CENG 351 - Data Management And File Structures - Fall 2010
Programming Assignment 2

(Modified) Linear Hashing With Partial Expansions

Due date: 22 December 2010, Wednesday, 23:55

1 Introduction

To get a full understanding of the homework, please read the whole document **twice** *at least and have a look at the given Java classes.*

Dynamic hashing schemes are designed to accomodate files that grow and shrink dynamically. With several improvements in dynamic hashing area, insertion, retrieval and deletion of records are nearly as simple as for classical static hashed files.

Linear hashing is one scheme which was originally proposed by W. Litwin in 1980. It is a technique for gradually expanding/contracting the storage area of a hash file. The file is expanded by adding a new page at the end of the file and relocating a number of records to the new page. As you see in class, this algorithm proceeds by first splitting page 0, then page 1, and so on. One important point is that the splitting not necessarily happens where the collision occurs.

Proposed by P. Larson, "Linear Hashing With Partial Expansions" is a generalization of the original linear hashing algorithm developed by Litwin. Its motivation is that original linear hashing creates a very uneven distribution of the load over the file. (The load of a page that has already been split is expected to be half of the load of a page that has not yet been split) And this slows down retrieval and insertions by creating a large number of overflow records. The idea is splitting a number of 'buddy' pages together. This has the effect of maintaining a more uniform load factor throughout the file and, hence improves performance. According to the results, increasing number of partial expansions improves the retrieval performance while increasing the insertions costs.

2 Directives

In this homework, you are going to implement a modified version of Larson's "Linear Hashing With Partial Expansions" algorithm which is described in the next section. The homework is originally based on the scheme defined by K. Ramamohanarao and John W. Lloyd in their publication named "Dynamic Hashing Schemes". Not needed for the homework, but if you want to look at the details, the papers are available via COW.

We have two file structures in this homework. "Data file" holds the actual records and the "hash file" holds the "PageEntry"s pointing to the records in the data file. Please keep the difference in mind while reading the algorithm and specifications. "Data file" will always stay in the secondary storage and you will never read the whole data file into memory. On the other hand, "hash file" will always be in the memory during the execution. Even we use the word *file* in "hash file", it lives in memory.

3 The Algorithm

The hash file consists of certain pages plus their associated overflow pages. Each page in the primary area has its own, possibly void, chain of overflow pages, which contain records that would not fit into the primary page. The primary area is divided into s groups of g pages.

- s is the segment size (number of pages in a segment)
- g is the group size, that is, the number of segments in the primary area.
- Primary area pages are indexed by $0, 1, ..., gs - 1$.
- The g pages indexed by $j, j + s, ..., j + (g - 1)s$ together form a group of *buddy pages*. Please notice that if g is equal to 1, then you get the original linear hashing algorithm.

As further records are inserted into the file, extra room will be necessary in the hash file to hold the corresponding positions. Normally, it is desirable to maintain a relatively constant load factor for the hash file. Our load factor will be controlled by the *load control* L. This means that after every L insertions, splitting occurs.

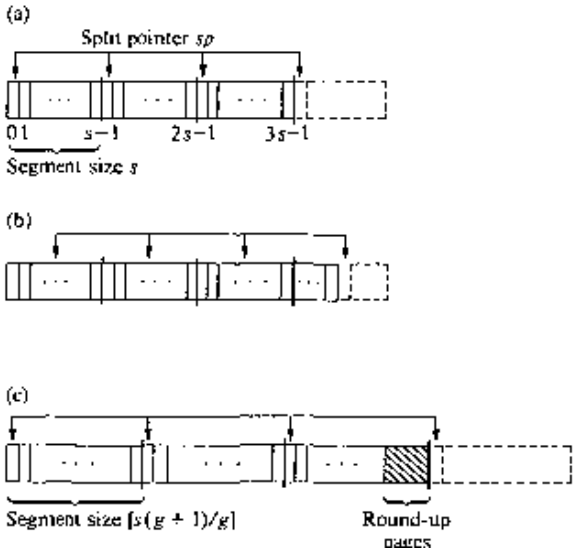

Figure 1: Structure of file at typical stages (for the case g=3): (a) at the start of an expansion; (b) during the expansion; (c) at the start of the next expansion - This figure does not show the overflow pages.

Adding a new page to the hash file is as follows:

- Receiving the record, first you have to insert it to the end of the data file and keep the starting position of this record to put into the hash file. You have to do this by **appending** the record to the data file. You will not read the whole data file and write it back (I will check it ;)).
- Now, you will calculate the hash value of the record and create a “PageEntry” to be inserted to the hash file and insert the “PageEntry” to the hash file.
  - If there is room in the corresponding page (you have 3 rooms max for nodes in each page, i.e. $Bkfr = 3$), just insert the node at the available place. The page must include the “PageEntry” objects sorted increasing order according to the key value.
  - If there is no room in the corresponding page, insert it into an overflow page.
    - If the primary page does not have an overflow page, then create it, add the new node and chain it to the primary page.

    - If the primary page has an overflow page, follow the chain until you find a room for the new node and insert into the first available place. If there is no room in the overflow chain, then create a new overflow page, insert the node and link to the existing chain.

- If you haven't reached to the load control L yet, do nothing after the insertion.
- If you reach to the load factor L, then you will start the splitting phase.

Figure 1 shows a graphical representation of the expansion procedure that you will follow. At the start of an expansion, a pointer called the *split pointer* sp, points to the first group of *buddy pages* indexed by $0, s, ..., (g-1)s$. As shown in Figure 1 (a), $sp = 0$, initially. After exactly L insertions, this group of g buddy pages is split in the following way:

- An extra page, indexed by gs is appended to the end of the primary area of the hash file.
- The records in the pages $0, s, ..., (g-1)s$, plus records in the overflow pages associated with these primary pages are redistributed according to the hash function described below, among the $g+1$ pages.
- The split pointer is then moved forward to the next group of buddy pages $1, s+1, (g-1)s+1$. ($sp = 1$)
- After the next L insertions, these pages are split and so on. Figure 1 (b) shows the file at a typical stage during the expansion.
- As the split operations performed, the split pointer will move to the last group of buddy pages $s-1, 2s-1, ..., gs-1$ and these pages will be split. After this last split, sp returns to the beginning of the file. (so that $sp = 0$) This cycle is called one **expansion**.

During the expansion, the file grew from gs pages to $(g+1)s$ pages and exactly sL records were inserted. Before starting the next expansion, the segment size needs to be enlarged since we have a fixed g during the lifetime of the execution. The new segment size is given by the following formula:

$$s' = \lceil s(g+1)/g \rceil$$

The new hash file size will generally not be divisible by g. (Note that, this problem does not arise in the original linear hashing scheme, since $g = 1$) To make the current file size divisible by g, you will append, if necessary, r extra pages to the end of the hash file, where $0 \leq r \leq g$. The extra pages added to the end are called *round up* pages. These pages will remain empty until the split pointer reaches them. The hash file now consists of s' groups of buddies and the new expansion can begin.

4 Specifications

You are going to implement a dynamic hashing scheme (the algorithm explained above) which holds indexes on the actual data file. The data contains health related information about patients, uniquely identified by TCKimlikNo attribute of records. Length of the record is fixed and all records will be kept in a single file. The structure of the file is given in the following sections.

On top of the data, you are going to create a hash index structure which is formed of a primary and an overflow area. The index files will hold the positions(byte offsets) of the records corresponding to the positions in the data file. We will call an addressable unit in the hash structure as "page", instead of "bucket". And each page is formed of 3 "PageEntry"s maximum (i.e. Bkfr is 3).

4.1 The Hash Function

Intially the hash file consists of gs_0 pages, where s_0 is the initial segment size. The file then goes through a number of expansions easily accomodating any number of insertions. The definition of *expansion* has been stated earlier. The number of the expansions the hash file has undergone is called its *depth d*. Initially, $d = 0$. The current state of the hash file is essentially characterized by the two variables, d and sp.

The function below returns the page address $hash(TCKimlikNo, d, sp)$ of a record. TCKimlikNo is the primary key for the records in the data file. For retrieval, a record will be found either in the page of the primary area with index returned by the function or in the overflow chain of that page.

```
begin hash(TCKimlikNo)
  h := k(TCKimlikNo)
  s := s0;
  for j := 0 to d-1 do
  begin
    h := h mod s + h_j(TCKimlikNo)*s;
    s := ceiling(s*(g+1)/g);
  end;
  if h mod s < sp then h := h mod s + h_d(TCKimlikNo)*s
  return h;
end.
```

In the function above,

- k is the function to take the modulo of TCKimlikNo. This is the ordinary *modulo* operation as you do in class. You know the initial size of the primary area of the hash file, which is $s_0 * g$. So, $k(TCKimlikNo) = TCKimlikNo \bmod (s_0 * g)$. Please keep in mind that this is the initial size of the primary area.
- h_j means h_j and h_d means h_d. This is a sequence of independent hash functions each of which maps from the key space of the records (TCKimlikNo in our case), into the set $\{0,1,...,g\}$. You have to implement this using pseuodorandom number generator in Java. This methodology is used to evenly distribute the "PageEntry"s among the pages in the hash file as the pages are splitted.

The following code can be used to implement h function. So, $h_j(TCKimlikNo)$ means calling the function below with parameters j, $TCKimlikNo$ and g. I share the code to clearly state the hash function. Just a warning, you can calculate the numbers for each TCKimlikNo once and keep in somewhere to look it up when needed. If you use the function defined below as it is, you calculate the series at each call.

```
public int h(int j, long TCKimlikNo, int g) {
    Random r = new Random(TCKimlikNo);
    int limit = g+1;
    for(int i=0;i<j;i++)
        r.nextInt(limit);
    return r.nextInt(limit);
}
```

Please note that during a split operation, it is not necessary to use the full algorithm to compute the new address of "PageEntry"s on the pages being split. Suppose the current segment size s has been stored. Then the new address for each record is given by $sp + h_d(K) * s$. This is essentially the last line of the algorithm.

4.2 Hash File

The hash file lives in the memory and you have to create it at the beginning of your execution(before inserting any records). The hash file is formed of "Page" objects which is defined as in the following:

```
package ceng351.phw2;

public class Page {

    /*maximum 3 nodes*/
    private PageEntry[] nodes;
    private Page overflow;

    public PageEntry[] getNodes() {
        return nodes;
    }

    public Page getOverflow() {
        return overflow;
    }

}
```

The size of the hash file is determined by s and g as stated earlier.

Each page can hold maximum 3 "PageEntry"s and the nodes are kept sorted in TCKimlikNo in the "nodes" attribute of the object. If "nodes" is null then this page does not hold any "PageEntry". If "overflow" is null then there is no attached chain of overflow records for this page. Please note that the overflow area is also a "Page" object and the overflow chains are constructed by linking "Page"s together.

The definition of the class is as in the following:

```
package ceng351.phw2;

public class PageEntry {

    private long TCKimlikNo;
    private long byteOffset;

    public long getTCKimlikNo() {
        return TCKimlikNo;
    }

    public long getAddress() {
        return byteOffset;
    }

}
```

The primary area and overflow area terms are actually abstract in the definition of this homework. The internal structure is totally up to you as long as you implement the methods correctly defined in the interface, "HashIndex". You will use the class definitions given to you by this homework specification. I will also upload them as separate Java classes to COW. You can introduce new set of classes, add anything to these classes, but you cannot modify anything in these given classes. For example, you cannot change or remove any given methods/attributes.

4.3 Data File

The data file is an ASCII file (keeping ASCII characters, not binary data) which includes fixed length records. The records are 100 bytes long each. A record will be given to the corresponding method of the "HashIndex" interface as in the following form:

```
TCKimlikNo|date|ICD10-Code|examination
```

An example is below:

```
12345678912|2010-11-29|N18.0|suspect of end-stage renal disease
```

The length of each field is also fixed except the last one:

- TCKimlikNo: 11 bytes
- date: 10 bytes
- ICD10Code: 5 bytes
- disease: 74 bytes, maximum

Other than the disease field, all fields of a record is fixed and the record length is also fixed. The disease field may have blanks to fill up to 100 bytes of the record. The data file will represent these blanks with ',' (comma character). The size of the data file is unlimited. An example of a record residing in the data file is as follows:

```
123456789122010-11-29N18.0suspect of end-stage renal disease,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
```

An example of the data file holding three records is as follows:

```
123456789122010-11-29N18.0suspect of end-stage renal disease,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,222222222222010-03-04N03.2diagnostically t
here is no problem with this patient but there is string evidence for b
eing sick,,,333333333332009-04-23,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
```

4.4 Operations

In the "HashIndex" interface, you can see the operations that you have to implement. In deed, they correspond to the implementation of retrieval, insertion and update operations. You have to implement additional methods also to get credit because I can only check your implementation by means of those extra methods.

The operations you have to implement are as follows:

- *public void initialize(int L, int s, int g)*: This function passes the required parameters to your environment. This method will be called once before any other method. L is the load control, s is the initial segment size (number of pages in a segment) and g is the group size (the number of segments in the hash file). It returns nothing.
- *public File getDataFile()*: This method should return a reference to the data file.
- *public Page getPage(long TCKimlikNo)*: In this method you have to return the Page object including the PageEntry belonging to the TCKimlikNo.
- *public String getRecord(PageEntry node)*: Given the PageEntry object, this method will return the string representation of the requested record. The return string is in the record form described above.
- *public Page[] getAllPagesInOrder()*: This method will return an array of Page objects. Actually, this method will return your hash file in an abstract manner. The return array should include the pages ordered by their indexes which are the "hash" function outputs.
- *public PageEntry insertRecord(String record)*: This is the insert operation. Inserting the record to the data file means that you have a PageEntry object in a Page object in your hash file for this record. You have to return that PageEntry object.
- *public void updateRecord(String record)*: This will locate the record with the TCKimlikNo, parse record string and update the record fields with the new values. If the record cannot be found, you will do nothing.

5 Technology Specifications

- Java compliance level should be JDK 1.6

6 Submission

- All code returned is expected to be your individual work. You are fully responsible for the outcomes of cheating.
- You will submit a single zip file (hw2.zip) which should include the class "e1XXXXXX.java" under the package "ceng351.phw2". Your class will be "ceng351.phw2.e1XXXXXX". As stated earlier, this class must implement the interface "ceng351.phw2.HashIndex" which is provided to you.
- To be more precise, when your hw2.zip is extracted, only a folder named ceng351 should come up. Inside, there should be a folder named "phw2". Inside phw2, you have your Java classes.
- Following half-implemented main function is expected to run with your submission (This Main function will be in the same folder with ceng351 folder):

```
import ceng351.phw2.*;

public class Main {

    public static void main(String [] argsv) {
        e1395458 hi = new e1395458();
        hi.initialize(60, 50, 3);
        String record = ...;
        h.insert(record);
        h.getPage(12345678912);
    }

}
```

- Please discuss any point about the homework in the course newsgroup.

7 Late Policy

As mentioned in the syllabus, you have 7 days to be used as late submission in all homeworks (except the last homework). After 7 days, late submissions are penalized up to 10% per day.

May it be easy.
–Anıl