CMPUT291 B1 Group Project 2 Report

Submitted by: Chuan Yang(1421992), Mengyang Chen(1412408), Ruilin Fu(1447466)

Introduction

The program is written in Python 3, and it uses randomized data set to measure the time cost for searching entries in Berkeley DB.

Running this program requires Oracle Instant Client SDK and Python3 library bsddb3. To start the program, user can locally execute

$python3 project2.py <DB_option>

Where DB_option can be one of b-tree/hash/indexfile

b-tree – The program would generate Berkeley DB file structured as a b-tree
hash – The program would generate Berkeley DB file structured as a hash table
indexfile – The program would generate Berkeley DB file as b-tree, plus the corresponding indexfile

Once mydbtest is permitted to execute, user can also run the following command locally

$mydbtest <DB_option>

Implementation

The program is written in Python 3 with external library bsddb3. Test data is generated by Python method random.seed(), which would create exact 100,000 tuple of (key, value) to the database file. Users are able to change database size by setting the environment variable.

Key search uses db_type.get() method from bsddb3. Data search for hash table and b-tree would traverse the entire database file to search specified entry, and would look for corresponding index file when it is available. Ranged search for hash table is same as a hash table traversal, given the fact that BDB stores hash type unordered. For b-tree and indexfile, since both are in increasing order, ranged search would first find the lowest possible entry, and then point the cursor to the next entry, until the upper bound is reached.

We created our index file as a (value, key) tuple b-tree, which would reduce the time cost on data traversal and matching data.

This program is written in a way to minimize extra time cost and complication during the test (from Python and the OS). It should be clear that this program is not optimized for the overall performance.

Performance Analysis

The following data was obtained from an VMWare image of CentOS 7, with Python 3.4_x64. The virtual machine exclusively uses 2 physical cores from Intel Core i5-3230M, and 4GB RAM.

TIME COST FOR KEY SEARCH (µs)					
KEY	1	2	3	4	AVERAGE
__b-tree	228.881	216.961	201.702	234.130	220.419
__hash	325.203	226.021	319.481	381.231	312.984
__index	223.398	229.836	205.278	205.994	216.127
found	1	1	1	1	

TIME COST FOR DATA SEARCH (μs)					
DATA	1	2	3	4	AVERAGE
__b-tree	440546.036	433022.975	421671.390	425150.632	430097.758
__hash	505213.737	504491.568	495060.682	490895.748	498915.434
__index	207.424	209.332	129.461	137.329	170.887
found	1	1	1	1	
TIME COST FOR RANGE (μs)					
RANGE	1	2	3	4	AVERAGE
__b-tree	49134.493	140391.350	27397.394	526836.395	185939.908
__hash	212511.539	225524.187	213924.170	218875.408	217708.826
__index	41790.428	153458.834	28549.671	522838.831	186659.441
found	7834	24774	4062	96222	

Key Search: The time cost for key search is nearly the same between 3 options. The small advantage for b-tree (also index file) is from the reduced time complexity (between $O(\log_i n)$ and $O(n)$) in the worst case to reach arbitrary key (which hash table has $O(n)$).

Data Search: It is clear that the data search time cost for b-tree and hash table are 2000 times higher than the result of index file. Within b-tree and hash groups, the time cost to obtain arbitrary data is the same as data traversal – $O(n)$. The use of index file would allow the search range be reduced to $O(\log_i n)$.

Ranged Search: The ranged search for b-tree and index file is the same, given the fact that they both share b-tree structure for key->data relationship. Since data is unordered, range search for hash table would cost the same as traversal. We noticed that for large range of data, b-tree and index file would perform worse than hash table (as shown in Range 4 of the range search form). Although the reason remains unclear, we believe that the extra cost was used to redirect between node cursors.

It should be clear that the data shown can only reflect some of the basic pattern, and the data scale is too small to exclude other interference from the Python language and the test environment. We would always encourage a more detailed analysis before reaching to any conclusion.

Following Ups

The project does have certain constraints and fixed marking scheme. In a real world situation, to further improve the performance of searching, the following measures would also improve the DB performance.

- Using concurrent read-only cursors, or split data set to multiple sub-systems
- Using a larger cache and better cache prediction model to reduce HDD IOs