Database Systems

Assignment 2

Daniel Atanasovski

S3658474

Contents

[Task 1 – Heap File 3](#_Toc100046331)

[Pre-processing 3](#_Toc100046332)

[Java Running 3](#_Toc100046333)

[Java Implementation 3](#_Toc100046334)

[Results 4](#_Toc100046335)

[Task 2 – Range Query 5](#_Toc100046336)

[Java Implementation 5](#_Toc100046337)

[Results 5](#_Toc100046338)

# Task 1 – B+ Tree

NOTE: The project depends on the dbload program created for assignment 1 as well as the pre-processing python file. These are all provided in the included submission.

Instructions to pre-process and generate a heap file are provided below.

## Pre-processing

The pre-processing script is implemented in python and depends on the package ‘Pandas’. It can be installed with the following command:

*Sudo pip install pandas*

The script simply removes un-needed columns within the source data file, artist.csv, and outputs it into the destination file ‘artist\_processed.csv’. The naming of the source data file is expected to be ‘artist.csv’ by the script and requires manual altercations to the python script to change the expected filename. The script can be run with the following command:

*Python preprocess.py*

## Generate heap file

After unzipping the source zip file, the java code can be compiled with the following command (assuming all java source files are in the src/ directory):

*Javac src/\*.java*

After compiling the code, the program expects the arguments ‘-p’ followed by the heap file size in bytes and then the source csv file that will supply the data to the program. An example command is as follows:

*Java -cp src dbload -p 4096 artist\_processed.csv*

## Java Implementation

The B+ tree implementation is like how is described in most descriptions online, however taking a ‘long’ data type for the key to represent the date, and a ‘long’ data type to represent the pointer or ‘record id’ in the heap file.

On the creation of a B+ tree, an order is defined for each node to limit themselves to, which is determined by the page size given to the program and the predefined size of a node in bytes. After determining the node size, a root node is created and awaits insertion for balancing and if necessary, splitting the node.

A node within the program is utilized by the same object that simply has a flag of whether it is a leaf node, as well as data structures pointing to either its children node values or its record id values.

Duplicate values within the program are not handled, and as such are simply ignored by the program, only taking unique keys as valid insertions.

Splitting a node once it is full simply breaks the current node into two and adds both nodes to the parent node. This would consequently create a chain reaction up the tree if they were all full.

## Improvements

Currently, the tree is saved using Java’s Serializable interface, which has added a lot of unnecessary data to the index file, and a more accurate way would be to save each node manually through byte arrays.

Duplicate keys are ignored, however, can be implemented by utilizing the connection between each leaf node, and iterating over that connection to find all the duplicate results.

## Results

Creating the tree can be slow as it constantly keeps the insertions sorted and the overall tree balanced, and with the cleaned data, can average ~8 seconds to create.

Text

Description automatically generated

To Validate the data, we will take the first result of the processed data and convert some data types from the Heap File for comparison.

Using a program like HxD we can view the byte data of the Heap File. The first 8 bytes are the values surrounding the birthdate, followed by 8 bytes for the death date, then the integer value for the wikipageID with 4 bytes before the string values. The first 8 bytes come to a value of 1371 which corresponds to the days since the epoch and is accurate to the value of the first record’s birthdate, 3/10/1973.

# Task 2 – Range Query

## Java Implementation

The range query is simple and begins by validating the input from the arguments, including whether the dates are valid and whether the Heap File is a valid file. After parsing the page size from the Heap File argument, we can determine the bytes to be read on each input of the Heap File.

The date arguments are parsed to the LocalDate datatype as it provides a clean API to convert from the days since the epoch (1970-1-1) to the LocalDate object, which is important to perform range query operations on the data.

On reading a page size worth of data from the Heap File, the data is split according to the fixed constant of record size, which is known from task 1. The record size of data is converted to an object that performs validation on the input and converts to the correct data types. After the conversion to a record object, the date is compared to the two-argument dates, in which case if it exists between the two provided dates, the name field, and birth date are printed out to the output, otherwise, it is ignored.

This is repeated until all pages are read and each record on the page is compared to the input dates. It finally outputs the number of matching records for the query.

## Results

An example query/command to run the program is provided:

*Java -cp src birthdate heap.4458 19700101 19701230*

This query should provide a lengthy output of 931 records individually printed to the console with the name and birthdate field, overall taking 8.26 seconds to complete the query.