SC3020 DBMS Project 1

A comprehensive analysis of the database system implementation & design

Storage System

B+ Tree Index

Performance Analysis

Created by: Chong Qi En, Goh Jun Keat,
Cheah Jun An Douglas, Chin Linn Sheng, Nah Wei Jie

Table Of Contents

Project Introduction

Project Overview

This project focuses on designing and implementing two core DBMS components:

- Disk-supported storage system with 4KB page size
- B+ tree index for efficient data retrieval
- Uses NBA game data for evaluation

Project Context

The project simulates real DBMS behavior while keeping complexity limited to:

- **Single binary file** for data storage
- **Separate binary file** for indexing

Project Components

Storage System

Fixed 4096 byte blocks, compact 22-byte records

B+ Tree Index

Efficient range queries on FT_PCT_home field

Data Loader

Parses NBA data from text file into compact records

Query Engine

Range queries and efficient record deletion

NBA Game Data

26,651 games, 9 attributes per game

Record Storage

22-Byte Record Format

9 Attributes

4 bytes **FG_PCT_Home** • stored as float	4 bytes **FT_PCT_Home** • stored as float	4 bytes **FG3_PCT_Home** • stored as float
4 bytes **Team ID** • stored as 32-bit integer	2 bytes **Date** • 16-bit int no. of days from 2000-01-01	1 byte **PTS_Home** • stored as 8-bit integer
1 byte **AST_Home** • stored as 8-bit integer	1 byte **REB_Home** • stored as 8-bit integer	1 byte **Home Team Wins** • stored as 8-bit integer

Record Structure

- Fixed Size: **22 bytes** with no internal padding
- Each record is contained within 1 single block
- Fixed-length fields used for efficient storage
- No record header
- Each field is represented with the minimum number of bits
- Field sequence reordered for space optimisation
- Optimised for 4KB block size

Compact Storage Techniques

- Used pragma pack (push,1)

Block Storage

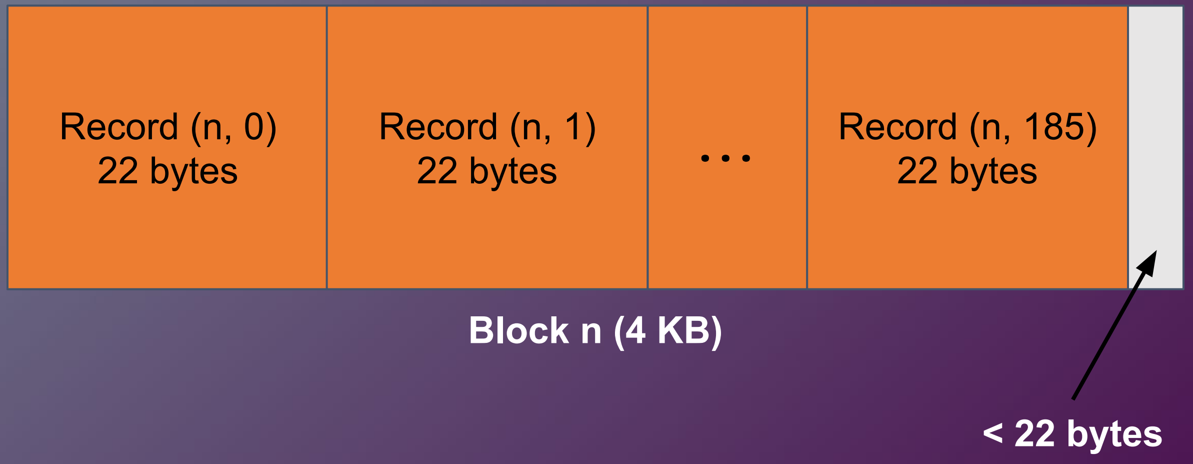
4KB Block Structure

Block Structure

- Fixed size: **4096 bytes** (4KB)
- Sequential packing of fixed-length records
- No page header, block header or slot directory
- Records may begin at any byte offset within a block
- Records are accessed via block ID and offset
- Blocks are stored contiguously on the disk
- Mimics modern file system block size

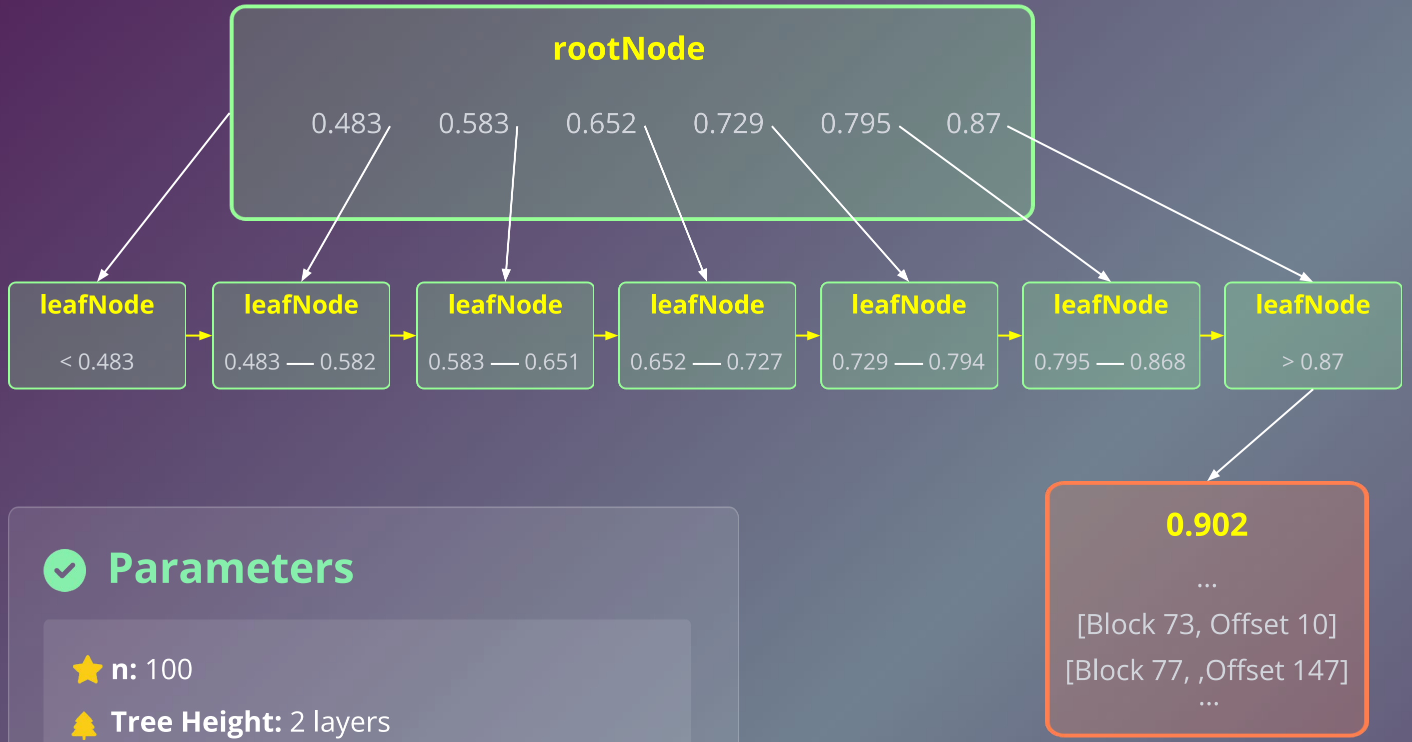
Block Information

Block size (byte)	**4096**
Record Size (byte)	**22**
Max Records per Block	**186**



B+ Tree Component Design

B+ Tree Structure



Parameters

- **n:** 100
- **Tree Height:** 2 layers
- **Total nodes:** 8

RecordRef Structure

- Contains block_id and record_offset
- Uniquely identifies physical location of records
- Multiple RecordRef objects grouped under same key in leaf nodes
- Supports duplicate key entries without duplicating the key itself

BPlusNode Structure

- Unified structure for both leaf and internal nodes
- Internal nodes use key arrays as separators
- Internal nodes maintain pointers to guide search
- Leaf nodes store key-record associations as RecordRef objects
- Leaf nodes linked sequentially through next_leaf pointers

B+ Tree Parameters & Optimization

Parameter Calculations

Block Size
4096 bytes

Node Overhead
27 bytes

Avg. Repeats
1.01466

Conservative Repeats
2

Optimal Parameters

- **Optimal n:** 254
- **Tree Height:** 2 layers
- **Leaf nodes:** 104

*Based on uniform distribution of keys

n-Value Comparison

Parameter n	Tree Height	Leaf Nodes
n=50	3	526
n=100	3	263
n=200	2	132

Key Considerations:

- Balance between tree height and node utilization
- n=100 provides good balance (3 layers, 263 leaf nodes)
- n=200 reduces tree height but increases branching factor

B+ Tree Operations

Insert Operation

- Inserts key-recordRef pairs while maintaining tree balance
- Follows insertion algorithm from course slides
- Handles splitting of nodes when overflow occurs

Range Query Efficiency

B+ trees are optimized for range-based searches due to their linked leaf nodes structure.

</> **searchGreaterThan(float key)**

Returns all records with keys greater than the search key

Insertion Algorithm

1. Search to find which leaf node to insert to
2. If leaf node is not full, insert it
3. Otherwise, split leaf node into 2 and distribute keys
4. Insert into parent and create new root if any
5. Repeat until a parent with no splits is required

Range Query Algorithm

1. Start from root node
2. Follow appropriate pointer based on search key
3. Continue until reaching leaf node
4. Scan leaf node and follow pointer to next leaf block
5. Continue scanning leaf nodes until range limit

Task 1 Results: Data Loading & Storage

Data Storage Statistics

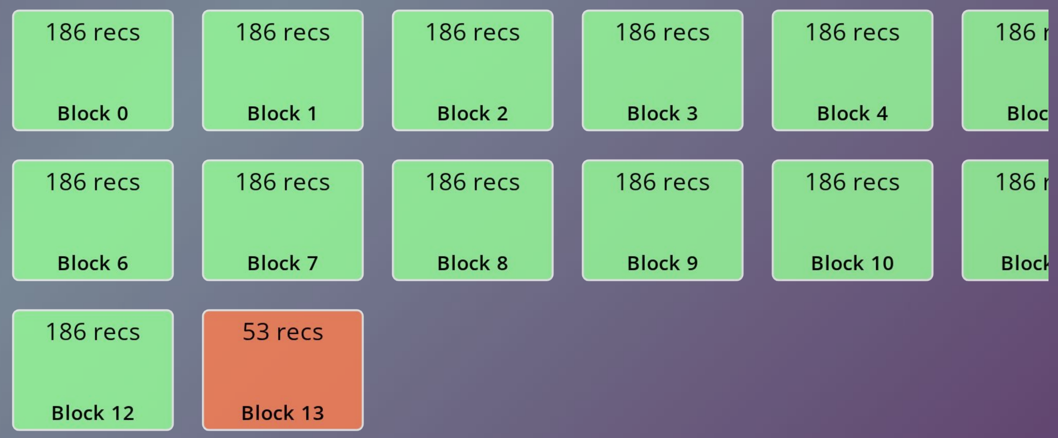
RECORD SIZE
22 bytes

TOTAL RECORDS
26,651

RECORDS PER BLOCK
186
max capacity

TOTAL BLOCKS
144

Block Structure Visualization



Full blocks (186 recs)
Last block (53 recs)

Block Calculation Formula

```
ttlBlks = (ttlRecs + MAX_RECORDS_PER_BLOCK - 1) /
          MAX_RECORDS_PER_BLOCK
```

Storage Efficiency

- Compact 22-byte record format minimizes space
- 186 records per 4KB block (optimal packing)
- Efficient use of space with minimal overhead

Task 2 Results: B+ Tree Index Construction

B+ Tree Statistics

Parameter n
100

Nodes
8

Levels
2

Construction Time
13 ms

Root Node Keys

0.483	0.583	0.652
0.729	0.795	0.87

Efficiency Highlights

- Compact 2-level structure with only 8 nodes
- Fast construction in just 13 milliseconds
- Optimal balance between depth and node capacity

Task 3 Results: Record Deletion & Index Update

Key Performance Metrics

Retrieval Time
71.295 ms
Running time of retrieval process

Records Deleted
1778 games
Records with FT_PCT_home > 0.9

Deletion Time
9442 ms
Total time for deleting 1,778 records

Index Node
2 nodes
Number of index nodes accessed

Task 3 results visualization

```
=== Task 3 Results ===
Number of index nodes accessed: 2
Number of data blocks accessed: 144
Number of games deleted: 1778
Average FT_PCT_home of deleted records: 0.939637
Retrieval time (index + heap): 71.295 ms
Running time of deletion process: 9442 ms
```

Deletion Summary

- **Average FT_PCT_home:** 0.939637
- **B+ Tree Updates:** 2 index nodes accessed

Task 3 Results: Statistics of the updated B+ tree & Brute Force comparison

Statistic of the B+ Tree Before and After

```
B+ tree loaded from disk: ft_pct_home.idx
=== B+ Tree Statistics ===
Parameter n: 100
Number of nodes: 8
Number of levels: 2
Root node keys: 0.483, 0.583, 0.652, 0.729, 0.795, 0.87
```

```
--- B+ Tree Statistics AFTER Deletion ---
=== B+ Tree Statistics ===
Parameter n: 100
Number of nodes: 7
Number of levels: 2
Root node keys: 0.483, 0.583, 0.652, 0.729, 0.795
B+ tree saved to disk: ft_pct_home.idx
```

Explanation:

All deletions are at the last leaf , >0.9 is above 0.87
Since n =100 , the minimum number of keys must be between ⌈n/2⌉ and n keys → 50 to 100 keys, After deletion of keys from >0.9, it reduces the number of key in the 0.87 root node to below 50, thus it performs a merge with 0.795 while still ensuring that the root node contains 50-100 keys.

Performance Analysis

```
=== Brute-force Comparison ===
Linear scan would access: 144 data blocks
Linear scan estimated time: ~720 ms (estimated)
Assumption: 5 ms per block access and scan
Index speedup: ~10.0989x faster
```

Key Insight

Matches are scattered across the heap, so both plans touch all 144 pages. The B+ tree still wins because it avoids a full record check path and coordinates access via the leaf range, yielding ~10× shorter retrieval time under the fixed timing brute force access.

Key Findings & Conclusions

Efficient Space Utilization

Compact 22-byte records and 4KB blocks achieve near-ideal space utilization, storing 26,651 NBA game records in 144 blocks with minimal overhead.

B+ Tree Performance

The B+ tree with n=100 achieves 2-level structure with 8 nodes, providing efficient data retrieval.

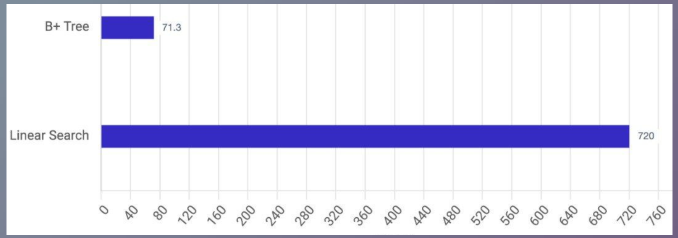
Record Deletion Efficiency

Successfully deleted 1,778 records with FT_PCT_home > 0.9, updating the B+ tree index while maintaining structural integrity.

NBA Data Insights

The NBA dataset (2003-2022) shows a mean FT_PCT_home of 0.795 with a standard deviation of 0.114, with values ranging from 0.143 to 1.

B+ Tree vs. Linear Scan Performance



Project Success

This project successfully implemented a disk-based storage system and B+ tree index, demonstrating efficient data storage and retrieval. The B+ tree structure reduced query time by over 10x compared to linear scanning, while maintaining structural integrity during record deletions.

Thank You!