Benchmarking Hash vs. B+ Tree Indexes

Eric Andrews, Christopher Chapman

CPS 377—Fall 2020

Goal

Goal

Empirically benchmark B+-Tree against Linear Hash for searches and range searches in terms of page I/O

Goal

Empirically benchmark B+-Tree against Linear Hash for searches and range searches in terms of page I/O

- Simulate page reads/writes

Goal

Empirically benchmark B+-Tree against Linear Hash for searches and range searches in terms of page I/O

- Simulate page reads/writes
- Implement Linear Hash

Goal

Empirically benchmark B+-Tree against Linear Hash for searches and range searches in terms of page I/O

- Simulate page reads/writes
- Implement Linear Hash
- Implement B+-tree

Goal

Empirically benchmark B+-Tree against Linear Hash for searches and range searches in terms of page I/O

- Simulate page reads/writes
- Implement Linear Hash
- Implement B+-tree
- Benchmark, compare to expected results

Implementation - Page Simulation

Implementation - Page Simulation

- Need to simulate and record page fetching

Implementation - Page Simulation

- Need to simulate and record page fetching
- Multiple types of pages

Implementation - Page Simulation

- Need to simulate and record page fetching
- Multiple types of pages
- To record: global counter struct, the `pageManager`

Implementation - Page Simulation

- Need to simulate and record page fetching
- Multiple types of pages
- To record: global counter struct, the `pageManager`
- Unify page types into `pageptr`, call `getPage` where appropriate

Implementation - Page Simulation

- Need to simulate and record page fetching
- Multiple types of pages
- To record: global counter struct, the `pageManager`
- Unify page types into `pageptr`, call `getPage` where appropriate
- Flexible, lightweight, easy, but potentially error-prone

Implementation - Page Simulation

- Need to simulate and record page fetching
- Multiple types of pages
- To record: global counter struct, the `pageManager`
- Unify page types into `pageptr`, call `getPage` where appropriate
- Flexible, lightweight, easy, but potentially error-prone
- Page size set at compile time by pre-processing definitions

Implementation - Linear Hash

Implementation - Linear Hash

- Implemented insert and search as described in textbook

Implementation - Linear Hash

- Implemented insert and search as described in textbook
- Created a range search function: simply searches hash table for every integer between supplied min and max keys

Implementation - B+ Tree

Implementation - B+ Tree

- Range search utilizes linked list structure in leaf nodes

Implementation - B+ Tree

- Range search utilizes linked list structure in leaf nodes
- Interlacing keys, pointers: `union kp`

Implementation - B+ Tree

- Range search utilizes linked list structure in leaf nodes
- Interlacing keys, pointers: `union kp`
- Alternatively, maintain two distinct arrays and do math

Implementation - B+ Tree

- Range search utilizes linked list structure in leaf nodes
- Interlacing keys, pointers: `union kp`
- Alternatively, maintain two distinct arrays and do math
- Pointers themselves need to be flexible—depending on level, either point to `treePage` or `ridPage`

Implementation - B+ Tree

- Range search utilizes linked list structure in leaf nodes
- Interlacing keys, pointers: `union kp`
- Alternatively, maintain two distinct arrays and do math
- Pointers themselves need to be flexible—depending on level, either point to `treePage` or `ridPage`
- Luckily, have the `pageptr` struct!

Expected Results

- Linear hash: constant time for single search (~ 1.2)

Expected Results

- Linear hash: constant time for single search (~ 1.2)
- Linear hash: range search depends on data; $1.2 \times (\max - \min)$

Expected Results

- Linear hash: constant time for single search (~ 1.2)
- Linear hash: range search depends on data; $1.2 \times (\max - \min)$
- Tree: $\log_{12}(n)$ time for a single search (assuming 512B page)

Expected Results

- Linear hash: constant time for single search (~ 1.2)
- Linear hash: range search depends on data; $1.2 \times (\text{max} - \text{min})$
- Tree: $\log_{12}(n)$ time for a single search (assuming 512B page)
- Tree: $\log_{12}(n) + \frac{m}{21.75}$ time to retrieve a range of length m from n records (assuming 512B page, 75% occupancy)

Page I/O Statistics

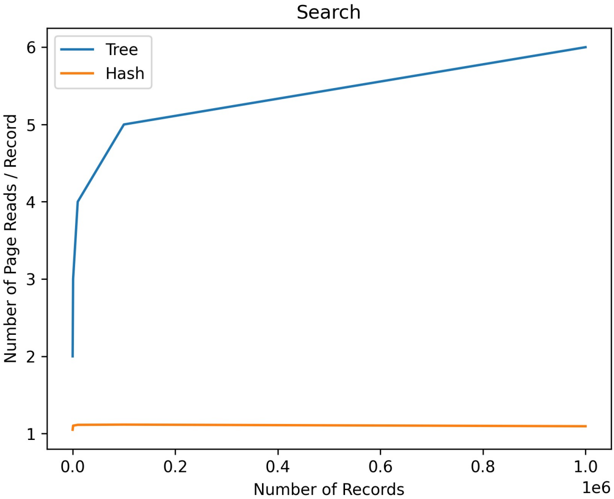
Operation	Reads	Writes	Est. Reads
Insert	3,223,374	2,857,734	-
Full Key Equality Search	1,095,543	0	1,200,000
Second Q Range Search	17,703,231	0	20,852,013
Full Range Search	102,532,199	0	120,245,644

Table: Hash Table with 1,000,000 Random Skewed Records

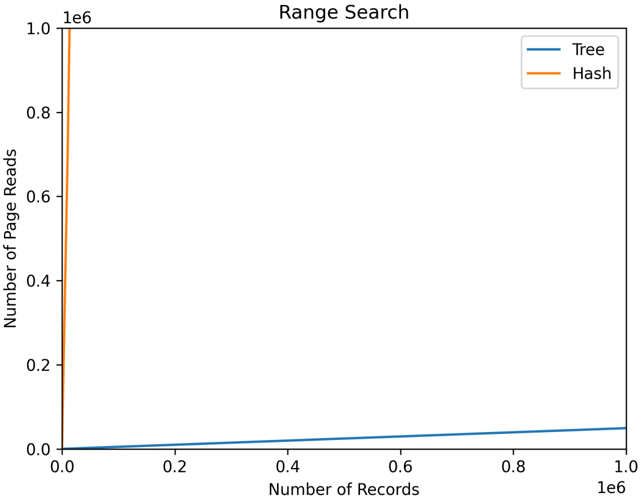
Operation	Reads	Writes	Est. Reads
Insert	4,361,765	1,250,899	-
Full Key Equality Search	6,000,000	0	6,000,000
Second Q Range Search	12,368	0	11,500
Full Range Search	49,310	0	45,983

Table: Tree with 1,000,000 Random Skewed Records

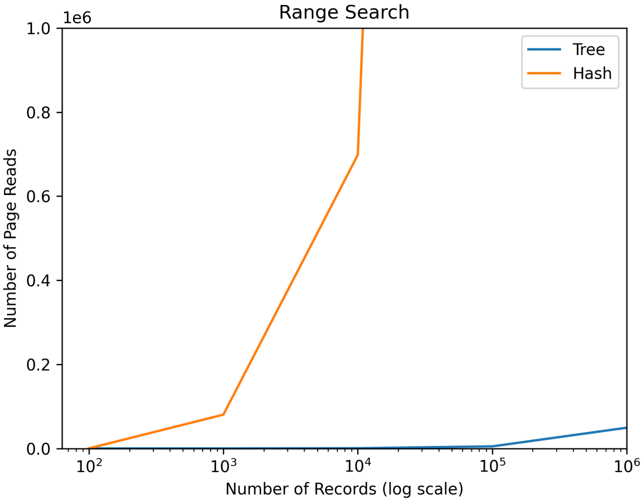
Search Performance



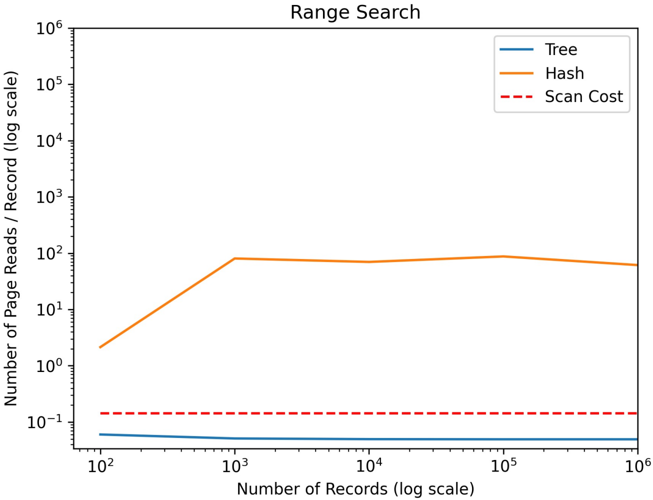
Range Search Performance



Range Search Performance – Log Scale



Range Search Performance – Log Scale


Range Search
Number of Page Reads / Record (log scale)
10^{6}
10^{5}
10^{4}
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
Number of Records (log scale)
Tree
Hash
Scan Cost