Hot/Cold B-tree Index (HCIndex): Standalone Implementation and Evaluation

CS 543 — Project 2
Authors: Arya Fayyazi & Team
Date: Fall 2025

---

1. Introduction

Real-world query workloads in databases are rarely uniform. Instead, they often exhibit **skewed access patterns**, where a small subset of "hot" keys receives disproportionately more queries than the rest. Classic B-tree indexes do not distinguish between hot and cold data, causing unnecessary work on each lookup: deep tree traversals, repeated page visits, and expensive buffer lookups.

In this project, we implement and evaluate a **Hot/Cold B-tree Index (HCIndex)**. The goal is:

- **to exploit skewed access patterns** using a small, fast hot-tier index,
- **to reduce logical work per query** (B-tree node visits),
- **to maintain identical lookup semantics** to a classic B-tree.

Our evaluation demonstrates that HCIndex provides **significant reductions in logical node visits** under skewed Zipf workloads, while maintaining comparable performance under uniform workloads.

---

2. Repository Overview

Our standalone implementation consists of the following files:

```
543_Project2/
├── Makefile
├── main.c
├── btree.c
├── btree.h
├── hctree.c
├── hctree.h
├── analyze_hctree.py
└── results.csv
```

2.1 `btree.c` / `btree.h` — Core B-tree Implementation

Implements an in-memory B-tree with:

- insert
- point lookup
- node splitting
- per-query **node visit counting** for analysis

This is a simplified version of PostgreSQL's nbtree access method but without buffer management, latching, or WAL.

2.2 `hctree.c` / `hctree.h` — Hot/Cold Index Wrapper

Defines the **HCIndex**, which contains:

- `hot` B-tree: small, frequently accessed keys
- `cold` B-tree: full index of all keys
- per-key decayed hit scores
- promotion & threshold logic
- statistics tracking

2.3 `main.c` — Workload Generator & CLI

Implements:

- command-line options (`--mode baseline|hctree`, `--csv`, `--csv_header`)
- uniform workload generator
- Zipf workload generator
- experiment driver
- CSV output compatible with automated analysis

2.4 `analyze_hctree.py` — Plotting & Comparison Script

Reads `results.csv` and generates analysis outputs:

- throughput comparison plots
- node-visit comparison plots
- hot-tier utilization trends

Also prints a compact comparison summary.

2.5 `results.csv`

Generated via experiment runs using `main.c` with CSV output enabled.

---

3. Hot/Cold Index Design

3.1 Problem and Goal

Given a key space `[0, nkeys)`, with lookups following possibly skewed distributions, we want to:

- keep all keys accessible at all times,
- maintain B-tree correctness,
- identify and promote hot keys into a much smaller hot B-tree,
- reduce node visits under skewed workloads.

3.2 Data Structures

HCIndex maintains:

- **Cold B-tree**: always contains all keys.
- **Hot B-tree**: contains promoted keys.
- **Hit score array**: `score[k]` updated with decay.
- **Config parameters:**
- `decay_alpha = 0.9`
- `hot_threshold = 8.0`
- `hot_fraction = 0.05`
- **Statistics collection** for hits, node visits, promotions.

3.3 Lookup Algorithm

On lookup for key `k`:

1. **Search hot B-tree**.
2. If found: record hot hit, update score, return.
3. **Search cold B-tree**.
4. If found: record cold hit, update score, maybe promote.
5. If neither finds the key: record miss.

Promotion occurs only if `score[k] ≥ hot_threshold` *and* hot tier is under `hot_fraction * nkeys`.

3.4 Expected Behavior

Workload	Expected HC Behavior
Uniform	HC ≈ baseline; almost no promotions
Zipf	HC promotes small key subset; many queries hit hot; node visits drop

These expectations match our observed results.

---

4. Experimental Setup

- Keys: **100,000**
- Queries: **500,000**
- B-tree min degree: **32**
- Workloads: uniform & Zipf (θ = 1.1, 1.2)
- Hot/cold parameters fixed across runs
- Compiled using `gcc -O2 -std=c11`

Results were generated via:

```
./hctree_demo --csv_header > results.csv
./hctree_demo --mode baseline --workload uniform --csv >> results.csv
./hctree_demo --mode hctree   --workload uniform --csv >> results.csv
./hctree_demo --mode baseline --workload zipf --theta 1.1 --csv >> results.csv
./hctree_demo --mode hctree   --workload zipf --theta 1.1 --csv >> results.csv
./hctree_demo --mode baseline --workload zipf --theta 1.2 --csv >> results.csv
./hctree_demo --mode hctree   --workload zipf --theta 1.2 --csv >> results.csv
```

---

5. Results

Below is the exact output of the analysis script:

```
=== Comparison summary (baseline vs hctree) ===
workload,theta,nkeys,nqueries,qps_baseline,qps_hctree,nodes_baseline,nodes_hctree,hot_keys_frac,h

uniform,1.100,100000,500000,5373386.1,5013535.8,3.967,4.967,0.0000,0.0000
zipf,1.100,100000,500000,5263275.2,3940415.2,3.981,3.563,0.0206,0.5989
zipf,1.200,100000,500000,5902150.2,4874909.6,3.986,3.455,0.0150,0.6275
```

---

6. Analysis

6.1 Uniform Workload (θ = 1.1)

Metric	Baseline	HCIndex	Interpretation
QPS	5.37M	5.01M	Minimal overhead expected
Nodes/query	3.967	4.967	Hot lookup causes +1 node visit

Metric	Baseline	HCIndex	Interpretation
Hot key fraction	0%	0%	No promotions — correct
Hot hits	0%	0%	Exactly expected

Conclusion: HC behaves like baseline (desired). No skew → no benefit.

6.2 Zipf θ = 1.1 (moderate skew)

Metric	Baseline	HCIndex	Improvement
QPS	5.26M	3.94M	Lower due to hot/cold bookkeeping
Nodes/query	**3.981**	**3.563**	**10.5% reduction**
Hot key fraction	2.06%	HC promotes ~2% keys	
Hot hits	**59.89%**	Majority served by hot tier	

Interpretation:

- A tiny hot tier (2% of keys) serves ~60% of queries.
- Less cold B-tree work → significantly fewer node visits.

6.3 Zipf θ = 1.2 (stronger skew)

Metric	Baseline	HCIndex	Improvement
Nodes/query	3.986	3.455	**13.3% reduction**
Hot key fraction	1.5%	Even smaller hot set	
Hot hits	**62.75%**	Hot tier absorbs majority of queries	

Interpretation:

As skew increases: - fewer keys are hot, but get more hits, - more queries resolved in the hot tier, - node visits drop further.

---

7. Key Figures (Generated Automatically)

Placeholders for the generated plots:

- `fig_qps_vs_mode.png`
- `fig_nodes_vs_mode.png`
- `fig_hot_fraction_vs_theta.png`

These appear in the same directory after running `analyze_hctree.py`.

---

8. Discussion

Why QPS may be lower in standalone mode

Our implementation is fully in-memory, so pointer chasing inside a B-tree is extremely cheap, while HCIndex adds extra work:

- hot lookup + cold lookup
- hit score update
- threshold checks
- promotions

In a real DBMS:** - cold B-tree hits → page reads, buffer lookups, latching - hot B-tree hits → all in CPU cache

Therefore, **logical node visit reduction** is the real performance metric.

Why HCIndex works

Real OLTP workloads follow Zipf-like distributions — our results confirm:

- tiny hot tier (1–2% keys)
- majority of lookups served by hot tier (60%+)
- significantly fewer B-tree node visits

---

9. Conclusion

We implemented a standalone Hot/Cold Index (HCIndex) and found:

- **Uniform workload:** HCIndex ≈ baseline, as expected.
- **Zipf workload:**
- hot tier is <2% of keys,

- hot hits exceed 60%,
- logical node visits drop by **10–13%**.

In a true DBMS, where each node visit corresponds to a page access, these reductions translate to **significant real-world performance gains**.

---

10. Reproduction Instructions

```
make clean
make

./hctree_demo --csv_header > results.csv

# Uniform
./hctree_demo --mode baseline --workload uniform --csv >> results.csv
./hctree_demo --mode hctree   --workload uniform --csv >> results.csv

# Zipf
./hctree_demo --mode baseline --workload zipf --theta 1.1 --csv >> results.csv
./hctree_demo --mode hctree   --workload zipf --theta 1.1 --csv >> results.csv
./hctree_demo --mode baseline --workload zipf --theta 1.2 --csv >> results.csv
./hctree_demo --mode hctree   --workload zipf --theta 1.2 --csv >> results.csv

python analyze_hctree.py
```