Assignment 5

Due at Friday 5 June 2015 at 23:59

Goal

Goal: Implement a simple DNS database based on a B+ tree, and then extend it so that the B+ tree is stored in a file rather than memory.

Resources and links

Download Zip file of starting code and data.

GUI.java (fixed).

test-list.txt of IP-hostname pairs for testing. (test-list-reversed.txt if you switched your test method around.)

Submit your answers.

MarkSheet (pdf)

To Submit

TWO Jar files and source code:

Your working program for the in-memory B+ tree DNS database. (Executable Jar file and source code)

Your working program for the file-based B+ tree DNS database. (Executable Jar file and source code)

Your `report` file

A B+ tree for a DNS database.

Most large databases store their data in files, and access the data as needed. To make this computationally efficient, database management systems (DBMS) need to have indexes into the data to enable them to find a particular record without having to search through the whole database file. Relational DBMS's structure their data in the form of tables of records, where each record contains several fields describing a single entity. A table consists of a large set of records all of the same type.

Typically, the DBMS will have at least one index for each table - on the key field of the table. If the user might want to search for items based on other fields of the table, then the DBMS will have an index for each other such field.

A standard way of implementing an index for a database table is to use a B+ tree.

A Domain Name Server (DNS) is a service that will allow a user to find the IP address (eg `130.195.6.22`) of a given domain-name (eg `bats.ecs.vuw.ac.nz`), or the domain-names associated with an IP address. It is essentially a very simple DBMS that stores a single table database that maps between IP addresses and domain-names,

IP addresses are usually written as four 8 bit numbers (eg 130.195.5.12), but they can be viewed as a single 32 bit integer. Domain names, or Host names, are strings, which consist of a sequence of labels separated by dots. Each label can be up to 63 characters long, and the whole name can be up to 255 characters (including the dots).

Part 1 B+ trees.

For this assignment, you must make a simple Domain Name Server, The DBMS should allow lookup by either IP address or domain-name: given an IP address, it will find and return the domain-name; given a domain-name, it will find and return the IP address. To do this efficiently, it will need two B+ trees, one for each index.

We have provided a `GUI` interface for your program, and top level class for the database. The interface is very simple - it only has a few functions:

- the user can enter an IP address, and the program will display the associated domain-name (or a message if there isn't one).
- the user can enter a domain-name, and the program will display the associated IP address (or a message if there isn't one).
- the user can enter a both an IP address and a domain-name, and the program will allow the user to test whether the pair is already in the database, or to add the pair to both indexes.
- the user can ask to print out (to System.out) a list of all the domain-name - IP address pairs, in order.
- the user can ask to check all the pairs in a test file.

We have also provided a `DNSDB` class that will contain the two B+ trees, and provides the top level interface accessed by the `GUI` class. Most, but not all, of the code in `DNSDB` is written for you.

You must write the two classes to hold the B+ trees:`BPlusTreeIntToString60.java` and `BPlusTreeString60toInt.java`. You must provide functionality for adding key-value pairs to the B+ tree, searching for the value associated with a key, and iterating through all the key-value pairs in the tree (by stepping along the linked list of leaves). You do not need to implement deletion. To work out the right sizes for the internal and leaf nodes (maximum number of keys, or key-value pairs in a node), you should read the second part of the assignment.

Note, it would be possible to write a single generic class if you really want to, but that will make the second part of the assignment a lot harder, so we strongly recommend having two non-generic classes.

We have provided a data file `host-list.txt` of IP addresses and domain-names (tab separated). All the domain-names are guaranteed to be at most 60 characters. There are also guaranteed to be no duplicates in the file: no duplicate entries, no IP addresses associated with more than one domain-name, and no domain-names associated with more than one IP address. Note that these restrictions are a simplification of the full problem, but making B+ trees that handle keys with multiple values is unnecessarily tricky for this course! The program will attempt to load the data in this file into the database (ie, adding each pair and its inverse to the two B+ trees respectively). You can give command line arguments to specify a different file to load.

The program can be bulk tested with the "Run Test" button, which will read all the IP : domain-name pairs in a file and check whether they are in the database. Please don't change the `testAllPairs` method, since we will use this for marking.

Part 2:

When a database is large, the B+ tree must be stored in a file, rather than held in memory. The B+ tree should be stored such that each node in the tree (internal and leaf) is stored in one block of a random access file. Each block will consist of a fixed number of bytes. The file system is able to return any individual block of the file, given an index specifying which block is required.

The internal nodes of a B+ tree contain a collection of keys and a collection of pointers to child nodes. When the B+ tree is stored in memory, a child node pointer will be a reference to the child node object; when stored in a file, a child node pointer will be an integer specifying the index of the block in the file holding the child node.

Node sizes: maximum number of keys and key-values.

The number of keys and key-value pairs in the B+ tree nodes will be determined by the size of the blocks and the size of the keys and values:

Each internal node of a B+ tree has a collection of up to n keys and $n+1$ child pointers, along with some header information. If the header requires h bytes, each key requires$k$ bytes, and each child index requires 4 bytes, then the internal node will require $h + k * n + 4 * (n + 1) = (k + 4) n + 4 + h$ bytes. If the block size is b bytes, then each node can contain at most $(b - h - 4)/ (k + 4)$ keys.

For example, if the header requires 5 bytes, the key is an IP address (k is 4), and the block size is 1024 bytes, then a node can contain up to (1024-9)/8 = 126 keys. On the other hand, if the key is a domain name of up to 60 characters, then a node can contain up to (1024-9)/64 = 15 keys.

Each leaf node of a B+ tree contains a collection of up to m key-value pairs, the index of the next leaf node, and some other header information. If the header requires $h$bytes, the next leaf index require 4 bytes, each key requires k bytes, and each value requires v bytes, then the leaf will require $h + 4 + (k + v) * n$ bytes. If the block size is$b$ bytes, then each node can contain at most $(b - h - 4)/ (k + v)$ key-value pairs.

For example, if the header requires 5 bytes, the key is an IP address (k is 4), the value is a domain name up to 60 characters, and the block size is 1024 bytes, then a node can contain up to (1024-9)/64 = 15 IP-domainName pairs. If the leaf stored a pointer to another file where the full record was stored, then a node could contain up to (1024-9)/8 = 256 pairs.

Note: the value stored in the leaf node may either be the actual value, eg the rest of the fields of the record, or an index into another file where the complete record is stored. Particularly when there are multiple indexes into the same table of records, the full records are stored only once, and the leaves of the trees point to the blocks containing the records. [For the first two stages of this assignment, you can save the actual value in the leaf node. As an optional step in Stage 3, you may choose to save the index in the leaf node to point to the blocks where the complete data records is saved.]

Block file version of DNSDB.

For part 2, you are to make a new version of your DNSDB program which stores all the nodes of the B+ trees in files, and accesses the nodes from the file as needed. (DO NOT OVERWRITE YOUR PART 1 PROGRAM!)

The best way of doing this is to construct code for converting a node from the B+ tree into an array of bytes that can be stored in a block of the file, and code for converting an array of bytes from a block in the file back into a B+ tree node. You can then modify/extend the code in your B+ tree classes so that when it tries to access a node (either the root of the tree, or a child pointer of an internal node), it will read the appropriate block from the file, and turn it into a Node object. If it modifies a node, then it will need to convert the node back into a byte array and write it back to the block in the file. This will involve replacing the child pointers in your nodes by integers representing indexes of blocks. The B+ tree object will no longer consist of a reference to the root node of the tree, but instead will contain a reference to the file, the index of the block containing the root node of the tree, the index of the leftmost leaf of the tree, and other meta information such as the maximum sizes of the nodes, and the numbers of bytes for representing the keys and values. [In general, the first block of the file should contain this meta information about tree, but for this assignment, since you only have two B+ tree files, it would be possible to leave this block out.]

The `find` method will read the root block, convert it into a node, and then work on this node as it did in the in-memory version. When it needs to get a child node, it will do the same thing: given the index, it will read the block and convert it to a node object. The `put` method will do the same thing, but every time it modifies a node, it will also need to convert the node back into an array of bytes, and write it back to the file (at the same index). If the `put` method creates a new node, it will then need to convert it and write it to the end of the file, remembering the index of the new block in order to push the index up to the parent node, along with the key. Note that your program may reconstruct several nodes during a call to `find` or`put`, but they should be stored temporarily in local variables since they do not need to be store permanently - you should not be trying to reconstruct the whole tree from the file!

We have provided the `BlockFile.java` file which defines a class for random-access files that read and write blocks. Your program must use the `BlockFile`class to read and write blocks from the files storing the B+ trees. The size of the blocks is specified when a `BlockFile` is created. The `read(int index)` method reads the `index`th block of the file and returns it as an array of bytes. The`write(byte[] block, int index)` method writes the byte array to the`index`th block of the file. The `write(Block block)` method writes the byte array to a new block at the end of the file and returns the index of that block of the file.

We have also provided the `Bytes` class, which contains various methods for dealing with bytes, including methods for converting sections of a byte array to and from integers and strings.

Your BPlusTree classes should have two constructors: one should be able to construct an index from an existing file, in which case the constructor is just passed the name of the file; the other should make a new,

empty tree, in which case the constructor is passed the name of the file, the block size, and the size (in bytes) of the keys and the size of the associated values. This will let you construct a new tree, load all the values into the tree (stored in the file) from `host-list.txt`, and exit the program, and then restart using the existing file. You will need to modify the interface to allow these two different ways of starting the DNS database program.

Stages and marking

Stage 1. (Up to 75%) Complete the in-memory B+ tree database program, including the `find`, `put`, and traversal methods in the B+ tree classes.
Stage 2. (Up to 90%) Extend your program to store the trees in files rather than as an in-memory data structure. You will need to modify the GUI interface to allow you to either create new files , or to load previously constructed files.
Stage 3.
Stage 3. (Up to 100%) One (or more) of the following:

Add a deletion function that allows the user to delete values from the database. It should be possible to specify either the IP address, or the domain-name, and the relevant record should be removed from both indexes.

Real DNS data allows an IP address to have more than one associated domain name (though not vice versa). Extend your program to allow multiple domain names to be associated with a single IP address.

Remove the redundancy of storing the entire database twice by making the leaves of the trees store the key and an index of a "data block" that holds the IP address - domain name pair, rather than the key and the associated value. Also, store both indexes in the same file, along with the data blocks, so the database file contains six kinds of blocks - a meta information block, internal and leaf nodes for each index, and data blocks.

Files

We have provided the following files.

`GUI.java` Top level user interface to the program

`DNSDB.java` The DNS data base, containing two indexes

`BlockFile.java` The class for the random access file of blocks

`Bytes.java` A utility class for dealing with bytes.

`InvalidBlockFileException.java` An exception class thrown by the BlockFile

`BPlusTreeIntToString60.java` A stub for your first B+ tree index

`BPlusTreeString60toInt.java` A stub for your other B+ tree index

`host-list.txt` A tab separated file of 115,489 entries; one IP : domain-name pair on each line. (This is much too small for a a full DNS, but it is already 5MB, and will be adequate for demonstrating your prototype).

For testing, you may want to make a smaller file, and also make the block size smaller than 1024 - with many keys per block, it will take a lot of data in order to make the tree large enough to test your program properly.

What to hand in: code and report.

You should submit two executable Jar files, one for part 1 and one for part 2, along with the associated source files.

You must also submit a report on the program to help the marker understand your code. The report should

describe what your code does and doesn't do.

describe any bugs that you have not been able to resolve.

report briefly on the performance of your program - how many block reads and writes were required on average to find a value or to add a new record.

outline how you tested that your program worked.

describe any of the extensions you did and discuss the results.

The report should be clear, but it does not have to be fancy - very plain formatting is all that is needed (even a plain text file would be OK). It does not need to be long, but you need to help the marker see what you did - without that help, the marker may miss things and not give you the marks you deserve!

The submission link can be found on the left or on the Assignments page.