CSC443 Assignment 1 – Report

1. Cost Analysis

Let

B – The number of data pages
R – Number of records per page
D – (Avg.) time to read or write disk page
F – The min # of items in each b-tree internal leaf

Then, analysis the following *cost of operations* table.

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	$D\log_2 B$	$D(\log_2 B$ + # pgs with match recs)	Search + BD	Search +BD
(3) Clustered	1.5BD	$D\log_F 1.5B$	$D(\log_F 1.5B$ + # pgs w. match recs)	Search + D	Search +D
(4) Unclust. Tree index	BD(R+0.15)	$D(1 + \log_F 0.15B)$	$D(\log_F 0.15B$ + # pgs w. match recs)	Search + 2D	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D	BD	Search + 2D	Search + 2D

Heap: ***Scan*** - need read all record, therefore we need to go over all pages. So, the time is *Number of pages * time to read each page* = BD.

Equality - the match record is uniformly disturbed across all pages, so the expedite time is ½ * (time to go over all pages) = ½ * BD.

Range - need to go over all record to find out which record is in the range. So, this will like scan, need to go over all pages = BD.

Insert - need to read the target page in memory first and then find out where is the, then modify the page, finally write it back to disk. Thus, the time will be 2 Disk OPS time = 2 * D.

Delete - need to search where the record is first and then write a delete mark on the record. Thus, the time will be *Search + 1 Disk OP* = Search + D.

Sorted: ***Scan*** - need to read all record, therefore we need to go over all pages, so time = BD.

Equality - Since pages are in sorted order, the expect number of pages to read in order to find the match page with *binary search* is $log_2 B$. Thus, total time will be D * $log_2 B$.

Range - First need binary search finds where is the match records starts, then do sequence read until reach the last match record. So, now we have $log_2 B$ + # of page with records in range. Time will be $D * (log_2 B + \#\ of\ pages\ with\ match\ records)$.

Insert - need to perform search first to find out where to insert. Then, after insert. Shift the record after insert point back by one which will take *BD* time. Thus, the time will be *Search + BD*.

Delete - Similar to Insert, need to search where is the target first, then perform a delete, then shift all the records after the delete point forward to maintain the sorted order. Thus, the time will be *Search + BD*.

Clustered:

Scan – due to the tree structure of clustered Indexes which has 67% occupancy rate, the file size will be 1.5 * data size. Therefore, the total number of pages that scan need to go over is 1.5 * B. So, the time for scan is 1.5 * BD.

Equality – For a clustered tree in order to find a record we need to move from the root all the way down to a leaf node, so the greatest number of levels we have to go through is the number of levels of the tree = $log_F(1.5 * B)$. So, the time will be $log_F(1.5 * B) * D$.

Range – First we need to find out the first match record which takes $log_F(1.5 * B)$ pages read. Then, we need to read until the last match record. Thus, the total pages get read is $log_F(1.5 * B) + \#\ of\ pgs\ with\ match\ records$. So, the total time is $(log_F(1.5 * B) + \#\ of\ pgs\ with\ match\ records) * D$.

Insert – First, we find the place to insert, since in avg. case, due to the 67% occupancy rate there will be some free spot for us to insert, so no need to touch the index or shifting elements. Thus, the expected time is search + D.

Delete – Same to ***Insert*** OP.

Unclustered Tree Index:

Scan – The time for scan can be split into two parts, first is the index reading time, the second is the records reading time. Since there is a 67% occupancy rate and index files have the size of 10% the data size. Therefore, the index file has size of 1.5 * 0.1 * B. So, the time for reading index is 0.15BD. Secondly, for the record reading, due to the data is unclustered each record read is expected to have one disk OP on avg., thus, the time for reading records will be $Numbers\ of\ records * Each\ Disk\ OP\ time = BDR$. So, we can get our total time = $BD * (R + 0.15)$.

Equality – For the previous part we can know, the size of index file is 0.15B. So the number of pg. we are expected time to read from root to the target will takes is $log_F(0.15 * B)$ * D. After finding out where is the record, we need to do one more read to read the data page. So, the total time is $D * (1 + log_F 0.15B)$.

Range – First we need to find out the first match record which takes $log_F(0.15 * B)$ pages read. Then, we need to read until the last match record. Thus, the total pages get read is $log_F(0.15 * B) + \#\ of\ pgs\ with\ match\ records$. So, the total time is $(log_F(0.15 * B) + \#\ of\ pgs\ with\ match\ records) * D$.

Insert – First we need to search where the record located, then read the data page out, insert the record, write the page back. Thus, the cost is $Search + 2D$.

Delete – Same as the ***Insert*** operations.

Unclustered Hash Index:

Scan – Due to the 80% occupancy rate, the File size is 1.25 time larger then data size. And the index is 10% size of the data File size. Thus, the index file size is 0.125B. Due to unclustered organization, reading each record need one disk OP. Total time = index reading time + data reading time = 0.125BDR + BDR = BD(R+0.125).

Equality – Need one Disk OP to read out the index file page located at the position given by the hash function. Then, one more read to data file.

Range – For neither index file is sorted not data file is sorted. **Unclustered Hash Index** do no help to range search. So, the range search will need to read all the data to find match record. Thus, the cost is BD.

Insert – First we need to search where the record located, then read the data page out, insert the record, write the page back. Thus, the cost is $Search + 2D$.

Delete – Same as the ***Insert*** operations.

2. Verification Using Real Data Files

In part 2, we will verify part of the estimation from part 1 by performing query on real database. In order to do this, first, we need to build a database, then, we wrote a database reader with timing function to measure time used to perform same query on different database.

Prepare Data

We use the sample data from eforexcel, which contain 500,000 fake employee info, to build our data base. (http://eforexcel.com/wp/wp- content/uploads/2017/07/500000-Records.zip)

In order to simplify our database reading, we are padding each col to its max char count with space by using a python program. (csv_pre_process.py).

Because there are duplicate Emp_ID's in the original csv file and we want to use Emp_ID as primary key later, so we need to remove the redundant Emp_ID.

1. Import the csv file into a sqlite database as table "Employee".
2. Use SQL "*delete from Employee where Employee.rowid not in (select MAX(Employee.rowid) from Employee group by Emp_ID)*" to remove the record with duplicate Emp_ID.
3. Export database to sql file for future use.

At last, we need to create a table schema for database. (*See table.sql*)

Create Database File

Now, we will import post-processed data into database files with 4 different configuration.

No index with 4k page

```
>sqlite3 no_idx_4k.db
> Pragma page_size = 4096;
>.read Employee.sql
```

No index with 16k page

```
>sqlite3 no_idx_16k.db
> Pragma page_size = 16384;
>.read Employee.sql
```

Un-Clustered index with 4k page

>sqlite3 no_idx_16k.db
> Pragma page_size = 16384;
>.read Employee.sql
>Create index Emp_index on Employee(EmpID);

Clustered index with 4k page

For clustered index, we first create an Employee table with "***Without ROWID***" at the end of create table SQL. And we also set Emp_ID as primary key.

>sqlite3 no_idx_16k.db
> Pragma page_size = 16384;
> Create table (…) without ROWID;
>.read Employee.sql

Database reader - Analysis App

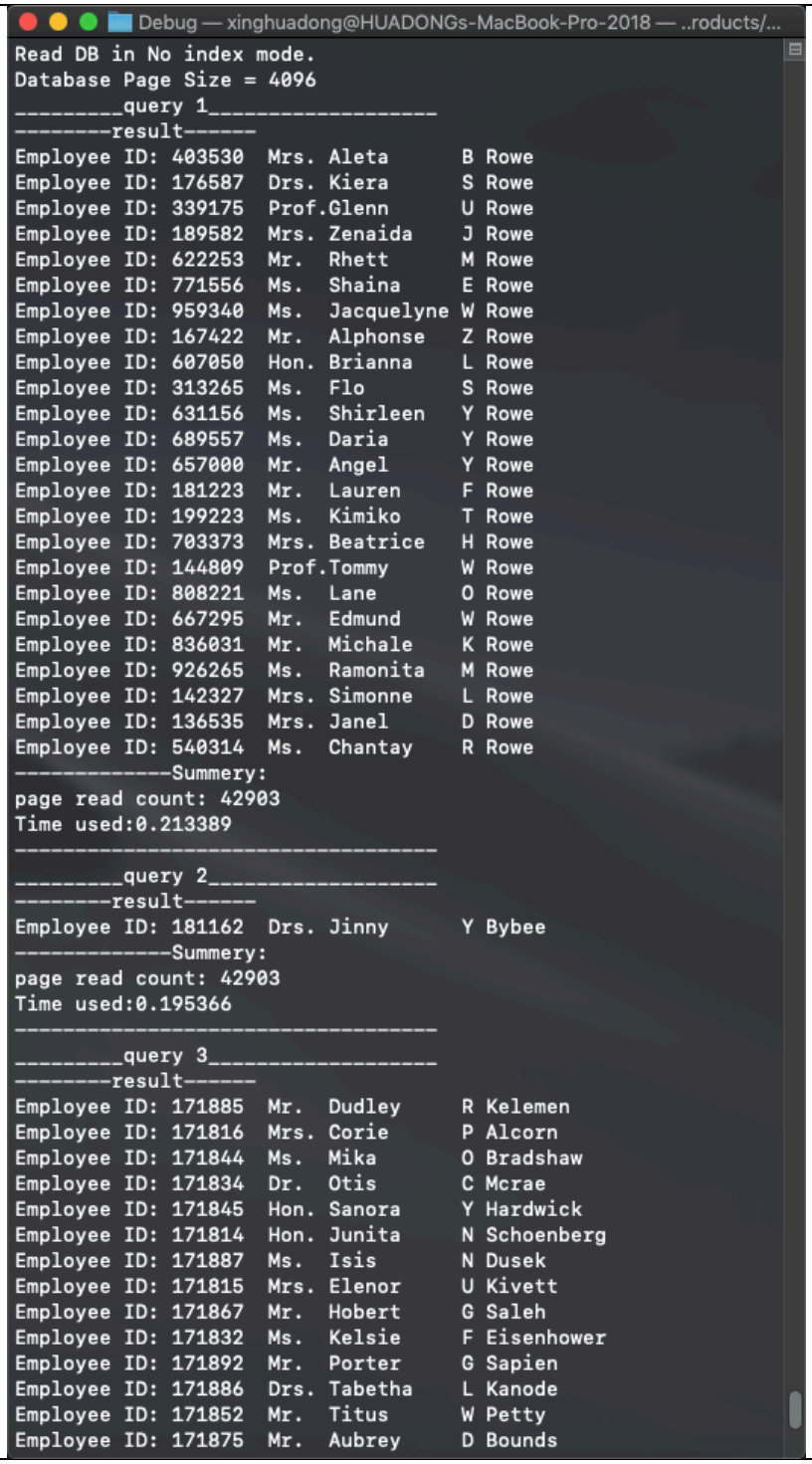
This app contains three pre-define queries,

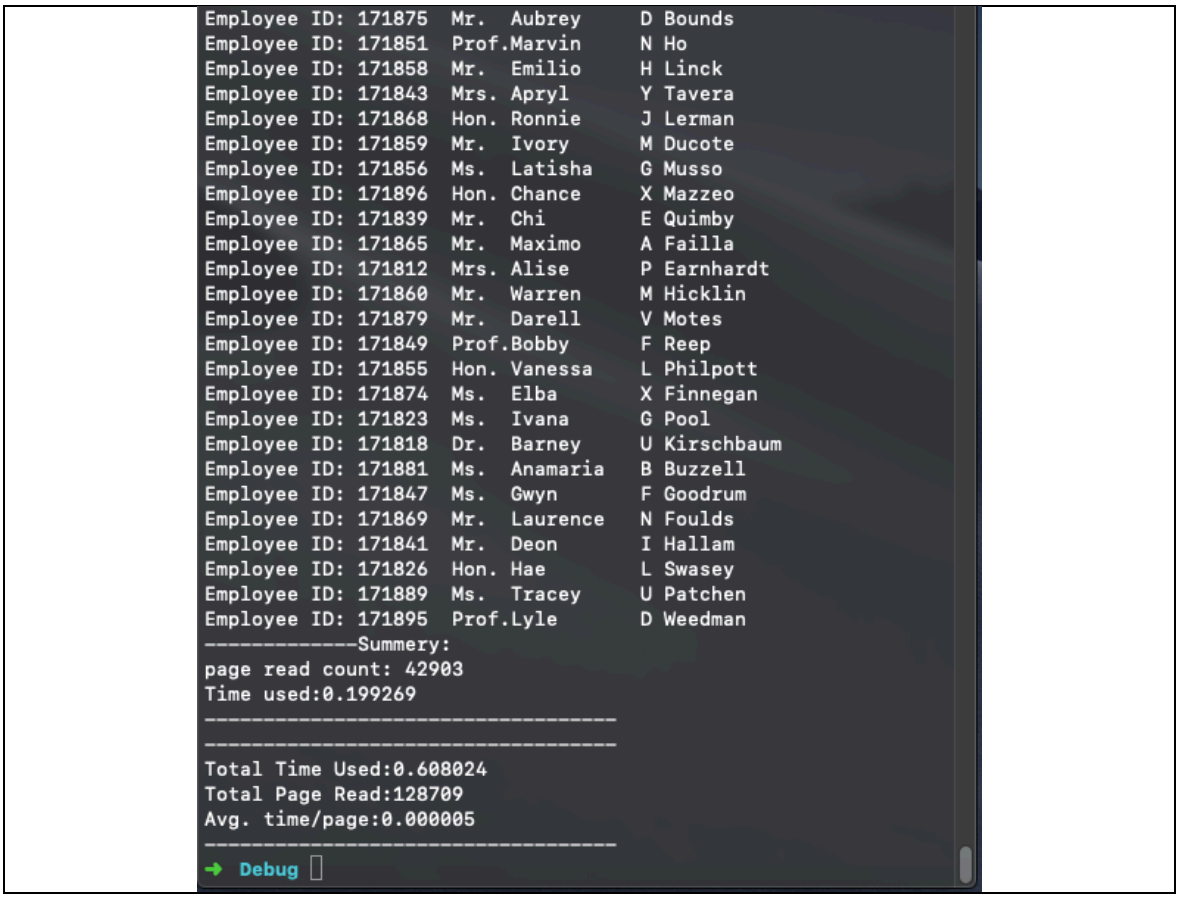
a) Find the employee ID and full name of anybody whose last name is "Rowe".
b) Find the full name of employee #181162.
c) Find the employee ID and full name of all employees with employee ID between #171800 and #171899.

and It can give the count of page read and avg. time/page of those four databases we created in the last step.

Result and Analyses

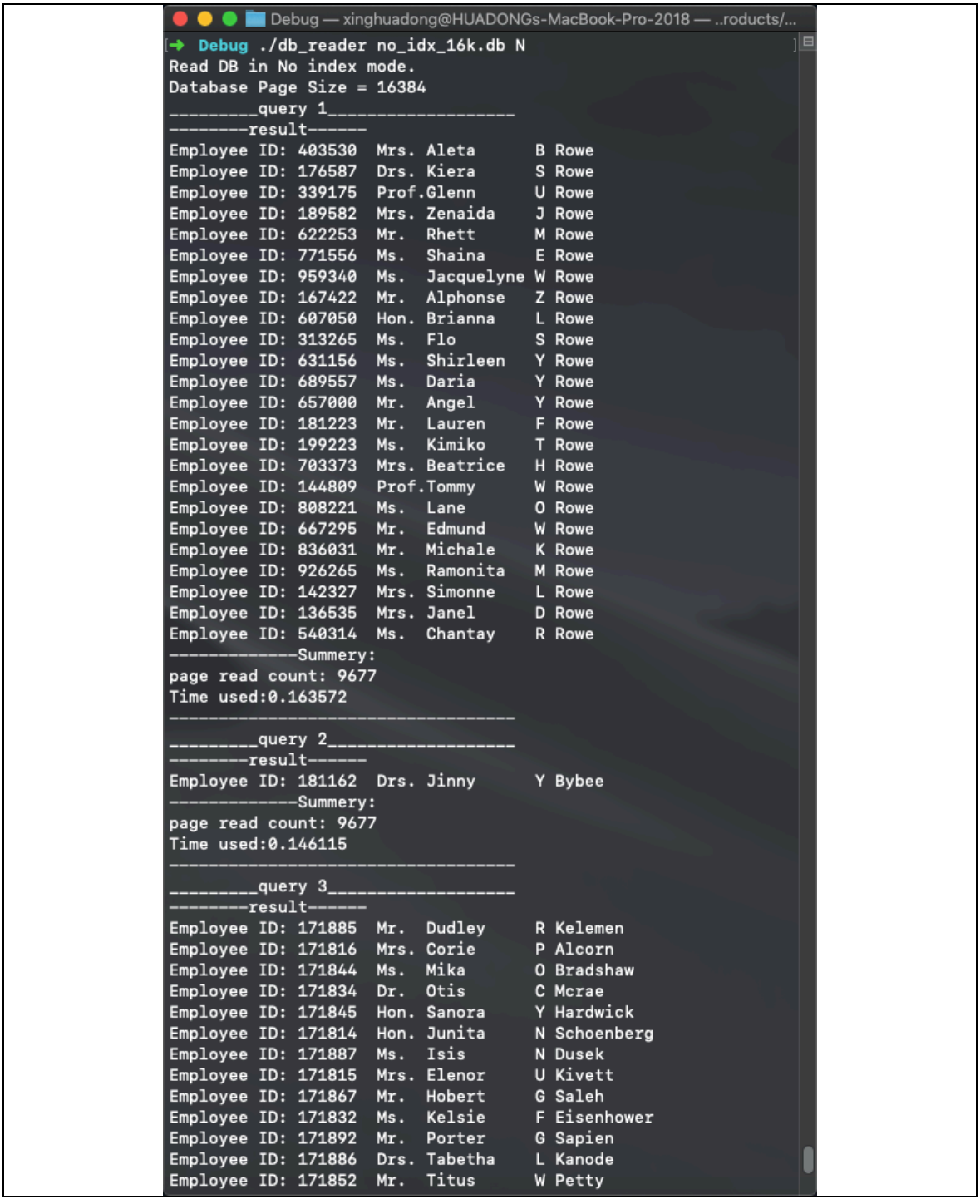
No index with 4k page

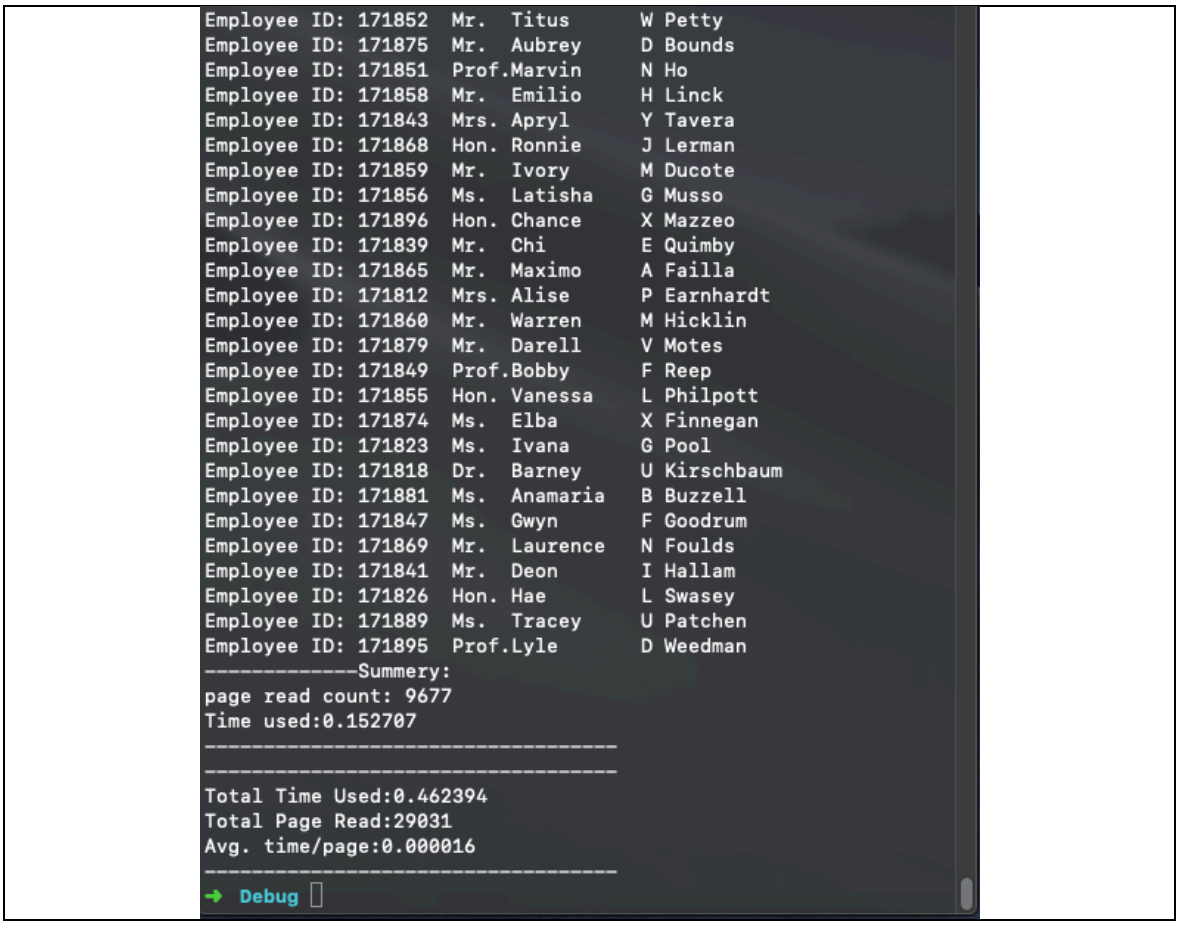




For the no index database, all 3 queries need to perform a full scan to the database, this match up our estimation in part 1, which scan for no index database required DB time.

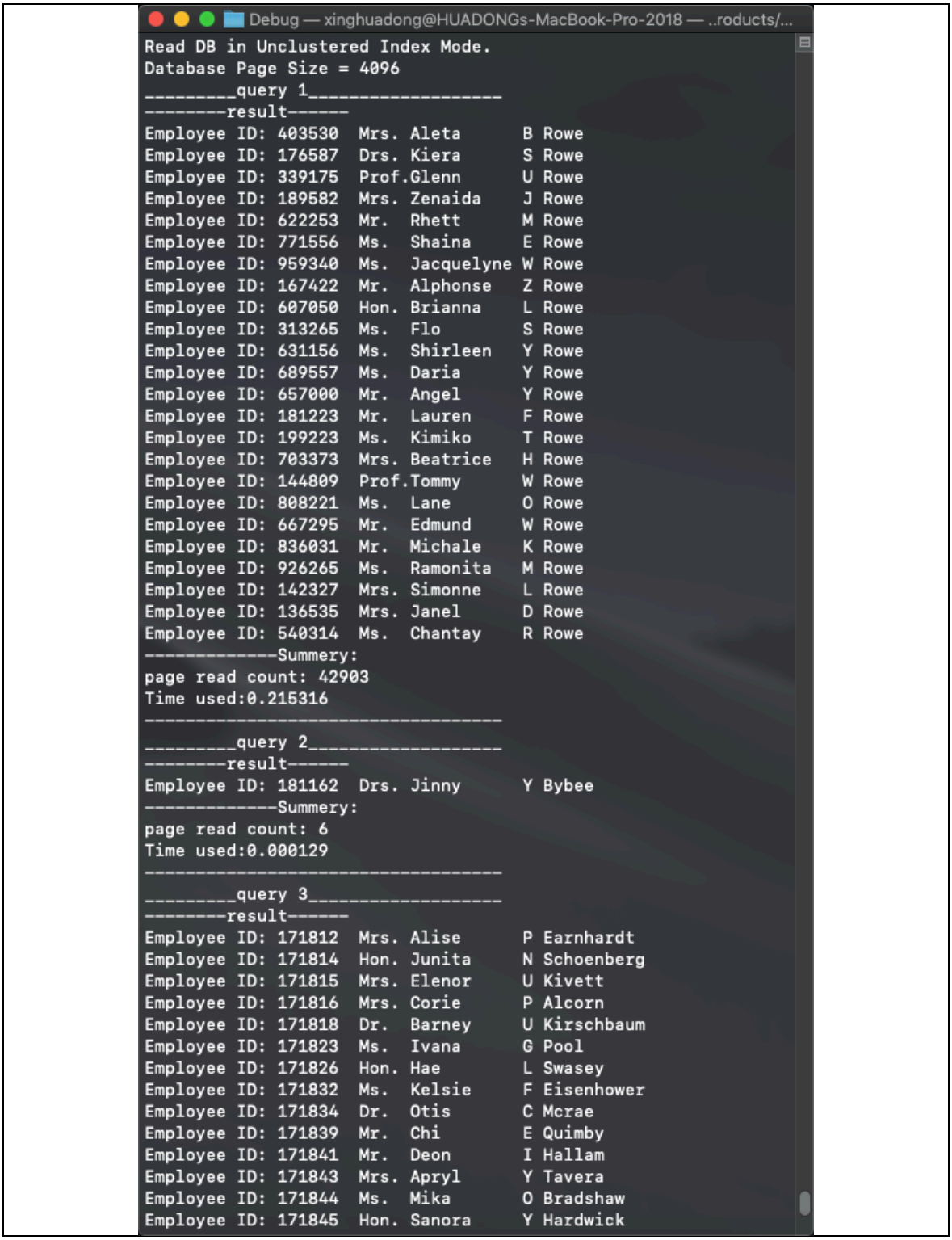
No index with 16k page

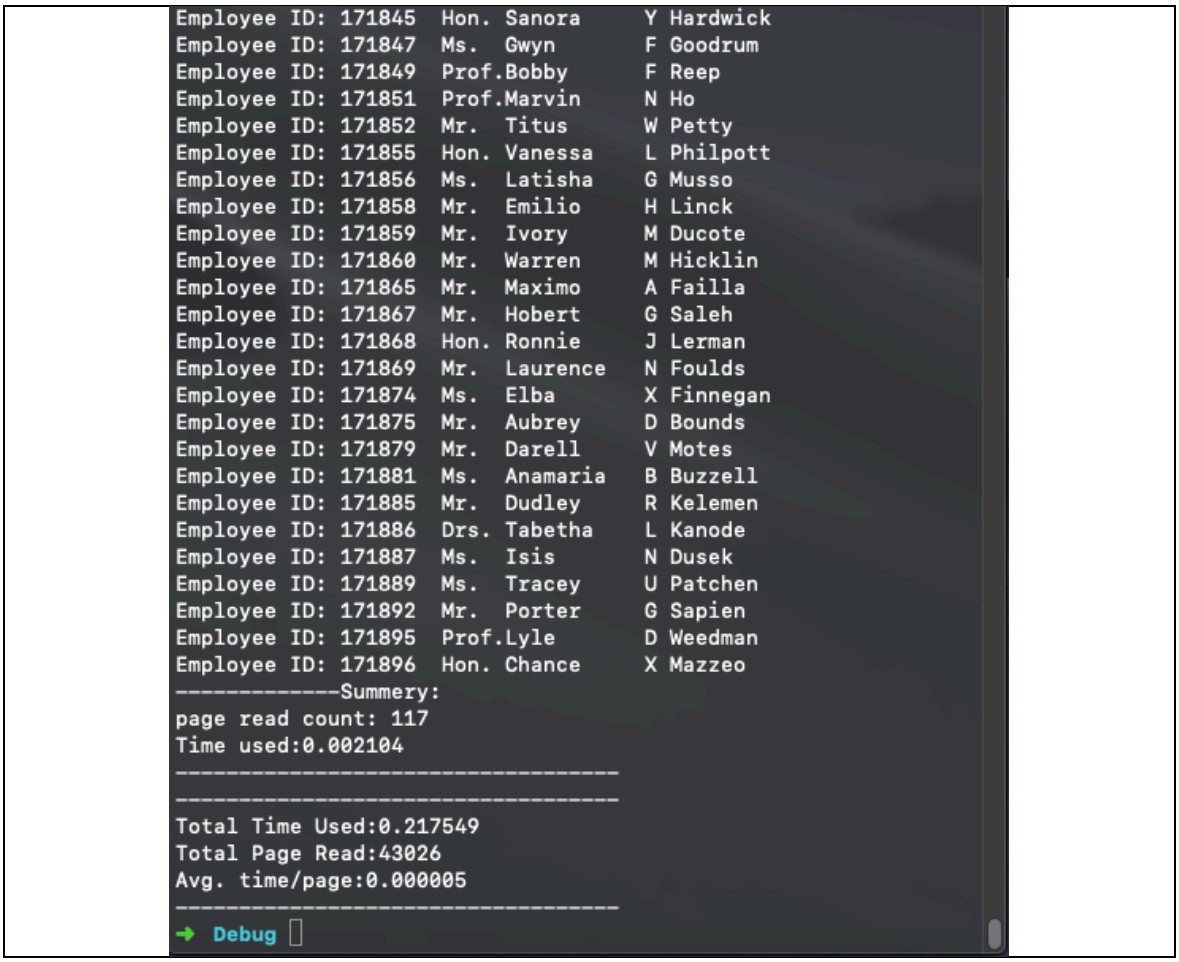




For the no index database with 16kb page size, compare to the 4k page size database, we have 4 time less pages, but 3 time more avg page reading time. Overall, performance of 16k is slight better than 4k.

Un-clustered index with 4k page

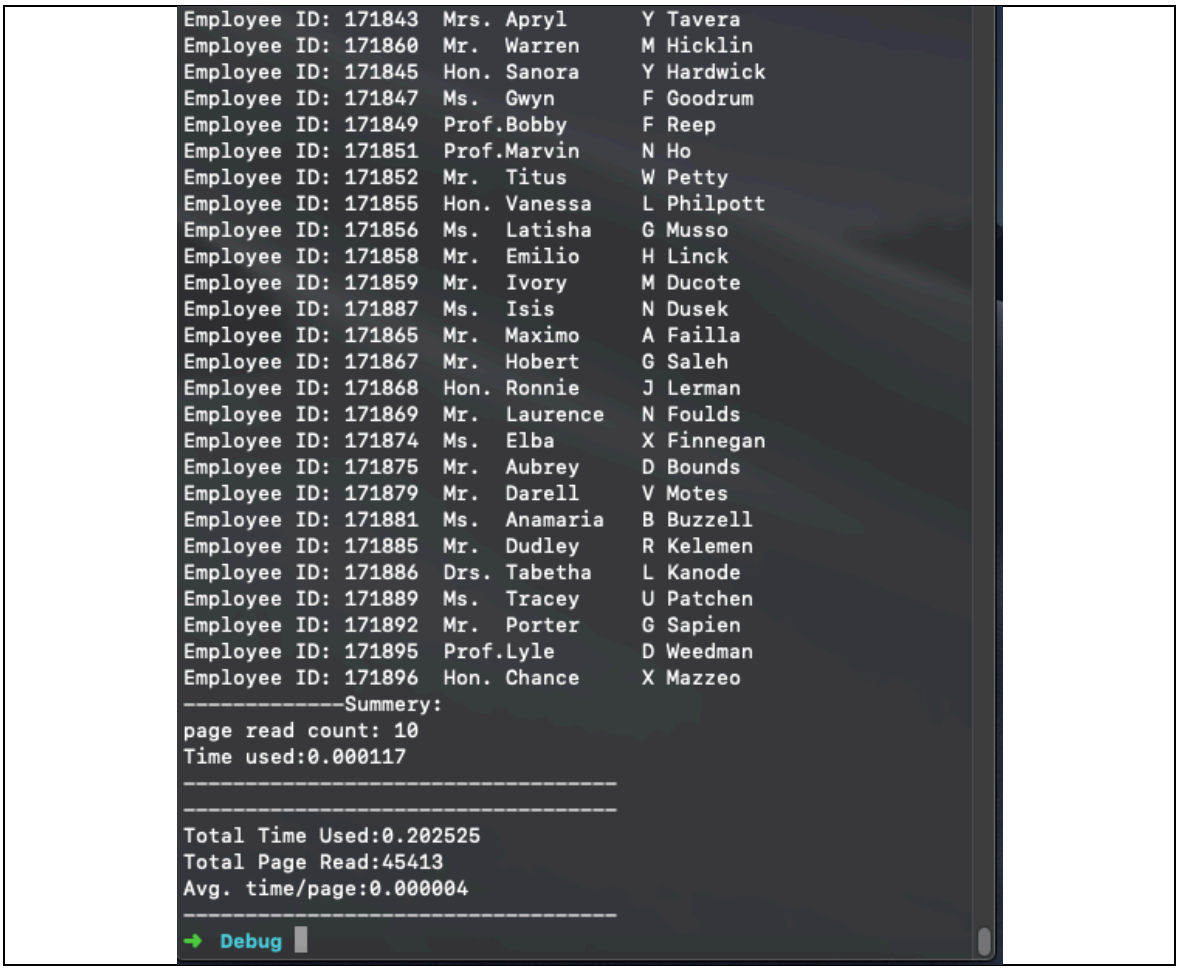




As, we can see query 1 has very bad performance compare to the other two. But it also make sense that our index is only on Emp_ID, but not on Last_Name. So, to perform query 1, we still need to scan the database, and it's takes overall the same time as the database without index for query 1.

However, for query 2 and 3, we can rely on index to speed up the search, we can see the time it takes for query 2 and 3 are significant reduced. Query 2 only read 6 pages to reach the result while Query 3 only need 117 pages read.

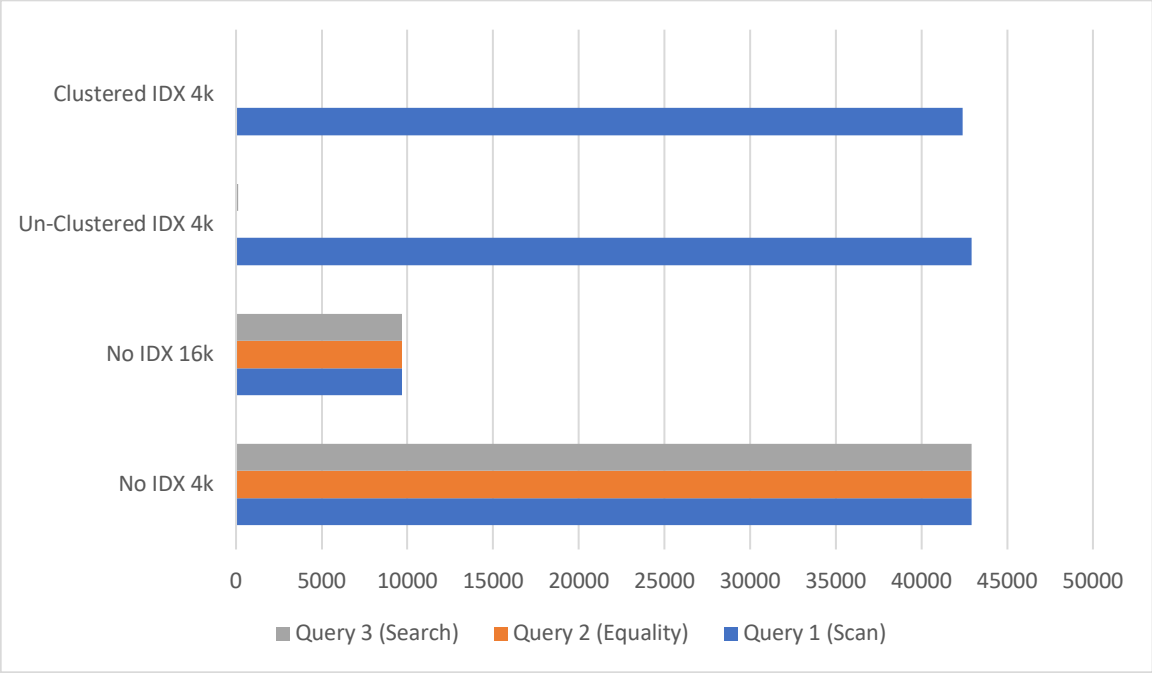
Clustered index with 4k page



For the clustered database, records of employee whose Emp_ID are closer are store together. This is very friendly to range search. For query 3, we only read 10 pages compare to 117 pages read on unclustered database.

There is also an interesting thing, for query 2, unclustered and clustered database both use 6 pages read. By analysis the database manually, we found clustered database has deeper depth than unclustered database. So clustered database need to read page to go deeper while unclustered database need to read page to jump between index and table.

Conclusion



Column1	Query 1 (Scan)	Query 2 (Equality)	Query 3 (Search)
No IDX 4k	42903	42903	42903
No IDX 16k	9677	9677	9677
Un-Clustered IDX 4k	42903	6	117
Clustered IDX 4k	42397	6	10

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	D(log 2 B + # pgs with match recs)	Search + BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	D(log F 1.5B + # pgs w. match recs)	Search + D	Search +D
(4) Unclust. Tree index	D(R+0.15)	D(1 + log F 0.15B)	D(log F 0.15B + # pgs w. match recs)	Search + 2D	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D	BD	Search + 2D	Search + 2D

By comparing the char, we can see for no index database, all three queries are O(DB).
For Clustered index, we don's have save space in sqlite, so our scan still in O(DB). For equality search, our branching F is overall around 10, however, we have good luck to find match record so fast. So, the time for equality search and range search are in the same magnitudes.
For the un-clustered index, it's similar to clustered, the only difference is it has to read the index file first.