LEXICON CACHING IN FULL-TEXT DATABASES

by

Jingyu Liu

B.E., Shenyang Institute of Aeronautical Engineering, 1993

M.E., Beijing University of Aeronautics and Astronautics, 1996

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Jingyu Liu 2005
SIMON FRASER UNIVERSITY
Spring 2005

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Jingyu Liu

Degree: Master of Science

Title of thesis: Lexicon Caching in Full-Text Databases

Examining Committee: Dr. Andrei Bulatov
Chair

Dr. Tiko Kameda
Senior Supervisor

Dr. Ke Wang
Supervisor

Dr. Jian Pei
SFU Examiner

Date Approved: March 9, 2005

SIMON FRASER UNIVERSITY

PARTIAL COPYRIGHT LICENCE

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.\

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

W. A. C. Bennett Library
Simon Fraser University
Burnaby, BC, Canada

Abstract

Caching is a widely used technique to leverage access time difference between two adjacent levels of storage in the computer memory hierarchy, e.g., *cells in main memory* $\leftrightarrow$ *cells in the cpu cache*, and *blocks on disk* $\leftrightarrow$ *pages in main memory.* Especially in a database system, buffer management is an important layer to keep hot spot data in main memory so as to minimize slow disk I/O and thus improve system performance. In this thesis, we present a term-based method to cache lexicon terms in full-text databases, which aims at reducing the size of the lexicon that must be kept in memory, while providing good performance for finding the requested terms. We empirically show that, under the assumption of Zipfs-like term access distribution, given the same amount of main memory, our term-based caching method achieves a much higher hit ratio and much faster response time than traditional page-based buffering methods used in database systems.

Acknowledgments

I owe a deep debt of gratitude to my senior supervisor, Dr. Tiko Kameda, for his invaluable guidance, insightful advice and continuous encouragement during my research. This thesis would not have been possible without his strongest support and patience with me.

My appreciation also goes to my supervisor Dr. Ke Wang and examiner Dr. Jian Pei. Their critical review and comments make this thesis more solid and comprehensive.

Finally, I would like to thank my family, without whose support and encouragement, this thesis would not have been finished in time. My thanks go to my parents, Haichen Liu and Defen Huang, my sister's family, Yang Liu, Yingmu Zhong and two lovely nieces, Joy and Kate, and my wife, Shuaiying Huang, whose constant comforting and encouragement made the writing of this thesis more than worth-while.

Contents

Approval ii

Abstract iii

Acknowledgments iv

Contents v

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Full-Text Database 1

1.1.1 Operational Model 3

1.1.2 Implementation Challenges 4

1.2 Need of Lexicon Caching in Full-Text Database 6

1.3 Outline of Thesis 8

2 Related Work 9

2.1 Database Storage 9

2.2 Access Path 10

2.2.1 Addressing Problem 10

2.2.2 B-Tree 10

2.2.3 Hash Index 12

2.2.4 Lexicon and Inverted Index 13

2.3 Traditional Buffer Management 15

2.3.1 Overview 15
2.3.2 Reference Locality 16
2.3.3 Replacement Strategies 18
2.4 Buffer Management in Full-Text Database 21

3 Lexicon Caching **24**
3.1 Caching Granularity 24
3.1.1 Page Caching 24
3.1.2 Hotset in Lexicon 25
3.1.3 Lexicon Caching Architecture 25
3.2 Memory Management 27
3.2.1 Memory Fragmentation 27
3.2.2 Memory Management Overhead 28
3.2.3 Special Requirements for Lexicon Cache Management 29
3.3 Lexicon Cache Design 29
3.3.1 Dynamic Hashing Chunk Cache 30
3.3.2 Internal Structure of Chunk 32

4 Empirical Studies **38**
4.1 Experimental System 38
4.2 Theoretical Analysis 39
4.3 Experiments 41
4.3.1 Data Statistics 42
4.3.2 Construction of Lexicon 43
4.3.3 Lexicon Search 49

5 Conclusions and Future Work **57**
5.1 Contributions 57
5.2 Future Work 58

List of Tables

1.1 Example Text 4
1.2 Example Inverted Index 5
1.3 Example Lexicon 6

4.1 Parameters of Experimental System 39
4.2 Document Repository Statistics. 42
4.3 Distinct Terms Statistics. 43
4.4 Lexicon B-Tree Statistics. 44
4.5 Cache Hit Ratio (Lexicon Construction). 44
4.6 Theoretical Cache Hit Ratio (Lexicon Construction). 45
4.7 Buffer Hit Ratio (Lexicon Construction). 45
4.8 Cache Response Time (Lexicon Construction). 48
4.9 Buffer Response Time (Lexicon Construction). 48
4.10 Cache Terms Statistics (Lexicon Construction). 53
4.11 Cache Hit Ratio (Lexicon Search). 54
4.12 Theoretical Cache Hit Ratio (Lexicon Search). 54
4.13 Buffer Hit Ratio (Lexicon Search). 55
4.14 Cache Response Time (Lexicon Search). 55
4.15 Buffer Response Time (Lexicon Search). 55
4.16 Cache Terms Statistics (Lexicon Search). 56

List of Figures

1.1 Full-Text Database Architecture 3

2.1 Database Buffer Manager 16
2.2 Fault curve for a join computed by nested scans using sequential scans. . . . 18

3.1 Lexicon Cache Architecture. 26
3.2 DHCC Memory Layout 30
3.3 Internal Structure of a Chunk. 33
3.4 BST Node Structure. 34

4.1 Experimental System Components 38
4.2 Experimental Hit Ratio vs. Theoretical Hit Ratio (Lexicon Construction) . . 46
4.3 Cache Hit Ratio vs. Buffer Hit Ratio (Lexicon Construction) 47
4.4 Experimental Hit Ratio vs. Theoretical Hit Ratio (Lexicon Search) 50
4.5 Cache Hit Ratio vs. Buffer Hit Ratio (Lexicon Search) 51

Chapter 1

Introduction

With the advent of the Internet, the amount of information available to the public has become tremendously large, and grows at an exponential rate. Such information exists in many forms: text, images, movies, sound, etc. The locations where information is stored can be found by their URLs (Uniform Resource Locator) [2]. A question naturally arises, as to how to find a particular piece of information on the *World Wide Web(WWW)* without knowing its URL? In a library, a reader may find the book he is interested in by looking up index cards if he knows the title or author(s) of the book. A similar mechanism is available in *WWW* in the form of search engines, which serve as 'index cards' for users to find a particular URL that may contain the information they are interested in. Actually, search engines are becoming an indispensable part of the *WWW* community. A person surfing on the web queries a search engine to find what he/she is interested in but has no idea where it is. Without powerful assistance by public search engines, it's hard to imagine how difficult it would be to browse in the exponentially-growing *WWW*. A full-text database is the backbone of a search engine.

1.1 Full-Text Database

A full-text database, also called a document database, is a text-oriented database system, which is not like a traditional database that is record-oriented. These two kinds of databases mainly differ in the following aspects:

1. Data Model

- The basic unit stored in a traditional database, whether of Network model, Hierarchical Model or Relational Model, is a structured record that contains one or more fields that have fixed/variable lengths of pre-defined data types.
- The basic unit stored in a full-text database is a block of text strings, which can be an article, an HTML file on the internet, a sentence in a paragraph, a chapter in a book, or even the whole book, depending on the granularity required by applications.

2. Storage Structure

- In a traditional database, storage contains two parts: records and one or more indices that are built on record fields to speed up record update/delete/search.
- In a full-text database, storage contains three parts: original documents, a lexicon that contains all distinct terms (keywords)[1] extracted from the original documents, and an inverted index that is a mapping from terms in the lexicon to documents.

3. Database Size

- The volume of data stored in a traditional database is normally from hundreds of kilo-bytes to hundreds of mega-bytes, with few exceptions of large applications that may contain giga-bytes of data. Compared to the size of the original data, the indices in a traditional database occupy just a small portion in the whole database.
- The volume of data stored in a full-text database is usually measured in gigabytes, even terabytes. The inverted index in a full-text database occupies a significant portion in the whole database, probably as much as the original documents[2].

---

[1] In this thesis, **term** and **keyword** are equivalent and interchangeable, unless distinguished explicitly.

[2] For each word occurrence in the text, the inverted index needs to store at least its position in the text, which normally takes 4 bytes, and usually more information like whether the word is in the title etc. is also stored. So each word occurrence will need more than 5 bytes in the inverted index, while most usually-used words are below 8 bytes. Please see Google on page 7 for an example.

1.1.1 Operational Model

As stated above, a full-text database is composed of three parts: documents, an inverted index and a lexicon. If depicted as a layered system, their positions, from bottom to top, are shown in Figure 1.1.

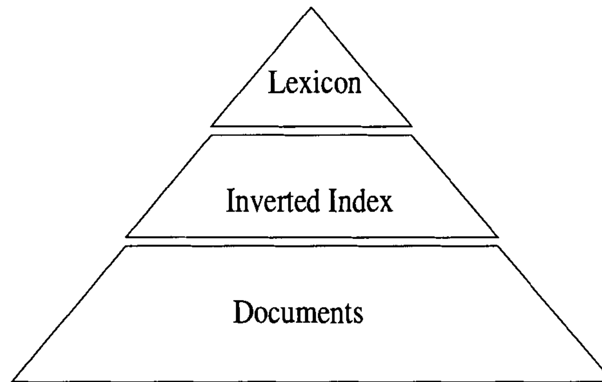

Figure 1.1: Full-Text Database Architecture

Documents are a collection of individual units that are targets of queries. After processing a query, the matched documents are returned to the user for further investigation.

Inverted index contains, for each term in the lexicon, a list of pointers to all occurrences of that term in the main text. Each pointer is a document ID in which that term appears. After a query is processed on the inverted index, a list of related documents is returned.

Lexicon stores both the terms that can be used to search the collection and the auxiliary information needed to allow queries to be processed. The minium information that must be stored in the lexicon is the term t, the address I_t in the inverted index, and the term's occurrence frequency f_t in the collection. To answer a query, the lexicon is first consulted to get each query term's address in the inverted index.

The following example illustrates the concept introduced above. Table 1.1 shows the first and second verses from *Genesis*, in which each sentence is deemed as a document.

Suppose that the granularity of the inverted index is a document. The inverted list for each term would be in the form of $< t, f_t; d_1, d_2, \ldots, d_{f_t} >$, where t is the term, f_t is the

Document ID	Text
1	In the beginning God created the heaven and the earth.
2	And the earth was without form, and void;
3	and the darkness was upon the face of the deep.
4	And the spirit of God moved upon the face of the waters.

Table 1.1: Example Text

term frequency in the text, d_1, d_2, ..., d_{f_t} are the document IDs in which the term occurs. Table 1.2 (on page 5) shows the inverted index corresponding to the text in Table 1.1.

The lexicon contains the disk address of each term's inverted list. The data structure of each entry in the lexicon would be in the form of $< t, f_t, D_{I_t} >$, where t is the term, f_t is the term frequency in the text, and D_{I_t} is the disk address of the term's inverted list. Entries in the lexicon are normally sorted in some way (e.g. alphabetical order, hash etc.) to quickly find a term. Table 1.3 (on page 6) shows the lexicon for the example text in Table 1.1.

You may notice that the first two fields (term t and frequency f_t) of Table 1.2 and Table 1.3 are the same. This way, the lexicon may be reconstructed by just scanning the inverted index so as to provide some degree of fault-tolerance, and can also provide consistence check.

1.1.2 Implementation Challenges

For the purpose of persistence and consistency, these three parts (Documents, index and lexicon) must be stored on secondary storage (e.g. hard disk), and only a portion of them can be kept in main memory at any moment. A full-text database normally deals with millions of documents, containing gigabytes or terabytes of data. It is the *size* that brings up two challenges when managing such huge volumes of data[41].

1. **Storing the data efficiently**

   No matter how much storage space is available, someone always finds something to fill it with. It seems that Parkinson's Law[29] applies here[3]. It has been observed since the mid-1980s that the memory usage of evolving systems tends to double roughly once every 18 months. Fortunately, the memory space available for constant dollars also tends to double about every 18 months. Unfortunately, however, the laws of physics guarantee that the latter cannot continue indefinitely. In full-text database

Term	Inverted List
and	$< 4; 1, 2, 3, 4 >$
beginning	$< 1; 1 >$
created	$< 1; 1 >$
darkness	$< 1; 3 >$
deep	$< 1; 3 >$
earth	$< 2; 1, 2 >$
face	$< 1; 4 >$
form	$< 1; 2 >$
God	$< 2; 1, 4 >$
heaven	$< 1; 1 >$
in	$< 1; 1 >$
moved	$< 1; 4 >$
of	$< 2; 3, 4 >$
spirit	$< 1; 4 >$
surface	$< 1; 3 >$
the	$< 4; 1, 2, 3, 4 >$
upon	$< 2; 3, 4 >$
void	$< 1; 2 >$
was	$< 2; 2, 3 >$
waters	$< 1; 4 >$
without	$< 1; 2 >$

Table 1.2: Example Inverted Index

implementation, compression is normally used to store more data in less space.

2. **Providing fast access through keyword search**
   When constructing an inverted index over a huge volume of documents, it may take several months to complete the construction without good analysis and design, and the resulting inverted index would probably occupy a similar amount of storage to the documents themselves. As in any database system, disk access time is the primary factor that affects performance. If the resulting inverted index is very large, it would take much longer to answer a query. In full-text database implementation, memory buffer and compression are used to eliminate or decrease necessary disk access time, and thus improve performance.

[3]**Parkinson's Law**: Data expands to fill the space available for storage.

Term	Term Frequency	Inverted List Disk Address
and	4	DI_{and}
beginning	1	$DI_{beginning}$
created	1	$DI_{created}$
darkness	1	$DI_{darkness}$
deep	1	DI_{deep}
earth	2	DI_{earth}
face	1	DI_{face}
form	1	DI_{form}
God	2	DI_{God}
heaven	1	DI_{heaven}
in	1	DI_{in}
moved	1	DI_{moved}
of	2	DI_{of}
spirit	1	DI_{spirit}
surface	1	$DI_{surface}$
the	4	DI_{the}
upon	2	DI_{upon}
void	1	DI_{void}
was	2	DI_{was}
waters	1	DI_{waters}
without	1	$DI_{without}$

Table 1.3: Example Lexicon

1.2 Need of Lexicon Caching in Full-Text Database

The current interest in full-text database research is in methods to do the following:

- Compress the text to save storage space.
- Reduce the time needed to construct the inverted index.
- Return the most related documents by improving the inverted index structure and query processing algorithm.

However, there are some methods that don't receive much attention but are actually very useful in practice. Caching lexicon terms is one of them.

In a full-text database, the lexicon is the most frequently accessed component, because no matter what operation is performed on the database — be it an update to the inverted

index or a query on combination of a few terms — they must first go through the lexicon to find the appropriate inverted list. Compared to the inverted index and document repository, a lexicon is tiny — probably occupying just 0.05% (or less) of the database's total storage space. However, when the documents repository becomes very large, the corresponding lexicon size will also become large in absolute terms as well and cannot be ignored in most cases. For instance, in the prototype of Google[3], 24 million web pages were fetched, and the whole database occupied 108.7 GB[4] of storage space, in which there were 14 million distinct terms taking up 293 megabytes! Their solution was to assign a computer as the lexicon server, whose main memory was totally used to cache the lexicon. This way, most access to the lexicon could be processed in main memory to reduce disk access. Another approach was a distributed full-text index[25], where an inverted index was distributed over a set of computers, each of which maintained a subset of the lexicon and the corresponding inverted index. As the lexicon was split into small subsets, it was possible to hold a part of the whole lexicon in each computer's main memory without too much memory requirement on average.

Although it may sometimes be possible to accommodate the whole lexicon in the main memory, there are always some situations in which the whole lexicon cannot be held in main memory. For instanc, the author had a chance to work on a distributed full-text search engine, where the lexicon and inverted index reside on *Index Server*, and the documents on *Data Server*. All the lexicon, inverted index and documents are fully duplicated on every node. Each node is a Pentium PIII 550 with 512MB of main memory, and runs under Linux with kernel 2.2.4. Since the Index Server needs to maintain the lexicon and inverted index in the same physical memory, and the lexicon is relatively large (occupies about 128 MB), it's not feasible to hold the whole lexicon in main memory — the inverted index also consumes a large amount of memory. A page-based buffer manager was implemented for this purpose. It was observed though that the lexicon's buffer performance was not as good as observed in traditional page-based database buffer management. It's this observation that motivated this thesis. Is there any other approach that would improve the lexicon buffering performance? The research on this topic resulted in an idea that is different from the traditional database buffering method. In the case of lexicon buffer management, term-based caching outperforms page-based buffering. To the best of our knowledge, no such

[4]It contains 53.5 GB of document repository (this is after compression; the original document size was 147.8 GB), a 55.0 GB inverted index, and a 293 MB lexicon.

research has been done in the literature. The remaining chapters of this thesis discuss and evaluate the idea thoroughly.

1.3 Outline of Thesis

The rest of this thesis is organized as follows:

Chapter 2 investigates related research works on access path and database buffer management.

Chapter 3 describes the proposed lexicon caching scheme in detail.

Chapter 4 shows the experimental results and performance analysis.

Chapter 5 concludes the thesis.

Chapter 2

Related Work

Although few research works related with the full-text database keyword caching can be found in the literature, some directly related concepts can be found in the methodologies of access path and buffer management in a database system. In this chapter, we look at access path and database buffer management in general.

2.1 Database Storage

Secondary storage (normally hard disks) is used to store data in a database system due to various reasons such as the following:

- Most databases typically require a large amount of storage space, usually in the hundreds of megabytes, gigabytes, or even terabytes. Therefore, it's usually not feasible to hold the whole database in main memory.
- The *durability* requirement of transaction *ACID* properties [34] suggests that durable secondary storage is more suitable than volatile main memory to store persistent data.
- It provides low storage cost per bit.

In modern operating systems, secondary storage is commonly managed by a *file system*, which provides a primitive interface to manipulate data on the secondary storage, consisting of read, write, delete operations, etc. The basic unit transfered between the secondary storage and the file system is called a *page*, which is a block of continuous area on disk.

Secondary storage is some orders of magnitude slower than main memory. Therefore reading/writing data on disk takes longer than reading/writing data in main memory. If too many disk I/O operations are issued, the performance of the database system would degrade very fast. Disk I/O can be reduced by two approaches, *access path* and *buffer management*, where the former tries to reduce disk I/O by providing the shortest path to the disk page containing the requested record, while the latter tries to keep as many pages as possible in main memory to avoid disk I/O.

2.2 Access Path

2.2.1 Addressing Problem

Records in a database are stored sequentially in disk pages of a file. Therefore, the address of a record on disk can be represented as a tuple <*file_number, page_number, page_offset*>, where *file_number* is the file descriptor in which the record resides, *page_number* is the page number inside the file, and *page_offset* is the in-page offset from where the record starts to be stored. A user usually wants to find a record just with the knowledge of some of its field values. Thus a mapping mechanism must exist between the record's field value and the record's disk address. Access path serves as the mapping mechanism and is implemented as an *index* built onto one or more fields of the records to provide as short a path as possible to the page containing the data.

The field used to build an index is called a *key*. The *primary key* consists of the field(s) whose value is unique for all records. On the other hand, a *secondary key* refers to a field whose value may not be unique.

Various index structures are available to implement the access path. We will look at a few of them that are relevant to this thesis.

2.2.2 B-Tree

The B-Tree index structure was first introduced by Bayer and Mcreight[1]. Most research works on B-Trees are based on it. Properties of a B-Tree are discussed in Knuth[16] and Comer[4]. Concurrency control on a B-Tree and similar data structures is investigated by Lehman and Yao[19], and Kung and Lehman[17]. A B-Tree is the most important index structure in all kinds of database systems.

Structure

A B-Tree is a balanced multi-level tree structure. Each node in a B-Tree is implemented by a disk page. The nodes in a B-Tree can be classified into two types, *leaf* nodes and *non-leaf (internal)* nodes as follows:

- Every leaf has the same depth, i.e., is at the same distance from the root.
- N_K, the number of keys contained in any node, satisfies

$$d \leq N_K \leq 2d \tag{2.1}$$

for some d, that is, every node (except the root) must be at least half-full.

- All the keys in a node are sorted in the non-decreasing order by some criteria.
- Each key K in an internal node has two pointers, P_l on the left and P_r on the right, where all keys C_{P_l} in the child node pointed to by P_l satisfy $C_{P_l} < K$, and all keys C_{P_r} in the child node pointed to by P_r satisfy $C_{P_r} \geq K$. So N_i, the number of pointers in an internal node, satisfies

$$d+1 \leq N_i \leq 2d+1 \tag{2.2}$$

- Each key K in a leaf node has just one pointer pointing to the record whose field value equals K. So N_l, the number of pointers in a leaf node satisfies

$$d \leq N_l \leq 2d \tag{2.3}$$

Properties and Implications

A B-Tree has the following properties:

- For $n \geq 1$, the upper bound on height h of a B-Tree containing n keys with parameter $d \geq 2$ can be determined by equation $h = \lceil \log_d n \rceil$. Note that accessing each node of a B-Tree requires one disk I/O operation.
- This suggests that the capacity of nodes in a B-Tree should be increased to contain more keys so as to increase nodes' fan-out, thereby keeping the tree short. For instance, for a B-Tree containing 1,000,000 keys, and $d = 100$, we have $h = 3$, which means only four[1] disk pages need to be accessed to get the requested record.

[1] records are normally not stored in B-Tree disk pages, but in other disk pages.

- If the nodes have a large fan-out, the number of nodes on level $n+1$ is much more than nodes on level n, so the number of higher-level nodes are relatively small, and it's feasible to keep them in main memory to further reduce disk I/O. For instance, in the example above, the first two levels only contain 202 nodes[2], but if these 202 nodes are kept in main memory, the disk I/O will be reduced by 50%, which is a big improvement.

2.2.3 Hash Index

Hashing is commonly used in computer systems, from operating systems to specific applications. Compared to a B-Tree, which provides a multi-level access path, a hash index provides a single-level access path to required records. Hashing algorithms are discussed and analyzed in detail Knuth[16] and Cormen et al.[5]. Two kinds of hashing are known in the literature: *static hashing* and *dynamic hashing.* Dynamic hashing includes *extendible hashing* [10] and *linear hashing* [21][18]. We will focus on static hashing, since it's the most practical hashing scheme implemented in commercial database systems[12].

Structure

A hash index contains two components, *hash functions* and *index pages.*

- *Hash function* is a map from keys to index pages. Suppose that the set of all keys is K and the set of all index pages is P. Then a hash function h is a mapping from K to P, i.e., $h(K) \subset P$. Let the number of elements in set K and the number of elements in set P be $|K|$ and $|P|$ respectively. Normally $|K| \gg |P|$, i.e., $|K|$ is much larger than $|P|$. Therefore, hash function h maps multiple keys into one index page, which is called *conflict* and is not avoidable. An important task of a hash function is to find a good hash algorithm to distribute keys into index pages uniformly.

- Keys are consecutively stored in each index page, either sorted or not. Each key has a pointer associated with it, pointing to the disk address of the record indexed by the key. Since hash conflict is not avoidable, some index page may be assigned too many keys. If the page's storage capacity is exceeded, then another index page needs

[2] as stated in equation 2.2, an internal node may have at most $2d+1$ child nodes, so for $d = 100$, there are at most 202 nodes in the first two levels.

to be *chained* onto this page to accommodate more keys. This is called an *overflow*. If a hash function does not produce uniform mapping, some pages may have several overflow pages, which degrades system performance since more disk I/O would be involved during the search for a key.

Properties and Implications

Hashing index has the following properties:

- Since the access path in a hashing index is single-level and disk I/O dominates the database operation time, the time to search a key can be deemed as constant $O(1)$[3].
- A good hashing function is uniform, which makes the keys randomly distributed over the index pages. A hashing index can only provide random access to individual records. There is no efficient way to access a range of keys, since such an operation needs to access each key individually. In contrast, a B-Tree doesn't have this disadvantage.

2.2.4 Lexicon and Inverted Index

In a full-text database, access paths are implemented as a lexicon and inverted index. We described the concepts of the lexicon and inverted index in Chapter 1. There are many implementations of the lexicon and inverted index available in the literature. Although these implementations differ in some aspects, they all have the architecture shown in Figure 1.1. In this section we will review some full-text database implementations, with emphasis on lexicon implementation.

McDonell[24] proposed an implementation in which each keyword is incorporated into the keyword's inverted list by hashing, so there is no separate lexicon. This way, access time is reduced thanks to fewer layers. The sample database in this paper is quite small though, just about 12,000 records and 1,000 keywords. This approach is apparently not suitable for giga-byte full-text databases.

Zobel, Moffat, and Davis[44] proposed an inverted indexing scheme based on compression, which ensures that storage requirement is small and dynamic update is straightforward. The only assumption they made is that the whole lexicon can be held in main memory, and

[3] The access to the overflow list is not considered here, because it's rare when hash function is good enough.

at most one disk access is needed to answer a query. This paper briefly mentioned that if the lexicon cannot be held in main memory, then it can be partitioned and an abridged lexicon could be maintained in main memory, and keyword occurrence patterns in texts should follow Zipf's Law[4] [43]. But they neither described how such a partial in-memory lexicon could be implemented, nor did they say whether the query terms' search pattern actually follows Zipf's Law as well.

In another paper[45] by the same authors of [44], *n-gram*[5] and a sorted lexicon scheme are proposed to improve performance for partially specified term searching (e.g., the word 'lab*r' means any word starting with 'lab' and ending with 'r', such as 'labor', 'laborer', and 'labrador' etc.). They assume that the lexicon can be kept in main memory in a static full-text database, since no update is needed and the lexicon can be compressed to save space.

The Google[3] web search engine implemented a new ranking scheme that takes web pages' references into account, which achieves high quality answers to queries. In their implementation, a dedicated computer is used to store the whole lexicon in its main memory, which is about 293MB containing 14 million keywords. The lexicon contains two parts: one part is a distinct keyword list in which each keyword is terminated with a null character; the other part is a hash table of pointers that point to the beginning of each keyword in the keyword list. This implies that Google doesn't compress its lexicon and doesn't support partially specified keyword search. Also, the lexicon and inverted index are static and updated offline.

Melnik, Raghavan, Yang, and Molinag[25] implemented a distributed inverted index for a large collection of web pages. Their main contribution was to introduce pipelining into the core index building process, substantially reducing index building time. The lexicon is partitioned to be held on each *indexer* (a computer responsible for building index), and a B-Tree structure is adopted to store the lexicon and inverted index. The B-Tree index enables their implementation to support *hot update* — ability to update the lexicon and inverted index when query is being processed.

Nagarajarao, Ganesh, and Saxena[26] implemented an inverted index that supports efficient query and incremental index update. The query can be answered in two modes:

[4]Please see Equation 3.1 on page 25 for details.

[5]*n-gram* is n-character slice of some longer string. For example, the word 'text' contains the following 2-grams: te, ex, xt.

immediate response and batch mode with proper scheduling. Their lexicon uses a variant implementation used in [3]. The lexicon contains two parts: a token array into which all tokens (null terminated) are concatenated together, and a hash array that stores the offset of tokens in the token array. When inserting a token, the token is concatenated to the end of the token array, and the token's hash code is computed. Aquadratic probing scheme[6] probes the empty slot in the hash array to store the token and its offset in the token array. This lexicon structure makes hot update possible.

2.3 Traditional Buffer Management

Buffer management is another important way to improve performance. In this section, we will look at buffer management in traditional record-oriented database systems.

2.3.1 Overview

Secondary storage is used as the main media to store data, while any available modern operating system can only manipulate data in main memory. Therefore, part of the database has to be loaded into a main storage area before manipulation and written back to disk after modification. A database *buffer* has to be maintained for the purpose of interfacing between main memory and disk[9]. Although modern operating systems allocate some main memory as the *cache* to file systems, and the virtual memory system also uses hard disk to swap active data into main memory and dormant data out to disk, most Database Management Systems (DBMSs) manage their own buffer pools in the user address space and don't take advantage of the file cache and virtual memory management provided by the underlying Operating System (OS) for various reasons[37][38][11]. File systems use *disk pages* as the basic unit for management, where a page's size is usually from 4KB to 64 KB, depending on the OS. The database buffer manager maintains a segment of main memory, which is split into frames whose size is the same as that of disk pages. The major tasks of a database buffer manager are as follows:

- Upon disk page request, search the buffer to locate the page in a buffer frame.

[6]Quadratic probing uses a hash function of the form $h(k) + c_1 \times i + c_2 \times i^2$ to calculate the next hash value if a conflict occurs, where h is the hash function, k is the key to be hashed, c_1 and c_2 are constants, and i is the probe number.

- if the page is not found in the buffer, which is called a *page fault*, it has to be brought into main memory.
- if all buffer frames are in use, a victim page must be selected by a replacement strategy, and the victim page is written back to disk if it's modified since it was brought into the current frame. Then the victim page is discarded, and the requested page is placed in that frame.
- provide FIX-UNFIX[12][7]operations on buffer frames so that a frame in use will not be kicked out by replacement algorithms.

Figure 2.1 shows the database system architecture from the perspective of the buffer manager.

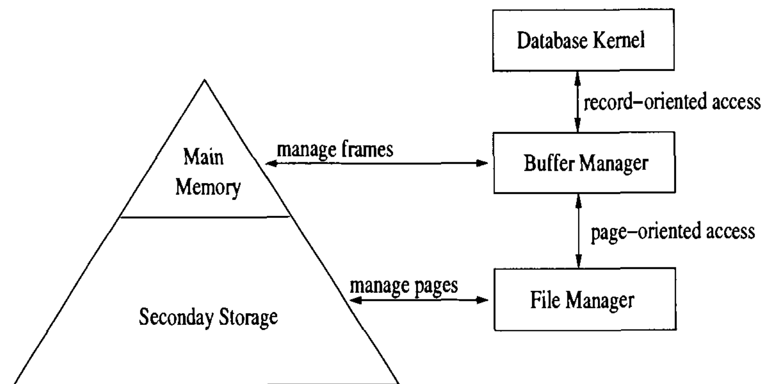

Figure 2.1: Database Buffer Manager

The performance of a buffer system is evaluated by *hit ratio*, which is expressed in the following:

$$hit\ ratio = \frac{number\ of\ times\ data\ found\ in\ buffer}{total\ number\ of\ data\ access} \tag{2.4}$$

2.3.2 Reference Locality

A page request is called a *logical reference.* If the requested page cannot be found in the buffer, a physical disk read needs to be issued to bring the page into buffer, which is

[7]Simply speaking, the FIX operation makes a buffer frame not replacable, while the UNFIX operation makes a buffer frame previously FIXed be available for replacement.

called a *physical reference.* A sequence of references $r_1 r_2 \cdots r_n$, from time t_1 to t_n, is called *a reference string.* Since disk I/O dominates database processing time, given a series of logical references, it's crucial to reduce physical references as much as possible. A well-known and publicly accepted method to do this is by taking advantage of *locality* behavior observed in both the operating system and the database system. *Locality* means that the probability of reference for the recently referenced pages is higher than the average reference probability. There are two models in the literature regarding locality analysis.

Working Set Model

Denning[7][8] proposed the *working set* model to analyze program behavior under virtual memory environment. The locality properties in programs are as follows:

1. Programs use sequential and looping control structures heavily, and they cluster references to given pages in short time intervals.
2. Programmers tend to concentrate on small parts of large problems for moderately long intervals.
3. Programs may be run efficiently with only a subset of their pages in main memory.

Briefly speaking, a program's working set $W(t,T)$ at time t[8] is the set of distinct pages referenced in the time interval $[t-T+1,t]$. The parameter T is called the "window size" since $W(t,T)$ can be regarded as the contents of a window looking backward at the reference string. The working-set size $w(t,T)$ is the number of pages in $W(t,T)$.

Effelsberg[9] indicated that a dynamic page allocation algorithm can be implemented according to the notion of the working set, and pages in the working set will not be selected when making a decision on replacement.

Hot Set Model

The *hot set* model proposed by Sacco and Schkolnick[32] characterizes the buffer requirements of queries in relational databases: relational systems use standard evaluation strategies, so the required buffer space can be estimated before queries are executed. The key idea of the hot set model is that the number of page faults caused by a query is a function

[8]Time t refers to the tth page request, so time t is discrete in this context.

of available buffer space and can be represented by a curve consisting of a number of stable intervals (within each of which the number of page faults is a constant), separated by a small number of discontinuities, called unstable intervals. Figure 2.2 shows two stable intervals and one discontinuity.

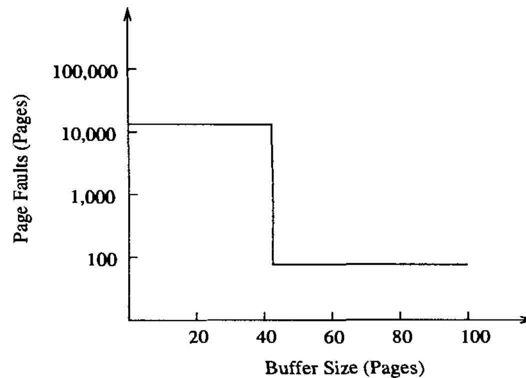

Figure 2.2: Fault curve for a join computed by nested scans using sequential scans.

This model has the following usages in buffer management:

- Allows the system to determine the optimal buffer space to be allocated to a query. For instance, to compute a join of two relations, $R1$ and $R2$, which have $P1$ and $P2$ pages respectively, if a nested loop is used to read the pages of these two relations, $1 + |P2|$ would be an optimal buffer size for the query, in which $|P2|$ buffer pages are used to contain all pages of $R2$, and one buffer page to contain all pages of $R1$ iteratively.

- Can be used by a query optimizer to derive efficient execution plans accounting for the available buffer space. Using the example above, if $|P2| < |P1|$, then the nested loop should be reversed and allocate $1 + |P1|$ buffer frames for the query.

- Can be used by a query scheduler to prevent thrashing.

2.3.3 Replacement Strategies

The working set and the hot set models are suitable for buffer allocation. In this section, we will look at another important aspect in buffer management: *replacement strategy.* When

the buffer pool is full, a replacement strategy must be applied to find a victim page to be replaced. Although more or less different from each other, replacement algorithms all share the common property that the history of page access is used to predict future page access. In the following we will review a few of widely used replacement algorithms.

FIFO

The First-In-First-Out (FIFO) strategy assumes that the first referenced page will most likely not be re-referenced in the near future, so the page with the oldest age will be replaced first. A FIFO buffer pool is maintained as a queue: if the requested page is not in the queue, the queue head is removed from the buffer, and the new page is appended to the end of the queue, so that the former second page in queue becomes the head, and the new page becomes the end; if the requested page is in the queue, the queue remains unchanged.

The advantage of FIFO strategy is that it doesn't need much extra space for bookkeeping the queue information: two pointers are enough to implement a circular queue; the disadvantage of FIFO is that it doesn't take recent references into account, and thus doesn't reflect real-time locality changes.

LFU

Least Frequently Used(LFU) strategy tries to remember the history of page requests and keep the most frequently used pages in a buffer pool. There are two kinds of LFU: *Perfect LFU* and *In-Cache LFU*. For Perfect LFU, each page contains a counter that maintains a number indicating how many times it has been accessed so far. If the requested page is in the buffer, its counter is incremented by one; if the page is not in the buffer, the page is read from disk and its counter is incremented by one. If the counter is not greater than the lowest counter value of the buffered pages, the page is written back to disk and discarded. If the counter is greater than the smallest counter value of the buffer pages, the page with the smallest counter value is written back to disk and the new page is inserted into its position as per its counter value. For In-Cache LFU, when a page is read into the buffer pool, its counter value is set to 1, then this value is incremented by one when it's re-referenced. When a replacement is needed, the page with the lowest counter value is replaced. LFU reflects the temporal locality of reference during a long time period.

The advantage of LFU is that it takes the whole history into account and can achieve a

good overall buffer hit ratio; the disadvantage is that some pages may get a large number of references in a short time, and no more later on, but they continue to occupy buffer pool space because of the high counter value. Furthermore, in Perfect LFU, a read operation also requires a write operation, because the counter value needs to be stored persistently, and therefore will actually degrade system performance.

LRU

Least Recently Used (LRU) is based on the assumption that recently referenced pages will be re-referenced in the near future, so LRU reflects the temporal locality of references during a short time period. The LRU buffer pool can be thought of as a stack: the stack contains all the pages that are accessed between timer interval $[t, t + \tau]$, the page on the stack top is most recently referenced (at time $t + \tau$), and the one on bottom is least recently referenced (at time t). If the requested page is in the buffer, it is removed from its current position in the stack and put on the top. If the requested page is not in the buffer, the page on the bottom will be removed from the stack, and the requested page is then read from disk and put on top of the stack.

The advantage of the LRU replacement strategy is that it is good for the hot set changing over time and doesn't incur too much bookkeeping overhead; the disadvantage is that maintaining LRU chain may become the bottleneck in concurrency control.

A generalized version of LRU, LRU-K replacement algorithm, can be found in J. O'Neil, E. O'Neil and Weikum[28].

CLOCK

The CLOCK algorithm attempts to simulate the LRU behavior by means of a simpler implementation. As in FIFO, a selection pointer is circulated in the buffer pool, and a use-bit is added to every buffer page, indicating whether or not the page was referenced during the recent circulation of the selection pointer. The page to be replaced is determined by the stepwise examination of the use-bits. Encountering a 1-bit causes a reset to 0 and the move of the selection pointer to the next page. The first page found with a 0-bit is the victim for replacement. Another name for the CLOCK algorithm is *Second Chance.*

The CLOCK replacement strategy has the same behavior as LRU, but doesn't have concurrent access bottle neck problem.

A generalized version of CLOCK, the GCLOCK replacement algorithm, can be found in Smith[35].

2.4 Buffer Management in Full-Text Database

Compared to proliferating research works on buffer management in traditional databases, research on buffer management is not very active in the full-text database research community. The most obvious reason is that, due to the fuzzy nature of queries submitted by users, there is no precise definition for what answers are right or wrong, so the main stream of research focuses on improving answers' relevance to queries. Another reason is that, because of the information explosion on the Web, full-text databases such as web search engines need to handle databases in size of giga-bytes or tera-bytes. How to efficiently store and construct indices and data is more critical than in traditional databases. In the following we will investigate some buffering/caching schemes in full-text databases.

Cutting and Peterson[6] proposed an efficient and easy-to-implement buffering method that takes advantage of a sort of 'clustered' data access. When updating an inverted index implemented with the B-Tree, since the word frequency follows Zipf's law in documents, all inverted index entries are sorted by word order before the update, so that the inverted index entries residing physically adjacent in the disk pages are updated together. This way less disk I/O is required than updating the inverted index without sorting words.

Jósson, Franklin and Srivastava[13] utilize a feature of queries submitted to a web search engine that users like to refine their queries and submit it to the search engine again if they are not satisfied with the original answers. Two techniques are proposed to improve query efficiency:

1. Buffer-aware query evaluation, which alters the query evaluation process based on the current contents of buffers.

2. Ranking-aware buffer replacement, which incorporates knowledge of the query processing strategy into replacement decisions.

Markatos[23] studied the trace logs of the *Excite*[9] search engine and have concluded that

1. There exists a significant amount of locality in the queries submitted to popular web search engines. Their experiments suggest that one out of three of the queries submitted has been recently submitted by the same or by another user.

2. Medium-sized main memory caches may serve a significant percentage of the submitted queries. Their experiments suggest that medium-sized caches (100MB) can result in hit ratios at around 20% (or even higher for warm caches).

3. Effective cache replacement policies should take into account both recency and the frequency of access in their replacement decisions. Their experimental results suggest that FBR[10], LRU-2[11], and SLRU[12] always perform better than simple LRU which does not take frequency of access into account.

Saraiva, Moura and Ziviani[33] implemented a two-level cache that combines the cache for query results and the cache for the inverted list, while reserving the document ranking. Their trace log shows that submitted queries follow the Zipf distribution.

Xie and Hallaron[42] analyzed the trace logs of *Vivisimo*[13]and *Excite* search engines, and their results show that queries exhibit significant locality, with the query frequency following the Zipf distribution. They argued that for popular queries shared by different users, the results should be cached on the server side. Individual users who submit many queries tend to use a small set of keywords to form queries, so with proxy or user side caching, prefetching based on user lexicon is promising.

Lempel and Moran[20] used several cache replacement strategies to examine the log of queries containing 7,175,151 keywords submitted to AltaVista search engine during the summer of 2001. They found that prefetching improves the cache hit ratio. They proposed a novel cache replacement policy, called *probability driven cache(PDC)*, which is based on a probabilistic model of search engine users. They also found that the query frequency conforms to the Zipf distribution.

Lu and McKinley[22] compared *partial collection replication* and *caching* that can be used to improve full-text database system performance. Caches are used when queries exactly match previous ones. Partial replicas are a form of caching that are used when the query is a

---

[9]Excite is a web search engin and a part of InfoSpace Inc. Excite search engine can accessed at http://www.excite.com

[10]Frequency-Based Replacement[30], in which replacement choices are made using combination of reference frequency and disk page age.

[11]a special case of LRU-K[28], in which replacement choices are made using last K references to disk pages.

[12]Segmented LRU[15], a frequency-based variant of LRU, which partitions LRU stack into three segments. The most recently referenced pages are placed into the topmost segment, and less frequently referenced pages are pushed down to the bottom segment gradually.

[13]Vivismo provides a clustering search engine. Visit http://vivisimo.com for more information.

similarity match with previous ones. Caches are simpler and faster, but replicas can increase locality by detecting similarity between queries that are not exactly the same. They used real traces from *THOMAS*[14] and *Excite* to measure query locality and similarity. They found that, with a very restrictive definition of query similarity, partial replicas improve query locality up to 15% over exact-match caches.

Tomasic and Molina[39] studied different inverted index organizations in distributed full-text database systems. Their research is based on IPSEC database on the FOLIO system at Stanford University, a database of abstracts of the literature on physics, computer science and electrical engineering etc. An inverted index cache is used to speed up processing queries. The policy for the cache is LRU. The inverted index size in this system is relatively small (308 MB). They found that a cache of about 3.8 MB can improve thoughput by about 136%.

As seen from the survey in this Section and in Section 2.2.4, most caching research in full-text databases is focused on caching the inverted index, and few of them studied how the lexicon could be cached when it is not possible to hold the whole lexicon in main memory. Our major contribution in this thesis is that we study and propose a reasonably manageable, fast and scalable lexicon caching scheme.

[14]THOMAS is a database which makes US Federal legislative information freely available to the Internet. Visit http://thomas.loc.gov for more information.

Chapter 3

Lexicon Caching

In this chapter we present a novel lexicon caching scheme for full-text databases, which exploits the skewed data access distribution. Our basic caching units are individual terms, in contrast with page caching commonly employed in databases.

3.1 Caching Granularity

Caching is widely used in computer systems to bridge the speed gap between different storage systems. Depending on the storage system's operational unit, caching granularity varies. For instance, most modern CPUs have an on-chip cache whose granularity is a line of main memory cells, which is normally a few machine words. In this section we will investigate what granularity should be chosen for a lexicon cache.

3.1.1 Page Caching

Traditionally, cache granularity in database systems is a *page*, which corresponds to a continuous area on disk and is the basic unit transfered to and from the disk. Page caching is suitable for traditional databases, because it reflects patterns of reference locality observed in database systems. Results of most queries performed on a database are tuples that are physically adjacent, which means caching one or more pages is efficient enough for tuples reference to be satisfied in memory buffer pool.

However, in case of lexicon caching, the situation is different. Most of the time users

submit queries that only involve two or three terms. No matter how the lexicon is implemented, terms are stored in disk pages. But terms in a query usually don't reside in the same disk page. Thus to answer a query two or three disk pages have to be examined to get the corresponding inverted list address. As a result, page caching doesn't exhibit good performance as it does in a traditional database. The behavior of page caching for the lexicon is that its cache hit ratio is linearly proportional to the number of pages resident in main memory. For example, if 90% of lexicon pages are in memory, the hit ratio is also close to 90%. This behavior is not desirable for a lexicon buffer, since it's expected to achieve the same hit ratio with comparably a smaller buffer size.

3.1.2 Hotset in Lexicon

To get a better caching scheme for a lexicon, we need to identify the hotset model in the lexicon access, i.e., what's the *basic unit* of data that comprises the hotset?

As described in Section 2.4, both the word occurrence in documents and the queries submitted to full-text databases approximately follow the Zipf distribution.

Zipf's law, found by George Kingsley Zipf, a Harvard linguistics professor, is the observation that the frequency of occurrence of some event P is related to its rank[1] r as follows: the probability that the event of rank r occurs is approximately given by

$$P_r = 1/r^{\alpha} \tag{3.1}$$

with α close to 1. For example, the population of the largest city is roughly $(1/1^{\alpha})/(1/2^{\alpha}) = 2^{\alpha}$ times the population of the second largest city. Formula (3.1) is referred to as the *Zipf distribution.*

Now we should be able to answer the question posed at the beginning of this section. The basic units comprising the hotset in lexicon access are *individual terms*, instead of *pages* that are found suitable for traditional database buffer management. This observation leads to a more efficient lexicon cache design that is further described in the sections below.

3.1.3 Lexicon Caching Architecture

Based on the discussion in the previous two sections, we propose a term-based lexicon cache structure, in which individual terms are the basic caching units.

[1] The smaller the rank number, the more frequently the event will happen. For instance, rank number 1

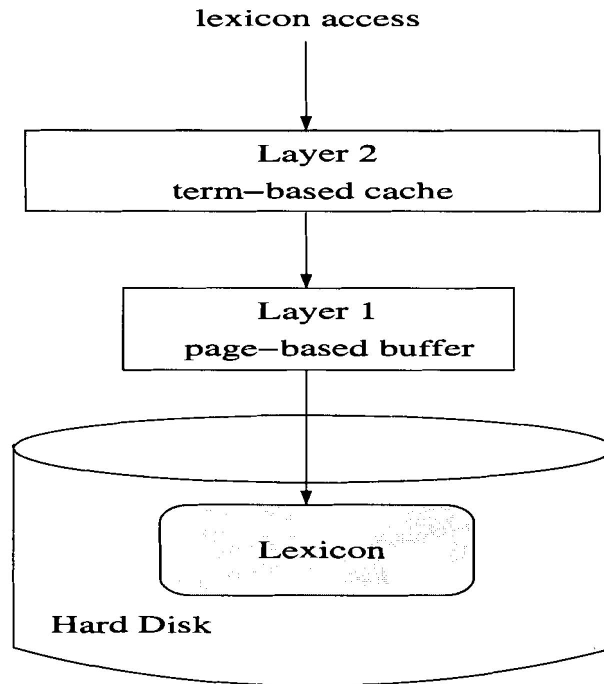

Figure 3.1: Lexicon Cache Architecture.

As shown in the figure above, the lexicon is stored in disk pages, and the lexicon caching structure consists of the following two layers:

Page-based buffer is maintained to cache lexicon pages. This page-based buffer does not need to be big. For instance, if the lexicon is stored in a B-Tree, only non-leaf nodes may be kept in this buffer so that a term access needs at most one physical disk page I/O.

Term-based cache is built upon the page-based buffer. The cache contains individual terms in the current hotset. Therefore most lexicon access can be resolved in the cache.

Although we don't restrict the structure of the lexicon, a B-Tree or similar structure is preferred for the following two reasons:

is the most frequently-happened event.

1. By only buffering the non-leaf nodes of the B-Tree, the layer 1 buffer can be kept very small while still providing fast physical access to lexicon pages. Only one disk access is needed if the requested term cannot be found in either the cache or buffer.

2. B-Tree structure is more extensible in that compression techniques (e.g., prefix or suffix B-Tree) can be easily adopted. It supports not only single key search but also range search, etc. So a B-Tree provides more flexibility to system implementors.

3.2 Memory Management

Before describing the lexicon cache design proposed in this chapter, let's look at some general issues that are normally encountered in designing memory management systems. The design of a memory management system will greatly affect the performance and efficiency of the cache. Wilson, Johnstone, Neely, and Boles[40] present a very good discussion and survey on dynamic memory allocation. The discussion in this section is based mainly on their work.

3.2.1 Memory Fragmentation

Fragmentation is inherent in all dynamic allocation algorithms. Although there is free space available in memory, it can't be allocated to new objects. Traditionally, fragmentation is classified into the following two classes:

1. Internal fragmentation, which arises when a large-enough free block is allocated to hold an object, but the size of the object is very small compared to the block. The unused portion of the block cannot be reused by other objects even if their sizes fit, so this portion is wasted. When there are non-consecutive free blocks in memory, no new object can be stored.

2. External fragmentation, which arises when there are free blocks in memory, but they are too small to hold the next object, so that these free blocks become actually unusable.

Robson[31] makes an observation that the lower bound on the worst case fragmentation is $M \log_2 n$, where M be the amount of live data and n is the ratio between the smallest and largest object sizes.

To avoid internal fragmentation, *splitting* is used to split a large free block into small blocks so that the rest of the block space can be used to hold other objects. *Coalescing* is used to coalesce (merge) adjacent free blocks to form a larger block. No algorithm is available to totally eliminate fragmentation, but some work well in practice, keeping fragmentations small enough to be ignored.

3.2.2 Memory Management Overhead

Most dynamic memory allocation algorithms use a hidden header field within each allocated block to store useful information, e.g., the size of the block, so that the size of the block doesn't need to be passed to block release functions, thereby simplifying a programmer's work. For instance, programming language *C* has a memory allocation function *malloc*, one of whose input parameters gives the required block size, but its corresponding block release function *free* doesn't take allocated block size as an input parameter. This is accomplished by storing block size in the block *header*. To support coalescing, many allocation algorithms also maintain a *footer* field within each allocated block, at the opposite end from the header within the block. The footer contains the block information like block size, an in-use bit indicating whether the block is in use or not. The header contains the same information. When a block is freed, the in-use bit inside the footer of the previous block and the header of the next block is examined to see if they are free to be merged. Normally, the size of the header and footer are just one word, which, in most systems, is 4 bytes (one byte is 8-bits). There are two words used in total for an allocated block. The average object size is normally small — typically 10 words. The overhead of the header is 10% and that for the footer is another 10%. Therefore, the overhead for memory management is quite high.

Management overhead can be reduced by optimization. Standish[36] gives an optimization algorithm that can avoid the footer overhead. When a block is in use, the size field in the footer is actually not needed. The size field is only needed when the block is free, so that its header can be located for coalescing. Only the in-use bit needs to be stored in the footer. Then the size field in the footer can be used to hold real data, thereby reducing the overhead.

3.2.3 Special Requirements for Lexicon Cache Management

When designing a cache for a lexicon, some lexicon properties should be taken into account since they lead to special requirements.

- A lexicon normally consists of millions of terms, so a bad design might result in unacceptable overhead. For example, if the overhead is 4 bytes per term, the overall overhead for a cache capable of storing 5 million terms is 20 MB, which is not acceptable for cache memory.
- The terms contained in a lexicon vary in size, from 1 byte to dozens of bytes. How to dynamically manage them in a fixed size of cache memory is a challenging problem. A naive implementation would just divide cache memory into fixed size cells that can hold the largest terms. But this design would incur excessive internal fragmentation and probably not be practical.
- Unlike dynamic memory allocation, in which blocks are acquired and released periodically, only inserts and updates occur in a lexicon, and replacement is done by a delete followed by a new insert. After the cache grows to a pre-defined size (cache is full), it never shrinks. Also, only one term can be selected to be the victim for replacement, so the selected term must not be smaller than the new term to be inserted. Otherwise, the new term cannot be placed into cache. But if the victim term is much larger than the new term, internal fragmentation will occur.

3.3 Lexicon Cache Design

Our lexicon cache design aims to achieve the following purposes:

- **Fast access to terms**, which requires term update/lookup in the cache to be done quickly.
- **Efficient memory management**, which requires cache memory management not to incur too much internal/external fragmentation and overhead.
- **Fine-grained cache granularity**, which requires individual terms be the basic unit for read, write and replacement.

In the following we present a design for an efficient lexicon cache that fulfills these requirements. The design exploits dynamic hashing to provide fast term lookup, and pages dedicated to storing terms of the same size to avoid fragmentation. We call this cache structure *Dynamic Hashing Chunk Cache*(DHCC). Other viable solutions would be possible as well, but in this thesis we will only focus on DHCC described below.

3.3.1 Dynamic Hashing Chunk Cache

Since a lexicon cache is supposed to store millions of terms of variable size, a good cache memory structure should reduce the potential fragmentation, which would make the system unusable. The idea of DHCC originated from traditional page buffering. A buffer pool is divided into an array of fixed-size frames whose size is the same as a disk page. Since a fixed-size page is the basic unit for replacement, there is no fragmentation at all. If we can organize terms in such a way that terms of different sizes are managed independently, then it's possible to replace the victim term with a new term of the same size.

Cache Architecture

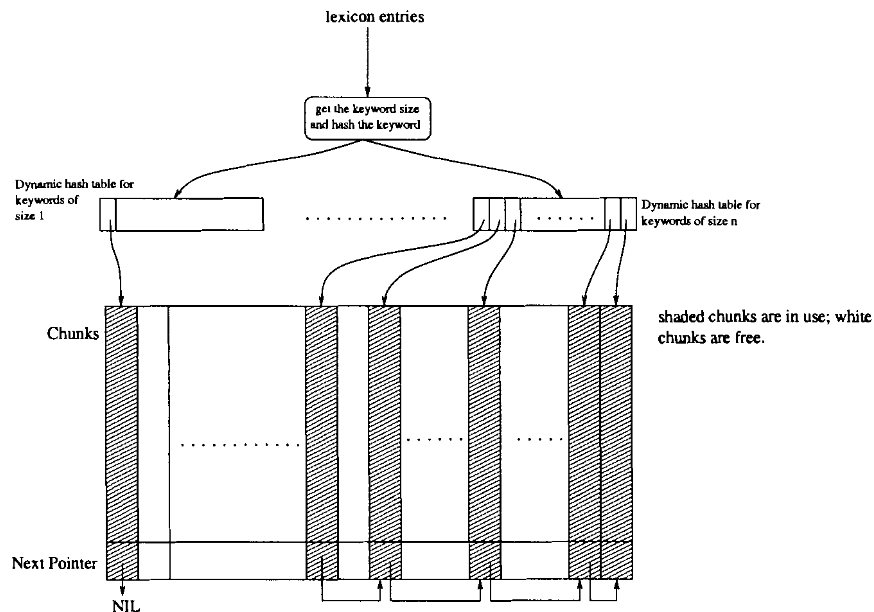

Figure 3.2: DHCC Memory Layout

As shown in Figure 3.2, Dynamic Hashing Chunk Cache is organized in memory in the following way:

1. The cache is a pre-allocated continuous memory space, and divided into fixed size chunks. The size of a chunk is user configurable, but should be big enough to hold dozens to hundreds of lexicon entries. This way the relative space needed for chunk administrative data is as little as possible, thus reducing overall overhead.

2. Every single chunk may only contain lexicon entries with the same size.

3. For every term size, there is a dynamic hash table[2] that hashes a term into chunks, and these chunks are chained together to enable sequential scanning. For example, if the cache is designed to store term sizes of up to 32 bytes, then there are 32 dynamic hash tables and chunk chains, respectively.

4. A dynamic hash table is maintained for every term size so that it can grow without incurring too much performance overhead. The *extendible hash table* [10] is used to implement a dynamic hash table.

Heuristic Chunk Allocation

Given a certain number of chunks, how should we allocate these chunks to each size of terms to store the lexicon entries? For instance, should more chunks be allocated to terms of size 5 than terms of size 6?

In the discussion of lexicon structure in Witten et al.[41], it's shown that in the 538,000-term *TREC* lexicon, the average term size is 7.36 letters, while in the 334,000,000-term *TREC* texts, the average term size is 3.86 letters. The difference in the average lengths is due to the fact that most common words are short, and these words are repeated many times in the text, but they appear only once in the lexicon. This observation suggests that most chunks should be allocated to short terms.

On the other hand, texts or queries can be considered as an infinite stream of terms, $[t_1, \cdots, t_n, \cdots]$. If we could know the ratio of words of certain size in the stream, chunks can then be allocated to each term size proportionally to its ratio. Consider the first q words in this infinite stream, $[t_1, \cdots, t_q]$, which is a sub-sequence of the stream. When q is big

[2] A dynamic hash table is defined as a a hash table whose size can be increased/decreased dynamically.

enough, the ratios of terms of various size in the sub-sequence would be close to those in the infinite stream.

Based on the discussion above, we devise a simple but effective heuristic chunk allocation algorithm as follows:

1. Initially, allocate an empty chunk for each possible term size.

2. For each term in the sequence, hash it and then insert it into its corresponding chunk. If the chunk becomes full, expand it by allocating a new chunk. Expand the hash table also if necessary.

3. Repeat step 2 until all chunks are allocated.

The algorithm is heuristic in the sense that it only uses the beginning portion of the full stream. One may argue that a dynamic hashing table incurs overhead for the growing hash table and chunks, which would degrade system performance. But the growth of the hash table and chunks only happens for the first q terms, i.e., before the cache becomes full. Once the cache becomes full, the hash table becomes static because it won't grow or shrink any more. So the construction time of DHCC can be ignored.

3.3.2 Internal Structure of Chunk

Now let's take a look at how a chunk's internal space is organized.

As shown in Figure 3.3, a chunk is split into two portions:

1. *Chunk Header*, which contains 5 fields: 1) a pointer to the next chunk in the chain. 2) the number of entries currently contained in the chunk. 3) the index number of the root entry in the portion of the entry BST. 4) local depth[3] used in the extendible hash. 5) a clock select pointer used in the CLOCK replacement algorithm.

2. *Entry BST*, which organizes lexicon entries in the form of a binary search tree (BST) [5]. The reason we choose a BST as the data structure to store entries is that it's easy to insert/delete/search nodes in a BST. Furthermore, with a little bit more effort, it's

[3]local depth is used by extendible hashing algorithm to decide when the hash table needs to be extended. It basically determines how many bits of a hash value are used to address the hash table. For example, if local depth is 3, then the most 3 significant bits of the hash value are used to address the hash table. Please see Section 4 of [10] for details.

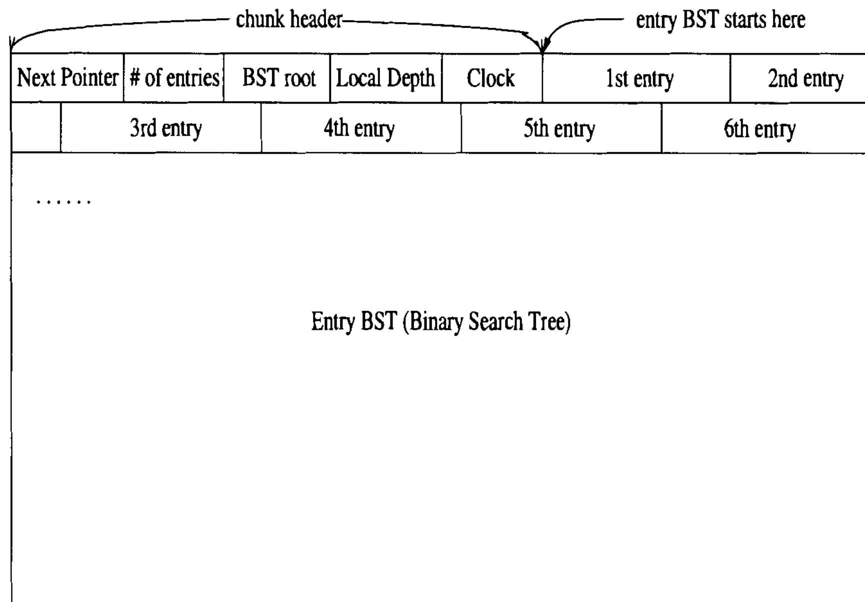

Figure 3.3: Internal Structure of a Chunk.

also easy to keep a BST balanced so that $O(\log n)$ running time is guaranteed for the above operations, where n is the number of nodes in the tree. We will look at the entry BST in detail later in this section.

In the following we will investigate the chunk internals in detail under the assumption that the target system on which the lexicon cache is deployed is a 32-bit system, where a *byte* is 8 bits, a *word* is 32 bits (= 4 bytes), and the addressable memory space is also 32 bits.

Chunk Internal Space Allocation

Given a chunk of a fixed number of bytes, how many bytes should be allocated to the header and how many to the entry BST?

The *next pointer* field in the chunk header doesn't need to store the physical address (4 bytes in 32-bit systems) for the next chunk. Since the chunk pool is allocated in continuous memory space, chunks can be considered as a sequence $[c_1, \cdots, c_n]$. Suppose the beginning physical address of chunk pool is A_c and the chunk's size is c, then for any chunk $c_i, 1 \leq i \leq n$,

its physical address A_{c_i} can be easily calculated:

$$A_{c_i} = A_c + (i-1) \times c \tag{3.2}$$

The *next pointer* in the chunk header may just store the chunk index number in the chunk pool. Normally 2 bytes should be enough since that corresponds to 2^{16} chunks.

The size of the other three fields of a chunk header is determined by the size of the BST, so we will return to this question after we discuss space allocation for the entry BST.

The rest of the chunk space after the header is allocated to the entry BST. A node in a BST contains two pointers, one pointing to its left child node and the other pointing to its right child node. Similar to the next pointer in the chunk header, lexicon entries in a chunk are of the same size, so the child pointers in an entry node may just store the child entry's index number in the entry BST. To minimize space overhead, we allocate one byte to each pointer, which can express $2^8 = 256$ lexicon entries. So two bytes are needed to maintain each node of the BST. The one byte pointer restricts the maximum number of lexicon entries that can be stored in a chunk (256 entries), and the maximum size of a chunk. Also, each of the other four fields in the chunk header need just one byte due to this restriction, and the chunk header needs 6 bytes in total. To implement a balanced BST and replacement algorithm, we need another flag byte for each term to bookkeep the balance data. Therefore, the overhead for the cache memory management is three bytes per term so far[4]. The structure of a BST node is shown in Figure 3.4.

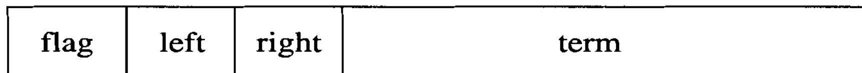

Figure 3.4: BST Node Structure.

A specific chunk, whose size is c bytes and contains lexicon entries of size e bytes, may contain

$$\left\lfloor \frac{c-6}{e+3} \right\rfloor \tag{3.3}$$

entries. If a lexicon entry is in the form of $< t, f_t, D_{I_t} >$ as described in Chapter 1, where f_t and D_{I_t} normally take 4 bytes respectively, then a lexicon entry takes $|t| + 8$ bytes, where $|t|$ is the term length. For example, if the term length is 4 bytes and the chunk size is 3072

bytes (3KB), the chunk may contain up to $\lfloor \frac{3072-6}{12+3} \rfloor = 204$ lexicon entries.

To get the maximum size of a chunk in such a structure, let's assume that the terms only consist of the literals from an alphabet of 36 characters ([0-9a-z][5]). The maximum size of a chunk is determined by the maximum number of 2-byte terms that can be contained in a chunk. When the term size is two bytes, then a lexicon entry consumes 13 bytes, 10 bytes of which are used to store the lexicon entry ($2+8$ bytes long), and 3 bytes of which are used to store *flags, left* and *right* pointers. Since the child pointer can express 256 child entries, the maximum size of the entry BST size is exactly $256 \times 13 = 3328$ bytes. Including the 6 bytes chunk header, the maximum size of a chunk is finally 3334 bytes. Since most 32-bit OSs work better when the memory boundary is multiples of 4 bytes, the maximum size of a chunk would be rounded to 3336 bytes, which is 264 bytes more than 3 KB (=3072 bytes).

Entry BST

The lexicon entries in a chunk will be inserted, searched and replaced frequently, so the data structure of lexicon entries must allow these operations to be done quickly.

Binary search tree (BST) is a data structure consisting of a set of objects that are linked together to form a binary tree. A node in a BST contains a *key* and two pointers, each of which points to the node's left and right child respectively. The nodes in a BST are linked in a way that satisfies the BST *property* : let x be a node in the BST, then

$$key[left(x)] \leq key[x] < key[right(x)] \tag{3.4}$$

where $left(x)$ and $right(x)$ point to node x's left and right child, respectively.

The basic operations in a BST are search, insertion and deletion of a node. Cormen et al.[5] give pseudo-codes for these basic operations, and show that the running time of these operations is $O(h)$, where h is the height of the BST. In the worst case, nodes are inserted in the sequential order so that the BST degrades to a linked list, and the running time for these operations becomes $O(n)$, where n is the number of nodes in the BST. If the nodes are inserted into a BST randomly, the height of the BST is $O(\log n)$, so that the running time

[4]The three bytes overhead can be further optimized. Since the information needed to maintain a balanced BST and replacement algorithm is just a few bits, so these bits can actually be stored in a separate bit array and accessed by the entry index number they represent. This method reduces that one byte to just 2 or 3 bits, so the overhead per term would be around 2.3 bytes.

[5]The characters [A-Z] are normally converted to [a-z] when consulting search/update on the lexicon.

for basic operations also takes $O(\log n)$, which is desirable for performance reason. Some variants of BST guarantees that the height of a BST is $O(\log n)$. For instance, both AVL tree[16] and Red-Black tree[5] keeps the BST balanced so that the tree's height doesn't exceed $O(\log n)$.

For a lexicon cache, another important operation is replacement, which replaces a victim entry in the cache with the new entry. The procedure **BST-Replace(*root, victim-entry, new-entry*)** replaces a *victim-entry* in a BST rooted at *root* with a *new-entry*. We give the pseudo-codes for this procedure as follows:

BST-Replace(*root, victim-entry, new-entry*)
1 BST-Delete(*root, victim-entry*)
2 BST-Insert(*root, new-entry*)

The replace procedure is very simple: line 1 deletes the victim entry from the BST, then line 2 inserts the new entry into it. Since the running time for both **BST-Delete** and **BST-Insert** is $O(h)$, the running time for **BST-Replace** is also $O(h)$, where h is the height of the BST. If the BST is implemented as a balanced tree, the running time for **BST-Replace** can be guaranteed to be $O(\log n)$.

Replacement Strategy

A lexicon cache is supposed to store millions of terms, so it's important to keep administrative data overhead as little as possible. If we choose LRU-like or LFU-like replacement strategies, a list has to be maintained to adjust each term's position in the reference history. This implies that a pointer has to be maintained for every term in cache to point to the next term in the list. The size of the pointer wouldn't be less than 3 bytes. Adding these extra 3 bytes to each BST entry, the space overhead would become 6 bytes per term, which is obviously too much for the cache memory. However, if the average lexicon entry size is dozens of bytes, 6 bytes overhead would not be a big problem. In such cases, LRU-like and LFU-like replacement strategies can still be used.

The CLOCK algorithm can approximate LRU replacement strategy and incurs very little space overhead to maintain the reference history. For this reason, we choose the CLOCK algorithm as the cache replacement strategy. As shown in Figure3.3, for every chunk chain, a clock selection pointer is maintained to scan all terms contained in that chunk. One bit of the flag field in a BST entry is used for the use-bit needed by the CLOCK algorithm.

When a term needs to be placed into cache, a victim entry will be chosen by moving the selection pointer to successive terms stored in the chunk.

Scalability

Our cache design is scalable in that:

- Not like the inverted index size, the lexicon size doesn't grow with the original document size. For instance, as we dicussed in Section 1.2, in Google's prototype implementation, the original document size is 53.5 GB, and the inverted index size is 55.0 GB, while the lexicon size is only 293 MB. This is because, in a certain language, there is a constant upper bound on the number of distinct words.
- As the lexicon size won't be too big, and when term access distribution follows Zipf's law, the cache size that would achieve high hit ratio can be relatively small (as we will see in the emprical studies described in the next chapter), which is desirable and affordable for resource-limited systems.
- Since we use hashing to assign the same size terms to different chunks and each chunk is orginized in BST form, no matter how big the dynamic hash table is, the average time to find a term in the cache is always the same, that is, $O(1) + O(\log n)$, where $O(1)$ is the time to find the corresponding chunk in the hash table and $O(\log n)$ is the time to find the requested term in the chunk's BST. Here we assume that n is the average number of terms contained in fixed-size chunks, then n can be deemed as a constant.

Chapter 4

Empirical Studies

In this chapter we examine the proposed lexicon cache scheme by experimenting with large volumes of data. Section 4.1 describes the design of the experimental system used to study the efficiency and performance of the lexicon cache. Section 4.2 gives the expected cache hit ratio with a theoretical analysis of the CLOCK algorithm. Section 4.3 shows the experimental results, which are then compared with the theoretical predictions.

4.1 Experimental System

To evaluate the design of our lexicon cache, we implement a system on which the experiments are conducted.

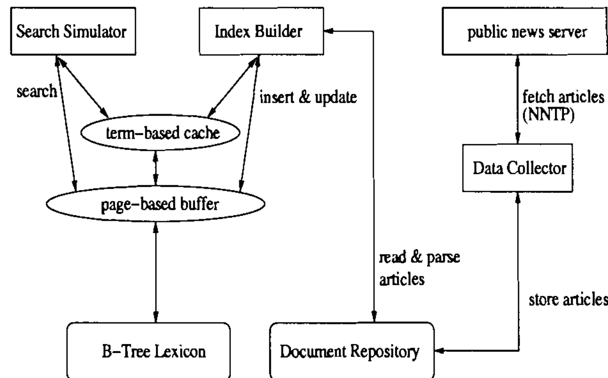

Figure 4.1: Experimental System Components

As shown in the figure above, the experimental system consists of six components:

Document repository that contains all documents in raw formats.

Data collector that fetches *articles* from public *newsgroups* via protocol NNTP[14], and stores the fetched articles in the documents repository residing on hard disk.

Index builder that parses articles in the document repository to extract individual terms and populate these terms into the B-Tree lexicon stored on the hard disk. An experiment performed by the index builder can observe the cache insert/update behavior.

Search simulator that follows the Zipf distribution to generate a sequence of terms to be searched in the lexicon to validate the cache search behavior.

Term-based cache that caches individual lexicon entries. It accesses the lexicon stored on disk through a page-based buffer.

Page-based buffer that is used to buffer lexicon pages.

Index Builder and Search Simulator may access the lexicon either through the term-based cache, or through the page-based buffer directly. In this way we can compare the performance between a term-based cache and page-based buffer. The experimental system is written in C++ with about 8,000 lines of code.

The following table summarizes some relevant parameters of the computer system on which the experiments run:

Operating System	Linux, kernel 2.4.7
CPU	AMD Athlon 800 MHz
Physical Main Memory	768 MB
Disk Page Size	4 KB

Table 4.1: Parameters of Experimental System

4.2 Theoretical Analysis

To better understand the experimental results, we study in this section the CLOCK algorithm from theoretical perspective so that we can then compare our experimental results with the theoretical predictions.

Nicola, Dan and Dias[27] developed an approximate analytical model for the CLOCK replacement algorithm under *Independent Reference Model* (IRM)[1] for skewed data access. A simple analysis and a refined analysis are carried out in their work, but we will only look at the simple analysis, since the refined analysis is computationally intractable when the total number of pages in the database is big (in the case of a lexicon, it's the number of terms, which is normally millions).

The model assumes that the database is composed of P partitions, and the size of partition p is S_p pages. Access to database pages follows the IRM model and the probability of accessing a page of partition p is r_p. When a page in partition p is brought into the buffer, an initial weight I_p is assigned to it. If a page request cannot be satisfied in buffer, the weight of the buffer page pointed to by the selection pointer is examined. If the value is 0, this page is replaced by the new page; if the value is not 0, the page's weight is decremented by 1 and the selection pointer advances to the next page in the buffer until a page with 0 weight is found.

Let n_p $(1 \leq p \leq P)$ be the steady-state average number of pages of partition p in the buffer, then the buffer hit ratio h_p for this partition is

$$h_p = \frac{n_p}{S_p} \tag{4.1}$$

The overall buffer hit ratio is

$$h = \Sigma_{p=1}^{P} r_p h_p \tag{4.2}$$

and the overall buffer miss ratio is

$$m = 1 - h \tag{4.3}$$

If the buffer size is B pages, then the following equation holds:

$$\Sigma_{p=1}^{P} n_p = B \tag{4.4}$$

The model then uses a Markov chain to represent the state of an arbitrary buffer page at the moment of a random page request, and assumes that in a clock cycle, the number of

[1]Under IRM, each buffer page access is independent of all previous page references.

buffer misses equals the number of buffer pages with 0 weight encountered in that cycle. The number of buffer pages with 0 weight is denoted as n_0, then the following equation can be derived[2]:

$$n_p = S_p\left(1 - \frac{1}{(1 + \frac{n_0}{m}\frac{r_p}{S_p})^{I_p+1}}\right) \tag{4.5}$$

The value of $\frac{n_0}{m}$ can be solved by substituting equation 4.5 into equation 4.4, and the resulting equation can be solved with the bisection method[3]. After getting the value of $\frac{n_0}{m}$, the hit ratio for each partition and overall buffer hit ratio can be computed with equations 4.1 and 4.2.

In the case of a lexicon cache, each term is deemed as a partition, which means $S_p = 1$. The initial term weight is 1 for all terms (CLOCK algorithm), which means $I_p = 1$. Then equation 4.5 can be simplified as

$$n_p = h_p = 1 - \frac{1}{(1 + \frac{n_0}{m} r_p)^2} \tag{4.6}$$

which is used to compute the expected hit ratio for a lexicon cache. The probability of accessing terms, r_p, follows Zipf's distribution.

4.3 Experiments

The performance experiments consist of two parts: the *lexicon build* and the *lexicon search*. In the experiments of the lexicon construction, we will build the lexicon from the document repository by using different sizes of cache and buffer memory, and compare their performance. In the experiments of lexicon search, we will study the cache and buffer performance for term lookup by simulating the queries with the Zipf distribution submitted to a full-text database. We will also compare the hit ratios obtained in the experiments with the theoretical ones.

---

[2] Refer to [27] for detailed derivation process.

[3] Suppose we have a continuous function $f(x) = 0$, where x is a real number. There can be one or more values for x that satisfy this equation. The bisection method works by assuming that we know of two values l and r such that $f(l) < 0$ and $f(r) > 0$, then there must be at least one value v that falls between l and r such that $f(v) = 0$.

4.3.1 Data Statistics

Before studying the experiment results, let's take a look at the data statistics with regard to the document repository, lexicon terms and the lexicon B-Tree. Although most information shown below can be observed only after all the documents have been processed (i.e., after the lexicon is built), it will help us to better understand the experimental results in the subsequent sections.

size of the document repository	1.74 GB of data
total number of newsgroups	2,135
number of newsgroup articles	1,377,044
number of all the terms in the repository	89,112,911
number of distinct terms	1,373,059
min size of a term	1 byte
max size of a term	32 bytes

Table 4.2: Document Repository Statistics.

Table 4.2 shows that there are 1.74 GB of data obtained from 2,135 newsgroups. Many newsgroups deal with just one topic, i.e., most articles posted to a newsgroup focus their discussions on a specific field. For example, articles posted to newsgroup 'van.forsale' are mainly with regard to buy&sell second-hand articles in the Greater Vancouver Area. We store all articles belonging to one newsgroup into its own file, so there are 2,135 files in the respository. The maximum size of a term is pre-determined to restrict the largest term size and also to simplify the parsing when building the lexicon.

The lexicon is built as follows:

1. Get every single term by parsing the files.

2. Check if the term's lexicon entry is in the cache. If yes, update the lexicon entry's information (increment frequency by one); if not, read the lexicon entry from the disk into the cache if it's already in the B-Tree, or add a new lexicon entry into the B-Tree and the cache if it's a new term.

3. For each term access, increment the cache hit count by one if it's found in the cache; otherwise, increment the cache miss count by one.

size	count	percentage	size	count	percentage
1	36	0.00002	17	3,944	0.00029
2	1,286	0.00093	18	2,948	0.00214
3	46,431	0.03381	19	2,210	0.00161
4	247,417	0.18019	20	1,599	0.00116
5	222,863	0.16231	21	1,126	0.00082
6	186,338	0.13571	22	926	0.00067
7	155,262	0.11308	23	736	0.00053
8	160,570	0.11694	24	563	0.00041
9	99,261	0.07229	25	304	0.00022
10	81,099	0.05906	26	276	0.00020
11	51,522	0.03752	27	156	0.00011
12	36,403	0.02651	28	167	0.00012
13	24,850	0.01810	29	114	0.00008
14	19,498	0.01420	30	85	0.00006
15	13,431	0.00978	31	89	0.00006
16	11,488	0.00837	32	61	0.00004
total count					1,373,059

Table 4.3: Distinct Terms Statistics.

Table 4.3 shows the statistics for the distinct terms. The *size* column shows the size of a term, and the *count* column indicates that, among all the distinct terms, how many have that size. The *percentage* column is computed by $\frac{term\ count}{total\ count}$. Our heuristic chunk allocation algorithm should allocate chunks to various size of terms with the ratios close to the ratios shown in this table. Table 4.3 also shows that the short terms comprise the majority of terms in the lexicon. There are not many terms whose sizes are larger than 16 bytes.

Table 4.4 shows the statistics for the resulting lexicon B-Tree. We can see that due to the large fan-out of a B-Tree node, both the height of the B-Tree and the number of the non-leaf disk pages are small.

In all of our experiments, 3 KB is chosen for the size of a chunk. MD5 is chosen as the hash function, the first 4 bytes of whose output are used as the hash value.

4.3.2 Construction of Lexicon

Table 4.5 shows the cache hit ratios for various cache size when building the lexicon. The *cache size* column shows the size of the cache measured in KB. The *#chunks* column shows

size of the B-Tree	46.21 MB
height of the B-Tree	3
total number of disk pages	11,830
number of non-leaf disk pages	66
number of leaf disk pages	11,764

Table 4.4: Lexicon B-Tree Statistics.

the total number of chunks contained in the cache. The *#terms* column shows the total number of terms contained in the cache. The *access to cache* column shows the total number of cache accesses. Because we limit the maximum term size to 32 bytes and the maximum term size in the cache is also 32 bytes, every access to terms goes to the cache first. This is why the numbers in this column for various cache sizes are all the same, i.e., 89,112,911, which is also the total number of terms in the document repository (see Table4.2). The *cache hit* column shows the total number of terms found in the cache. The *hit ratio* column is calculated by $\frac{access\ to\ cache\ -\ \#terms}{cache\ hit}$[4].

cache size (KB)	#chunks	#terms	access to cache	cache hit	hit ratio
126	42	5,682	89,112,911	75,403,833	0.846
255	85	13,196	89,112,911	82,309,910	0.924
510	170	27,976	89,112,911	84,607,217	0.950
1,023	341	57,332	89,112,911	85,804,980	0.963
5,115	1,705	294,298	89,112,911	87,347,084	0.983
15,345	5,115	878,285	89,112,911	87,664,178	0.993
25,575	8,525	1,285,376	89,112,911	87,726,505	0.998
35,805	11,935	1,371,937	89,112,911	87,739,586	0.999

Table 4.5: Cache Hit Ratio (Lexicon Construction).

From table 4.5, we see that the cache hit ratio is quite high even for a small size cache when the lexicon is constructed. This looks too good to be true at a first glance. For example, for a cache size of 510 KB, which is about 1% of the whole lexicon, the hit ratio is as high as 94.9%! But if we look at the number of terms in the cache, we can explain

[4] i.e., the cache misses due to the cache fill-up are not counted, since at the beginning the cache is empty, and counting cache misses when the cache is not full doesn't accurately reflect the hit ratio.

cache size (#terms)	hit ratio
5,682	0.902
13,196	0.927
27,976	0.945
57,332	0.959
294,298	0.985
878,285	0.996
1,285,376	0.999
1,371,937	0.999
α	1.27

Table 4.6: Theoretical Cache Hit Ratio (Lexicon Construction).

why. For the 510 KB cache, there are 27,976 terms contained in the cache. Since we already know that the term occurrence in documents follows Zipf's law, the theoretical hit ratio can be easily computed by using Equation 4.6. Then it's found that the cache hit ratio approximates the Zipf distribution with $\alpha \approx 1.27$. The theoretical hit ratios are shown in Table 4.6 The experimental and theoretical hit ratio are compared in Figure 4.2 on page 46.

buffer size (KB)	buffer access	buffer hit	hit ratio
5,120	89,112,911	26,179,751	0.294
15,360	89,112,911	57,468,521	0.645
25,600	89,112,911	76,490,065	0.858
35,840	89,112,911	85,292,872	0.957
46,080	89,112,911	89,099,350	0.999

Table 4.7: Buffer Hit Ratio (Lexicon Construction).

Noteworthy is the comparison of the hit ratios for cache scheme and buffer scheme. Table 4.7 shows the hit ratios for the case that only the buffer is used. The cache and buffer hit ratios are then compared in Figure 4.3 on page 47 with the data from Table 4.5 and Table 4.7. We can see that, with the same amount of memory, the cache hit ratio is much higher than the buffer hit ratio. Figure 4.3 also shows that the curve for the buffer hit ratio follows some kind of power law function, instead of a linear function as we claimed in Section 3.1.1. This is because our lexicon is empty at the beginning. So for a buffer that can hold n terms,

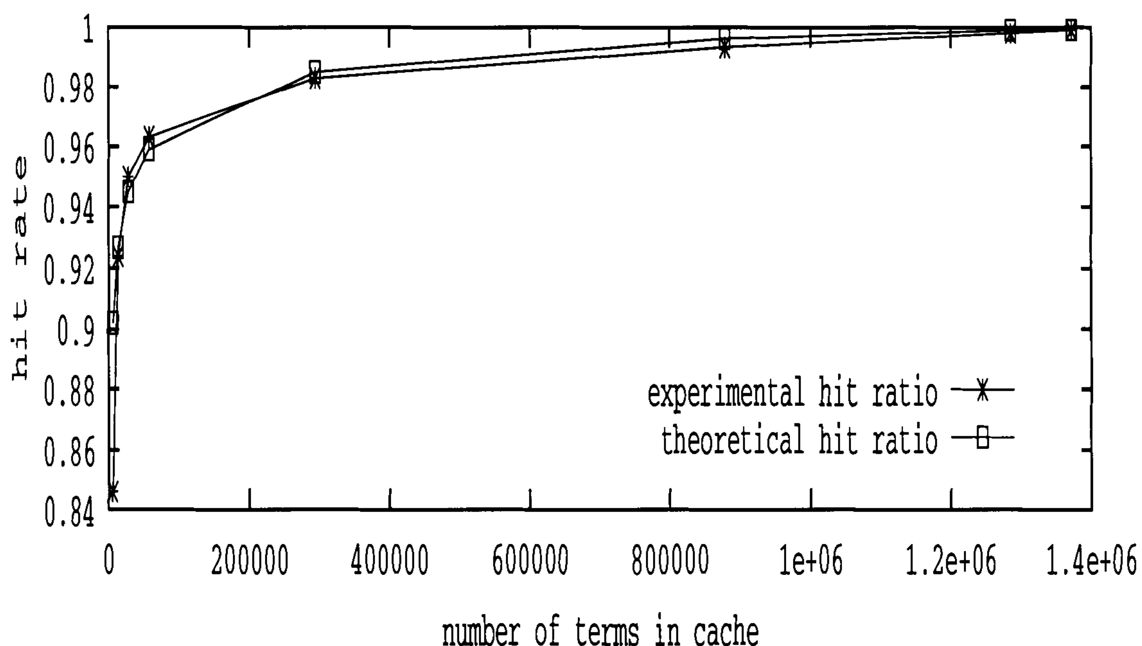

Figure 4.2: Experimental Hit Ratio vs. Theoretical Hit Ratio (Lexicon Construction)

the first new n terms don't result in a buffer miss. Therefore, the buffer hit ratio is actually higher than it should be with the real work load. In practice, the lexicon should be *stable* and changes little during a relatively long period, i.e., after running the system for a while, all possible terms are collected in the lexicon, so normally there are few new terms to be added to it. Therefore, for a buffer that can hold n terms, it's not guaranteed that the first n terms can be found in the buffer, since these n terms are distributed to all the nodes, the number of which is equal to the number of B-Tree leaves. We will see this phenomenon in the experiments of lexicon search. However, this doesn't apply to the cache scheme, since the cache's basic storing unit is a term, not a block containing many terms.

The next thing we want to compare is the response time for the buffer and cache scheme. If the response time for the cache scheme is not significantly faster than the buffer scheme, using term-based cache doesn't make any difference even if it has much higher hit ratio. Table 4.8 on page 48 and table 4.9 on page 48 show the response time for different sizes of cache and buffer, respectively. We can see that the cache response time is generally much

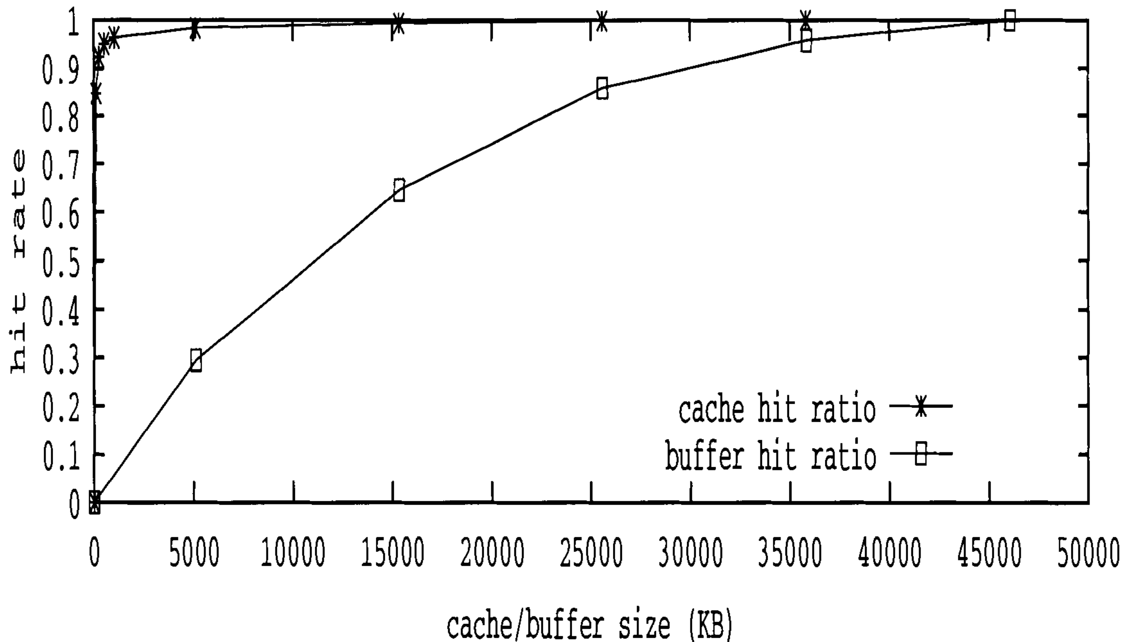

Figure 4.3: Cache Hit Ratio vs. Buffer Hit Ratio (Lexicon Construction)

faster than the buffer. For instance, a 126 KB cache is even about 4 times faster than a 5 MB buffer, and a 5 MB cache is about 15 times faster than a 5 MB buffer! We also notice that the response time in these two tables appears not decreasing consistently when the cache/buffer size increases, e.g., the response time of the 35,805 KB cache is slower than that of the 25,575 KB cache, and the response time of the 35,840 KB buffer is slower than that of the 25,600 KB buffer. Two factors may result in this consequence:

1. The Linux operating system does its own disk buffering to speed up the file system operatoins. So it's possible that most of B-Tree pages are already buffered in main memory, which makes our buffering almost useless. This phenomenon will be actually verified when we measure response time in the experiments of lexicon search later in the next section.

2. When constructing the lexicon B-Tree, new keywords need to be inserted into the lexicon, which requires the split of B-Tree nodes in buffer or chunks in cache etc. These administrative operations may dominate the response time if most of B-Tree pages are

already buffered by the operating system itself. So sometimes smaller cache/buffer size may have faster response time.

Due to these factors, our measure of response time is not accurate, but it should roughly reflect the response time for these two schemes in general.

cache size (KB)	response time (second)
126	15,212
255	8,714
510	7,794
1,023	7,551
5,115	4,967
15,345	3,756
25,575	3,138
35,805	3,445

Table 4.8: Cache Response Time (Lexicon Construction).

buffer size (KB)	response time (second)
5,120	60,489
15,360	58,401
25,600	51,862
35,840	52,956
46,080	50,904

Table 4.9: Buffer Response Time (Lexicon Construction).

Now let's take a look at how our heuristic chunk allocation algorithm works with the real work load. Table 4.10 on page 53 shows the statistical data for the 1023 KB cache (close to 1,024 KB = 1 MB) The column *term size* shows the size of a term, the column *#chunks* shows the number of chunks allocated to that size of terms, the column *#terms* shows the number of terms of that size contained in the cache, the column *global depth* shows the dynamic hash table size defined by extendible hash, and the *percentage* column is calculated by $\frac{\#terms}{total\ \#\ of\ terms\ in\ cache}$. Compare the percentages in each row of this table with that of Table 4.3. We can see that they are very close, which means our heuristic algorithm works very well for building the lexicon. From this table we can also see that most of *#chunks*

in each row is approximately $2^{global\ depth}$, which means our hash function evenly distributes terms to chunks and the selected hash function works very well too.

4.3.3 Lexicon Search

Because we don't have real work load to evaluate lexicon search performance of the cache, a simulation experiment was performed instead. The simulation works as follows:

1. Randomly assign a unique rank to each term in the lexicon.
2. Run an iteration of searches to the lexicon. Each search contains a single term. The access frequency to a term corresponds to its rank with Zipf's distribution. We choose $\alpha = 1.0$ for the simulation.
3. For each term access, the cache hit is incremented by one if the term is found in the cache; otherwise, the missed term is brought into the cache.

Table 4.11 on page 54 shows the simulation results. The meaning of each column is the same as that of Table 4.5. In this table, the total number of cache access is 30,000,001 times for all cache sizes. This number is also the the number of searches in an iteration. One thing to notice in this table is that after the cache hit ratio reaches 0.95, it increases very little with the increase of the cache size, and the number of terms contained in the cache increases very little as well. The reason for the latter is probably that the random number generator used in our simulation doesn't work perfectly so that some terms are never accessed. The same reason could explain the former because if the random generator is not perfect, the generated search sequence doesn't strictly conform to Zipf's distribution. The number of chunks in the last row is 10,852, but in this round of the experiment, the actual cache capacity is 11,935 chunks, which means that not all chunks were used due to the same reason. This is also why it is the last experiment for the lexicon search, because increasing the cache size further doesn't make any difference. Another possible factor might be that the clock select pointer is only circulated within each chunk, not in the whole cache. So our clock replacement implementation is also not strict. How these factors or others may affect the hit ratio is beyond the scope of this thesis and may be answered in future work.

The theoretical hit ratios are shown in Table 4.12 on page 54. The experimental and theoretical hit ratio are compared in Figure 4.4.

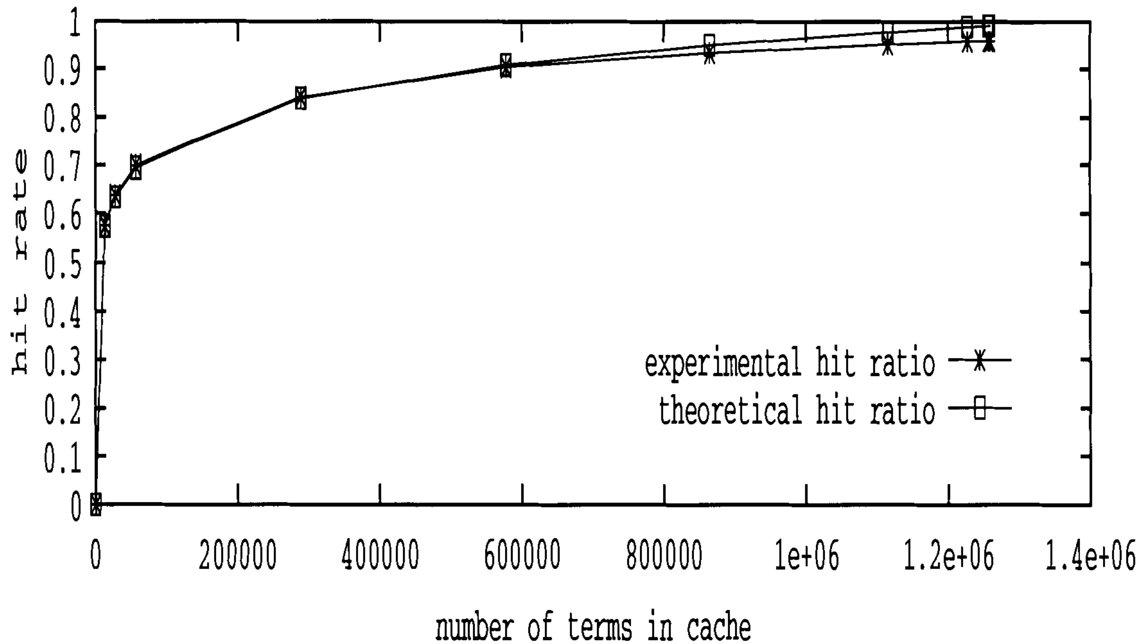

Figure 4.4: Experimental Hit Ratio vs. Theoretical Hit Ratio (Lexicon Search)

Table 4.13 on page 55 shows the buffer hit ratio (without a cache) for the lexicon search. Figure 4.5 compares the cache hit ratio and the buffer hit ratio. Figure 4.5 shows that with the same amount of memory, the cache scheme achieves a much higher hit ratio than the buffer scheme. As we predicted in Section 4.3.2, we can also see that the buffer hit ratio follows a linear function, because when doing the searching experiment, the lexicon is not changed and is in a really stable state. The figure shows that the slope of the buffer hit ratio changes after the point where the buffer size is 42,496 KB. It may be explained by the thrashing phenomenon. Since the terms in the hotset are scattered throughout the disk pages of the whole lexicon, if the buffer is not big enough, not all popular terms can be held in the buffer. To access some popular terms that are not currently in the buffer, the buffer frames that contain other popular terms have to be replaced. If the buffer is big enough, all disk pages containing the popular terms can be held in memory so that thrashing doesn't occur very often.

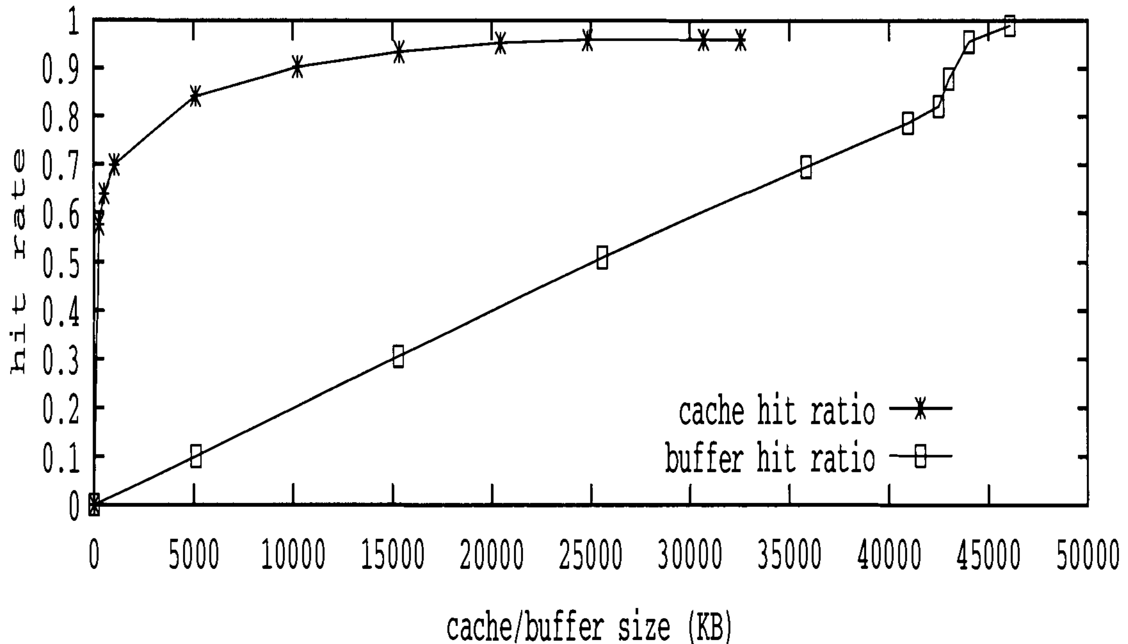

Figure 4.5: Cache Hit Ratio vs. Buffer Hit Ratio (Lexicon Search)

Table 4.14 on page 55 and table 4.15 on page 55 show the response time for different sizes of cache and buffer. It's obvious that the cache response is much faster than the buffer response, e.g., a 5M cache is about 6 times faster than a 5M buffer. We also notice that, in table 4.15 the response time doesn't decrease too much when the buffer size increases significantly. This verifies the phenomenon observed in the previous section that the disk buffering in Linux operating system buffers most of tje B-Tree already, so that our own buffering doesn't cause much additional delay. Altough this makes our measuring of the response time not accurate, it still roughly reflects the fact that the term-based cache is much faster than the page-based buffer.

We also evaluated the heuristic chunk allocation algorithm by studying the term statistics in a cache of a certain size. Table 4.16 on page 56 shows the statistical data for a 1,023 KB cache (same as Table 4.10). The meaning of each column is the same as that of Table 4.10. By comparing it with Table 4.3, we find that, unlike the statistics shown in Table 4.10, the percentages don't exactly match those in Table 4.3. But we can see that the percentages

still roughly match the curve representing the percentages in Table 4.3. Also, most values for the number of chunks are approximately $2^{global\ depth}$. So our heuristic chunk allocation algorithm also works well for lexicon search.

term size	chunk #	term #	global depth	percentage
1	1	36	0	0.00006
2	8	1,286	3	0.00221
3	256	46,420	8	0.07997
4	512	104,448	9	0.17995
5	511	97,601	9	0.16815
6	439	79,020	9	0.13613
7	349	59,330	9	0.10222
8	353	56,833	9	0.09791
9	256	39,168	8	0.06748
10	243	35,235	8	0.06070
11	128	17,792	7	0.03065
12	119	15,827	7	0.02727
13	64	8,128	6	0.01400
14	63	7,686	6	0.01324
15	33	3,861	6	0.00665
16	32	3,616	5	0.00622
17	8	872	3	0.00150
18	8	840	3	0.00145
19	6	612	3	0.00105
20	4	392	2	0.00068
21	3	285	2	0.00049
22	2	184	1	0.00032
23	2	180	1	0.00031
24	2	174	1	0.00030
25	1	85	0	0.00015
26	1	82	0	0.00014
27	1	80	0	0.00014
28	1	78	0	0.00013
29	1	76	0	0.00013
30	1	74	0	0.00013
31	1	72	0	0.00012
32	1	61	0	0.00011
total term # in cache				580,434
total chunk # in cache				3,410

Table 4.10: Cache Terms Statistics (Lexicon Construction).

cache size (KB)	chunk #	term #	access to cache	cache hit	hit ratio
255	85	13,062	30,000,001	17,262,402	0.575
510	170	27,563	30,000,001	19,170,083	0.639
1,023	341	56,860	30,000,001	20,998,555	0.699
5,115	1,705	288,465	30,000,001	25,244,075	0.841
10,230	3,410	577,992	30,000,001	27,091,983	0.903
15,345	5,115	864,699	30,000,001	28,003,749	0.933
20,460	6,820	1,115,151	30,000,001	28,536,907	0.951
24,855	8,285	1,227,104	30,000,001	28,713,651	0.957
30,690	10,230	1,256,240	30,000,001	28,739,581	0.958
32,556	10,852	1,258,958	30,000,001	28,741,043	0.958

Table 4.11: Cache Hit Ratio (Lexicon Search).

cache size (term #)	hit ratio
13,062	0.574
27,563	0.635
56,860	0.695
288,465	0.839
577,992	0.907
864,699	0.949
1,115,151	0.977
1,227,104	0.988
1,256,240	0.990
1,258,958	0.991
α	1.0

Table 4.12: Theoretical Cache Hit Ratio (Lexicon Search).

buffer size (KB)	buffer access	buffer hit	hit ratio
5,120	30,000,001	2,999,639	0.100
15,360	30,000,001	9,176,284	0.306
25,600	30,000,001	15,293,147	0.510
35,840	30,000,001	20,891,955	0.696
40,960	30,000,001	23,593,848	0.786
42,496	30,000,001	24,629,258	0.821
43,008	30,000,001	26,342,583	0.878
44,032	30,000,001	28,617,021	0.954
46,080	30,000,001	29,628,317	0.988

Table 4.13: Buffer Hit Ratio (Lexicon Search).

cache size (KB)	response time (second)
255	7,659
510	7,161
1,023	6,294
5,115	3,759
15,345	1,884
25,575	1,277
35,805	1,214

Table 4.14: Cache Response Time (Lexicon Search).

buffer size (KB)	response time (second)
5,120	18,094
15,360	18,026
25,600	17,724
35,840	17,631
46,080	17,050

Table 4.15: Buffer Response Time (Lexicon Search).

term size	chunk #	term #	global depth	percentage
1	1	34	0	0.00006
2	2	470	3	0.00081
3	68	14,824	7	0.02564
4	512	10,4448	9	0.18071
5	512	97,792	9	0.16919
6	484	87,120	9	0.15073
7	352	59,840	9	0.10353
8	446	71,806	9	0.12423
9	256	39,168	8	0.06777
10	255	36,975	8	0.06397
11	128	17,792	7	0.03078
12	123	16,359	7	0.02830
13	68	8,636	6	0.01494
14	64	7,808	6	0.01351
15	42	4,914	6	0.00850
16	32	3,616	5	0.00626
17	16	1,744	4	0.00301
18	12	1,260	4	0.00218
19	8	816	3	0.00141
20	7	686	3	0.00119
21	4	380	2	0.00066
22	4	368	2	0.00064
23	4	360	2	0.00062
24	2	174	1	0.00030
25	1	85	0	0.00015
26	1	82	0	0.00014
27	1	80	0	0.00014
28	1	78	0	0.00013
29	1	76	0	0.00013
30	1	74	0	0.00013
31	1	72	0	0.00012
32	1	55	0	0.00010
total term # in cache			577,992	
total chunk # in cache			3,410	

Table 4.16: Cache Terms Statistics (Lexicon Search).

Chapter 5

Conclusions and Future Work

In this chapter we summarize the thesis and point out the potential directions for future work.

5.1 Contributions

In this thesis we proposed and evaluated a new caching scheme for the lexicon in a full-text database. Our main contributions include:

- Identified the need for efficiently caching the lexicon in a full-text database, which is an overlooked research area in the current literature. We also indicate that the traditional page-based buffer scheme is not an adequate method to cache the lexicon due to its special characteristics.
- Proposed a term-based lexicon caching method that is totally different from the traditional paged-based buffer method in the database systems. The novelty in our approach exploits two special characteristics of the lexicon: 1) the basic unit handled by the lexicon is individual terms; 2) the terms in the hotset are scattered throughout the whole lexicon and don't exhibit the reference locality found in page-based databases.
- Combined efficiently the different and well-developed technologies into a new framework. The framework doesn't expose any restrictions on what data structures and algorithms must be used. The data structures and algorithms adopted in this thesis should not be thought as a guide for how the framework would be implemented.

The system developers may choose whatever data structures and algorithms they find suitable to their needs.

- Evaluated the new caching scheme thoroughly by conducting extensive theoretical and empirical studies and analyses. The results show that, with the same amount of memory, the term-based cache scheme apparently outperforms the page-based buffer scheme. Our results can be used as the basis for further research in this area.

5.2 Future Work

Our research in this area can be further improved and extended in the following aspects:

- If the trace logs from commercial systems can be used to perform empirical studies, the term-based caching scheme can be better evaluated and compared with the page-based buffer scheme, especially in the case of lexicon search. Furthermore, our experimental data are still very small compared to commercial systems, whose document repository is normally of hundreds of gig-bytes. It would be interesting to see how our caching scheme works with the workload of commercial systems.

- Other data structures and algorithms may be investigated for the term-based caching scheme. The data structures proposed in this thesis are based on the hashing technique, which has some inherent limitations, e.g., the range search or wildcard search is not supported, etc. It is possible to organize chunks in tree-like structures such as B-Tree, thus providing range or wildcard access to terms in the cache.

- The space overhead per term in our caching scheme is over two bytes, which is still a little bit high, especially for the terms of short length. . From empirical studies, we can see that the terms with sizes from 4 bytes to 8 bytes contribute to about 70% of the lexicon. A term entry in the lexicon contains the other 8 bytes to store the term frequency and the disk address of the term's inverted list. So most term entries in a lexicon are about 12 bytes to 16 bytes long. If the space overhead per term is three bytes, then for most terms in the lexicon, the space overhead is from 25% to 19% of the actual data size. It would be a great improvement to the term-based caching scheme if the space overhead can be reduced to less than 2 bytes.

- With the new lexicon caching scheme, it's worth investigating the issue that, given a full-text database with the data and the lexicon of certain sizes and the access frequency distribution, how to best utilize the available memory resources to obtain the best cost-performance.

Bibliography

[1] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

[2] T. Berners-Lee, L. Masinter, and M. McCahill. *RFC 1738: Uniform Resource Locator.* Internet Engineering Task Force, December 1994.

[3] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, April 1998.

[4] Douglas Comer. The Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[6] Doug Cutting and Jan Pedersen. Optimization for dynamic inverted index maintenance. In *Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 405–411. ACM Press, 1990.

[7] Peter J. Denning. The working set model for program behavior. In *Proceedings of the first ACM symposium on Operating System Principles*, pages 15.1–15.12. ACM Press, 1967.

[8] Peter J. Denning and Stuart C. Schwartz. Properties of the working-set model. *Commun. ACM*, 15(3):191–198, 1972.

[9] Wolfgang Effelsberg and Theo Haerder. Principles of database buffer management. *ACM Transaction on Database System*, 9(4):560–595, December 1984.

[10] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing - a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[11] Daniel Fellig and Olga Tikhonova. Operating System Support for Database Management Systems Revisited. http://citeseer.nj.nec.com/fellig00operating.html, 2000.

[12] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Francisco, 1993.

[13] Björn T. Jónsson, Michael J. Franklin, and Divesh Srivastava. Interaction of query evaluation and buffer management for information retrieval. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 118–129. ACM Press, 1998.

[14] Brian Kantor and Phil Lapsley. *RFC 977: Network News Transfer Protocol.* Internet Engineering Task Force, February 1993.

[15] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[16] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, 1973.

[17] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, 1980.

[18] Per-Åke Larson. Linear hashing with separators – a dynamic hashing scheme achieving one-access. *ACM Trans. Database Syst.*, 13(3):366–388, 1988.

[19] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[20] Ronny Lempel and Shlomo Moran. Predictive caching and prefetching of query results in search engines. In *Proceedings of the twelfth international conference on World Wide Web*, pages 19–28. ACM Press, 2003.

[21] Witold Litwin. Linear hashing: a new tool for file and table addressing. In *Proceedings of 6th Conference on Very Large Data Bases*, pages 212–223. ACM, 1980.

[22] Zhihong Lu and Kathryn S. McKinley. Partial collection replication versus caching for information retrieval systems. In *SIGIR '00: Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 248–255. ACM Press, 2000.

[23] Evangelos P. Markatos. On caching search engine results. Technical Report 241, Institute of Computer Science(ICS), Foundation for Research and Technology – Hellas(FORTH), Greece, 1999.

[24] Ken J. McDonell. An inverted index implementation. *Computer Journal*, 20(3):116–123, May 1977.

[25] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the Web. *ACM Transactions on Information Systems(TOIS)*, 19(3):217–241, Jul 2001.

[26] Ajith Nagarajarao, Jyothir Ganesh R., and Abhishek Saxena. An inverted index implementation supporting efficient querying and incremental indexing. http://www.cs.wisc.edu/ ajith/docs/inverted_index.pdf, May 2002.

[27] Victor F. Nicola, Asit Dan, and Daniel M. Dias. Analysis of the Generalized Clock Buffer Replacement Scheme for Database Transaction Processing. *SIGMETRICS Performance Evaluation Review*, 20(1):35–46, June 1992.

[28] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 297–306. ACM Press, 1993.

[29] C. Northcote Parkinson. *Parkinson's Law: The Pursuit of Progress.* John Murray, London, 1958.

[30] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *SIGMETRICS '90: Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 134–142. ACM Press, 1990.

[31] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *J. ACM*, 18(3):416–423, 1971.

[32] Giovanni Maria Sacco and Mario Schkolnick. Buffer management in relational database systems. *ACM Transactions on Database System*, 11(4):473–498, December 1986.

[33] Paricia Correia Saraiva, Edleno Silva de Moura, Novio Ziviani, Wagner Meira, Rodrigo Fonseca, and Berthier Riberio-Neto. Rank-preserving two-level caching for scalable search engines. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 51–58. ACM Press, 2001.

[34] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, Third edition, 1997.

[35] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.*, 3(3):223–247, 1978.

[36] Thomas A. Standish. *Data Structure Techniques.* Addison-Wesley Longman Publishing Co., Inc., 1980.

[37] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[38] Michael Stonebraker. Virtual Memory Transaction Management. *ACM SIGOPS Operating Systems Review*, 18(2):8–16, April 1984.

[39] Anthony Tomasic and Hector Garcia-Molina. Caching and database scaling in distributed shared-nothing information retrieval systems. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 129–138. ACM Press, 1993.

[40] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.

[41] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing gigabytes : compressing and indexing documents and images.* Morgan Kaufmann Publishers, San Francisco, 1999.

[42] Yinglian Xie and David O'Hallaron. Locality in search engine queries and its implications for caching. citeseer.nj.nec.com/article/xie01locality.html, 2002.

[43] George Kingsley Zipf. *Human Behaviour and the Principle of Least Effort: An Introduction to Human Ecology.* Hafner Publications, 1949.

[44] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. An efficient indexing technique for full text databases. In *18th International Conference on Very Large Data Bases*, pages 352–362, Vancouver, Canada, 1992.

[45] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. Searching large lexicons for partially specified terms using compressed inverted files. In *Proceedings of the 19th Conference on Very Large Databases*, pages 290–301, Dublin, Ireland, 1993.