INTERNATIONAL JOURNAL FOR ADVANCE RESEARCH IN ENGINEERING AND TECHNOLOGY
WINGS TO YOUR THOUGHTS.....

To improve performance of indexing using cache for personal dataspaces

Kannu Wadhwa

Seth Jai Prakash Mukand Lal Institute of Technology,Radaur
kannuwadhwa14@gmail.com

Abstract: *The development of relational database management systems served to focus the data management community for decades, with spectacular results. In recent years, however, the rapidly-expanding demands of "data everywhere" have led to a field comprised of interesting and productive efforts, but without a central focus or coordinated agenda. The most acute information management challenges today stem from organizations (e.g., enterprises, government agencies, libraries, "smart" homes) relying on a large number of diverse, interrelated data sources, but having no way to manage their data spaces in a convenient, integrated, or principled fashion. This paper proposes a scheme for improving performance of indexing in personal data spaces using cache.*

Keywords: *personal dataspace, lucene, indexing, DSSP.*

1. INTRODUCTION

A Personal Dataspace includes all the data pertaining to a user on all His local disks and on remote servers such as network drives, email, and web servers. This data is represented by heterogeneous mix of files, emails, bookmarks, music, pictures, persona information stream and so on. In this we focus on personal dataspaces that is the total of all the personal information pertaining to a certain person. DSSP (DATASPACE SUPPORT PLATFORM) is a system abstraction capable of Managing all the data of a particular organization regardless of its format and location.[1-3]PDSMS (PERSONAL DATASPACE MANAGEMENT SYSTEM) is the total of all personal information pertaining to a certain person.[7] This paper is organized as follows. The next section outlines the cache memory .In section 2 we list the current features of our software which is open source and available under Apache. Next section determines the demonstration outline.

2. CACHE MEMORY

Definition and Rationale:

Cache memories are small, high-speed buffer memories used in modern computer systems to hold temporarily those portion of the contents of main memory which are (believed to be) currently in use. Information located in cache memory may be accessed in much less time than that located in main memory. Thus, a central processing unit (CPU) with a cache memory needs to spend far less time waiting for instructions and operands to be fetched and/or stored. For example in typical large, high-speed computers (e.g., Amdahl 470V/7, IBM 3033), main memory can be accessed in 300 to 600 nanoseconds; information can be obtained from a cache, on the other hand, in 50 to 100 nanoseconds. Since the performance of such machines is already limited in instruction execution rate by cache memory access time, the absence of any cache memory at all would produce a very substantial decrease in execution speed. On a read access, it is possible to read the cache line at the same time that the tag is read and compared. On a hit, the required data is readily available, and on a miss, the read data can be discarded awaiting cache line replacement.

However, on a write hit, it is necessary to first read the cache line, modify only the effected bytes before writing the line back to cache. It is possible to pipeline writes to hide the two steps required for a write. It is also necessary to address how the main memory is notified of the changes made to cache lines. In write-through, all writes to a cache line will also be written to main memory. Often buffers are used to process writes to main memory, to avoid blocking the processor on a write. In write-back (orcopy back) the main memory is updated only when a cache line is replaced.[8]

3. TOOL FOR INDEXING: LUCENE

Lucene is an extremely rich and powerful full-text search library written in Java. Lucene can be useed to provide full-text indexing across both database objects and documents in various formats (Microsoft Office documents, PDF, HTML, text, and so on. The basics of using Lucene to add full-text search functionality to a fairly typical J2EE application: an online accommodation database. Roughly, supporting full-text search using Lucene requires two steps: (1) creating a lucence index on the documents and/or database objects and (2) parsing the user query and looking up the prebuilt index to answer the query [5]. lucene provides search over documents. A document is essentially a collection of fields where a field supplies a field name and value. Lucene manages a

INTERNATIONAL JOURNAL FOR ADVANCE RESEARCH IN ENGINEERING AND TECHNOLOGY

WINGS TO YOUR THOUGHTS.....

dynamic document index, which supports adding documents to the index and retrieving document index using a highly expressive search API. Lucene does not in any way constrain document structures. An index may store a heterogeneous set of documents, with any number of different fields which may vary by document in arbitrary ways. Lucene can store numerical and binary data as well as text .What actually gets indexed is a set of terms. A term combines a field name with a token that may be used for a search. For instance, a title like Molecular Biology, 2nd Edition might yield the tokens molecule, biolog, 2, and edition after case normalization, stemming and stop listing. Provides the reverse mapping from terms, consisting of field names and tokens, back to documents. To search this index, we construct a term composed of the field title and the tokens resulting from applying the same stemming and stop listing to the text we are looking for a Lucene search takes a query and returns a set of documents that are ranked by relevancy with documents most similar to the query having the highest score. [6]

Lucene's search scoring algorithm weights results using TF-IDF, term frequency inverse document frequency. Term frequency means that high frequency terms within a document have higher weight than do low-frequency terms. Inverse document frequency means that terms which occur frequently across many documents in a collection of documents are less likely to be meaningful descriptors of any given document in a corpus and are therefore down-weighted.

In Lucene, documents are represented as instances of the final class Document in package org.apache.lucene.document. Documents are constructed using a zero-arg constructor Document(). Once a document is constructed, the methods add (Field able) is used to add fields to the document. Lucene does not in any way constrain document structures. An index may store a heterogeneous set of documents, with any number of different fields which may vary by document in arbitrary ways. It is up to the user to enforce consistency at the document collection level. A document may have more than one field with the same name added to it. All of the fields with a given name will be searchable under that name (if the field is indexed, of course). The behavior is conceptually similar to what you'd get from concatenating all the field values; the main difference is that phrase searches don't work across the concatenated items [5]. Lucene employs analyzers to convert the text value of a field marked as analyzed to a stream of tokens. At indexing time, Lucene is supplied with an implementation of the abstract base class Analyzer in packageorg.apache.lucene.analysis. An analyzer maps a field name and text value to a Token Stream, also in the analysis package, from which the terms to be indexed are retrieved using an iterator-like pattern.[5]

4. DEMONSTRATION OUTLINE

The goal is to build an application that indexes files of different formats and search for specific keywords [4]. For this latest stable version of lucene is downloaded from Apache download mirrors and a new Eclipse project is set up and JAR is included in project's class path. Before executing search queries an index is built against which queries will be executed with the help of a class named Index Writer, which is the class that creates and maintains an index. The Index Writer receives Documents as input, where documents are unit of indexing and search. To create an Index Writer, an Analyzer is required. This class is abstract and concrete implementation used is Simple Analyzer. A class is constructed and the location of the index is provided i.e. where the index data will be saved on the disk ("c:/index/"). Then the data directory is provided, i.e. the directory which will be recursively scanned for input files.

("C:/programs/eclipse/workspace/"). S. The "index" method takes into account the previous parameters and uses a new instance of Index Writer to perform the directory indexing. The "index Directory" method uses a simple recursion algorithm to scan all the directories for files with .java suffix. For each file that matches the criteria, a new Document is created in the "index File with Index Writer" and the appropriate fields are populated. The class is run as a Java application via Eclipse, the input directory will be indexed and the output directory will look like the one in the following image.

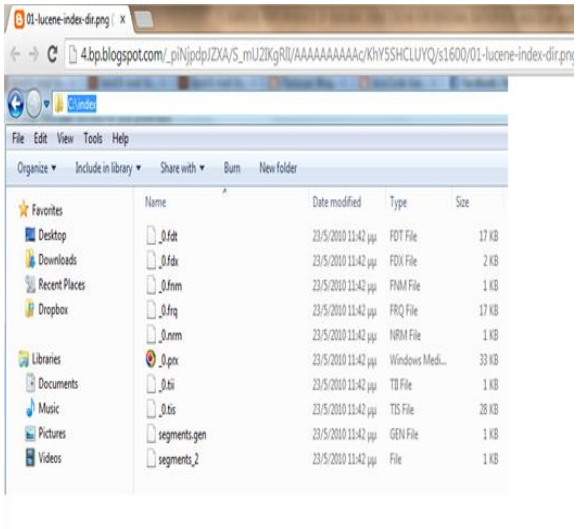

Figure 1: Indexing files

INTERNATIONAL JOURNAL FOR ADVANCE RESEARCH IN ENGINEERING AND TECHNOLOGY

WINGS TO YOUR THOUGHTS.....

For the searching part of the equation, an Index Searcher class is needed, which is a class that implements the main search methods. For each search, a new Query object is needed and this can be obtained from a Query Parser instance. The Query Parser has to be created using the same type of Analyzer that the index was created with. A Version is also used as constructor argument and is a class that is "Used by certain classes to match version compatibility across releases of Lucene", according to the Java Docs. When the search is performed by the Index Searcher, a Top Docs object is returned as a result of the execution. This class just represents search hits and allows us to retrieve Score Doc objects. Using the ScoreDocs we find the Documents that match our search criteria and from those Documents we retrieve the wanted information.

We provide the index directory, the search query string and the maximum number of hits and then call the "search Index" method. In that method, we create an Index Searcher, a Query Parser and a Query object. Note that Query Parser uses the name of the field that we used to create the Documents with Index Writer ("contents") and again that the same type of Analyzer is used (Simple Analyzer). We perform the search and for each Document that a match has been found, we extract the value of the field that holds the name of the file ("filename") and we print it. That's it, let's perform the actual search. Run it as a Java application and you will see the names of the files that contain the query string you provided.

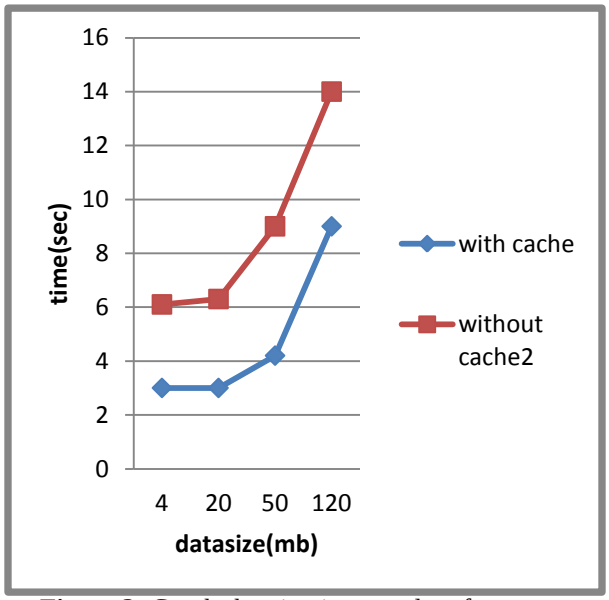

Figure 2: Graph showing increased performance with cache

5. CONCLUSION

The key concept used here is cache memory. The words that is searched the most is put in cache thus the next time it is accessed it is accessed from the cache thus reducing the access time and improving the performance .Also cache replacement policy is used which makes room for incoming data thus keeping only those words in cache that are most frequently accessed.

REFERENCES

[1] Michael Franklin, Alon Hacvey, David Maier "From databases to dataspaces, A new abstraction for information management" ACM SIGMOID Record December 2005.

[2] Michael Franklin, Alon Halvey, David Maier "A First Tutorial on Dataspaces"

[3] Xin Dong, Alon Halvey, "Indexing dataspaces,"

[4] Shaoxu Song, Lei Chen, "Indexing Dataspaces with partitions", Springer, Volume 16, Issue 2, pp 141-170

[5] Wikipedia, Dataspaces http://en.wikipedia.org/wiki/Dataspaces

[6] "A short introduction to lucene" http://oak.cs.ucla.edu/cs144/projects/lucene/

[7] Alon Halevy, Michael Franklin, David Mae "Principle of database system.

[8] Cache Memories http://www.eecs.berkeley.edu/~knight/cs267/papers/cache_memories.pdf