisz and Gavin Millar. *The Technique of Film Editing (Second Edition).* Hastings House, New York, NY, USA, 1982.

[Ros96] Phil E. Ross. Cops versus robbers in cyberspace. *Forbes Magazine*, pages 134 – 139, September 9 1996.

[SAC+79] 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. In P. A. Bernstein, editor, *SIGMOD*, pages 23–34, Boston, MA, 1979. acm.

[Sal88] Gerald Salton. Term-weighting approaches in automatic text retrieval. *Information processing & management.*, 24(5):513, 1988.

[Sal89] Gerard Salton. *Automatic Text Processing: The transformation, analysis, and retrieval of information by computer.* Addison-Wesley, 1989.

[Sal92] Gerald Salton. The state of retrieval system evaluation. *Information processing & management.*, 28(4):441, 1992.

[SB] Gerald Salton and Chris Buckley. The SMART information retrieval system.

[SB88] Gerald Salton and Chris Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513 – 523, 1988.

[SB91] Michael J. Swain and Dana H. Ballard. Color indexing. *International Journal of Computer Vision*, 7(1):11–32, 1991.

[SBW98] Olin Sibert, David Bernstein, and David Van Wie. Securing the content, not the wire, for information commerce. http://www.intertrust.com/architecture/stc.html, 1998.

[SGM95] Narayanan Shivakumar and Hector Garcia-Molina. SCAM:a copy detection mechanism for digital documents. In *Proceedings of 2nd International Conference in Theory and Practice of Digital Libraries (DL'95)*, Austin, Texas, June 1995.

[SGM96] Narayanan Shivakumar and Hector Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *Proceedings of 1st ACM Conference on Digital Libraries (DL'96)*, Bethesda, Maryland, March 1996.

[SGM97] Narayanan Shivakumar and Hector Garcia-Molina. Wave-Indices: Indexing evolving databases. In *Proceedings of ACM SIGMOD International Conference*

on Management of Data (SIGMOD'97), pages 381 – 392, Tuscon, Arizona, May 1997.

[SGMC98] Narayanan Shivakumar, Hector Garcia-Molina, and Chandra S. Chekuri. Filtering with approximate predicates. In *Proceedings of the International Conference on Very Large Database (VLDB'98)*, pages 263 – 274, New York, New York, August 1998.

[SL97] Mark Stefik and Giuliana Lavendel. Libraries and digital property rights. In *Proceedings of European Conference on Digital Libraries*, pages 1 – 10, Pisa, Italy, December 1997.

[Smi56] William E. Smith. Various Optimizers for Single-Stage Production. *Naval Research Logistics Quarterly*, pages 59 – 66, 1956.

[Sri95] Rohini K. Srihari. Automatic indexing and content-based retrieval of captioned images. *IEEE Computer Magazine*, 28(9):49 – 56, September 1995.

[ST99] Betty Salzberg and Vassilis J. Tsotras. A comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(1), March 1999.

[TA94] Alfonse Tal and Rafael Alonso. Integration of commit protocols in heterogeneous databases. *Distributed and Parallel Databases*, 2(2):209 – 234, April 1994.

[TGMS94] Anthony Tomasic, H. Garcia-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*, pages 289 – 300, Minneapolis, Minnesota, May 1994.

[TK95] Vassilis J. Tsotras and Nickolas Kangelaris. The Snapshot-Index, an I/O optimal access method for timeslice queries. *IEEE Transactions of Knowledge and Data Engineering*, 7(4), 1995.

[TPC99] TPC-Committee. Transaction processing council (TPC). http://www.tpc.org.

[Ull88] Jeffrey D. Ullman. *Principles of database and knowledge-base systems*, Volume 1. Computer Science Press, 1988.

[VP97] Vasilis Vassalos and Yannis Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *Proceedings of the International Conference on Very Large Databases (VLDB 97)*, pages 256 – 265, August 1997.

[Wie87] Gio Wiederhold. *File organization for database design.* McGraw-Hill (New York, NY), 1987.

[WKSS96] H. Wactlar, T. Kanade, M. Smith, and S. Stevens. Intelligent access to digital video: The informedia project. *IEEE Computer*, 29(5), May 1996.

[WVZT90] Kyu-Young Whang, Brad T. Vander-Zanden, and Howard M. Taylor. A linear-time probabilistic counting algorithm for db applications. *ACM Transactions on Database Systems*, 15(2):208 – 229, 1990.

[YGM95] Tak Yan and Hector Garcia-Molina. SIFT – A tool for wide-area information dissemination. In *Proceedings of USENIX Winter Conference*, pages 177 – 186, 1995.

[YL95] Boon-Lock Yeo and Bede Liu. Rapid scene analysis on compressed video. *IEEE Transactions on Circuit and Systems for Video Technology*, 5(6):533 – 544, December 1995.

[Zip49] George K. Zipf. *Human behavior and the principle of least effort.* Addison-Wesley Press, Cambridge, Massachusetts, 1949.