B-Trees and B+ Trees

Disk-Friendly Balanced Trees for Indexing and Storage

Minseok Jeon
DGIST

November 2, 2025

Table of Contents

Introduction

What are B-Trees?

B-Trees: Self-balancing tree data structures optimized for disk storage

Key Characteristics:

- Generalization of binary search trees (multi-way trees)
- Designed for systems with slow, block-based storage (disks)
- Minimize disk I/O operations
- All leaves at the same depth (perfectly balanced)
- High branching factor (many children per node)

Why B-Trees?

- Disk access is 100,000x slower than RAM
- Reading a disk block has fixed cost (4KB, 8KB)
- Solution: Pack many keys per node to reduce tree height

Historical Context

Invented in 1970 by Rudolf Bayer and Edward McCreight

Timeline:

- **1970**: B-Tree invented at Boeing Research Labs
- **1979**: B+ Tree variant introduced
- **1980s**: Adopted by major database systems
- **Today**: Standard for database indexes and filesystems

Impact:

- Revolutionized database indexing
- Enabled efficient large-scale data storage
- Foundation of modern relational databases

Node Structure and Order

Node Structure and Order

Order (m): Maximum number of children a node can have

Node Properties:

- Each node contains up to $m - 1$ keys
- Each node has up to m children
- Keys stored in sorted order
- **Internal nodes**: keys + child pointers
- **Leaf nodes**: keys + data (or pointers to data)

Common Order Values:

- Small trees: $m = 3, 5, 7$
- Disk-based systems: $m = 100$ to 1000

B-Tree Node Implementation

```python
class BTreeNode:
    def __init__(self, order, is_leaf=False):
        self.order = order                # Maximum children
        self.keys = []                    # Sorted keys
        self.children = []                # Child pointers
        self.is_leaf = is_leaf            # Leaf flag
        self.n = 0                        # Current number of keys
```

Node Constraints:

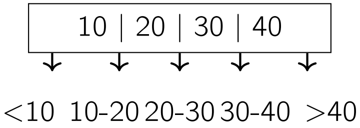
- **Root**: 1 to $m - 1$ keys, 2 to m children (if not leaf)
- **Internal nodes**: $\lceil m/2 \rceil - 1$ to $m - 1$ keys, $\lceil m/2 \rceil$ to m children
- **Leaf nodes**: $\lceil m/2 \rceil - 1$ to $m - 1$ keys
- **All leaves at same depth** (balanced)

Example: B-Tree of Order 5

Node Structure ($m = 5$):

- Min keys (internal): $\lceil 5/2 \rceil - 1 = 2$
- Max keys: 4
- Min children (internal): $\lceil 5/2 \rceil = 3$
- Max children: 5

Example Node:



Memory Layout Considerations

Node Size Matching Disk Block Size

Example Calculation:

- Disk block size: 4KB (4096 bytes)
- Key size: 8 bytes
- Pointer size: 8 bytes
- Available space: $\approx$ 4000 bytes (accounting for metadata)

Order Calculation:

- Node contains: m pointers + $(m-1)$ keys
- Space: $8m + 8(m-1) = 16m - 8$ bytes
- $16m - 8 \leq 4000$
- $m \leq 250.5$
- **Order** $m \approx 250$

Impact: 250-way branching means very shallow trees!

Insertion and Split

Insertion Algorithm Overview

Steps:

1. **Search for leaf**: Traverse tree to find appropriate leaf node
2. **Insert in leaf**: Add key in sorted position
3. **Check overflow**: If node has m keys (too many), split
4. **Split operation**:
   - Create new node
   - Move upper half of keys to new node
   - Promote middle key to parent
   - If parent overflows, split recursively up to root

Key Property: Splits propagate upward, may create new root

Insertion Implementation - Main Function

```python
def insert(root, key):
    # If root is full, split it
    if root.n == root.order - 1:
        new_root = BTreeNode(root.order)
        new_root.children.append(root)
        split_child(new_root, 0)
        root = new_root

    insert_non_full(root, key)
    return root

def insert_non_full(node, key):
    if node.is_leaf:
        # Insert key in sorted position
        node.keys.insert(bisect_left(node.keys, key), key)
        node.n += 1
    else:
        # Find child to insert into
        i = bisect_left(node.keys, key)
        if node.children[i].n == node.order - 1:
            split_child(node, i)
            if key > node.keys[i]:
                i += 1
        insert_non_full(node.children[i], key)
```

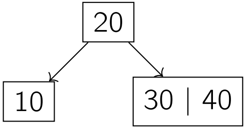
Split Child Operation

```python
def split_child(parent, index):
    full_child = parent.children[index]
    new_child = BTreeNode(full_child.order, full_child.is_leaf)

    mid = full_child.order // 2

    # Promote middle key to parent
    parent.keys.insert(index, full_child.keys[mid])
    parent.children.insert(index + 1, new_child)

    # Split keys and children
    new_child.keys = full_child.keys[mid+1:]
    full_child.keys = full_child.keys[:mid]

    if not full_child.is_leaf:
        new_child.children = full_child.children[mid+1:]
        full_child.children = full_child.children[:mid+1]

    # Update counts
    new_child.n = len(new_child.keys)
    full_child.n = len(full_child.keys)
    parent.n += 1
```

Insertion Example: Step-by-Step

Insert into B-Tree of order 3 (max 2 keys per node)

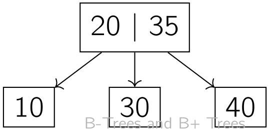
Initial tree:



Insert 35:

- Navigate to right child [30 | 40]
- Insert 35: [30 | 35 | 40] - overflow!
- Split: promote 35 to parent

After split:



Deletion and Merge

Deletion Algorithm Overview

More Complex than Insertion - Three Cases:

Case 1: Key in Leaf Node

- Simply remove the key
- Check for underflow

Case 2: Key in Internal Node

- Replace with predecessor or successor
- Delete predecessor/successor from leaf
- Handle any resulting underflow

Case 3: Underflow Handling

- Node has fewer than $\lceil m/2 \rceil - 1$ keys
- **Option A**: Borrow from sibling (if sibling has extra keys)
- **Option B**: Merge with sibling (if sibling at minimum)

Deletion Case 1: Delete from Leaf

Simple Case - No Underflow

Before:

10	20	30

Delete 20

After:

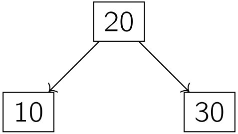
10	30

Condition: Resulting node still has $\geq \lceil m/2 \rceil - 1$ keys

Deletion Case 2: Delete from Internal Node

Replace with Predecessor

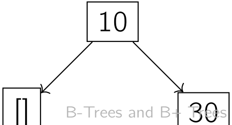
Before:



Delete 20:

- Find predecessor (largest in left subtree): 10
- Replace 20 with 10
- Delete 10 from leaf

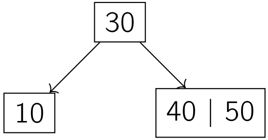
After:



Deletion Case 3: Merge Siblings

When Underflow Occurs and Sibling Can't Lend

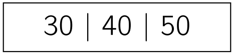
Before:



Delete 10:

- Left child becomes empty (underflow)
- Right sibling has only 2 keys (can't lend)
- **Solution**: Merge left and right with parent key 30

After merge:



Result: Parent key pulled down, siblings merged

Height and Complexity

Height Formula and Analysis

For n keys and order m:

Height Bounds:

- **Minimum height**: $\log_m(n+1)$ (fully packed nodes)
- **Maximum height**: $\log_{\lceil m/2 \rceil}((n+1)/2)$ (minimum occupancy)

Example: 1 Million Keys, Order $m = 100$

- Minimum children per internal node: $\lceil 100/2 \rceil = 50$
- Height $\leq \log_{50}(1,000,000) \approx 3.5$
- **Only 4 disk accesses to find any key!**

Key Insight:

- High branching factor dramatically reduces height
- Each level = one disk I/O
- Shallow tree = fast queries

Time Complexity Analysis

Operation	Time Complexity	Disk I/Os
Search	$O(\log_m n)$	$O(\log_m n)$
Insert	$O(\log_m n)$	$O(\log_m n)$
Delete	$O(\log_m n)$	$O(\log_m n)$
Range Scan	$O(\log_m n + k)$	$O(\log_m n + k/b)$

Where:

- n = number of keys
- m = order (branching factor)
- k = number of results in range query
- b = keys per block

Space Complexity: $O(n)$

- Minimum 50% space utilization (except root)
- Average 67-75% utilization in practice

Comparison with Binary Search Trees

Tree Type	Height for 1M keys	Disk I/Os
Binary tree (balanced)	$\log_2(10^6) \approx 20$	20
B-Tree ($m = 100$)	$\log_{100}(10^6) \approx 3$	3
B-Tree ($m = 1000$)	$\log_{1000}(10^6) \approx 2$	2

Why B-Trees are Efficient:

- **High branching factor** $\rightarrow$ shallow tree
- **Fewer disk accesses** (dominant cost in I/O-bound systems)
- **Each node fits in one disk block** (4KB, 8KB)
- **Sequential access within nodes** (cache-friendly)

Result: 6-7x reduction in disk I/Os compared to balanced BST

B-Tree vs B+ Tree

B-Tree Structure

Classic B-Tree Characteristics:

Structure:

- Keys and data in **all nodes** (internal + leaf)
- Each key appears **exactly once**
- No linked list between leaves

Advantages:

- Better for **exact-match queries** (may find in internal node)
- Slightly less space (no duplicate keys)

Disadvantages:

- Range queries less efficient
- Variable-size records complicate node management
- Must traverse tree for sequential access

B+ Tree Structure

Enhanced Variant for Databases:

Structure:

- All data in **leaf nodes only**
- Internal nodes contain only keys (for routing)
- Keys may be duplicated (in internal + leaf)
- **Leaves linked in sorted order** (doubly linked list)

Advantages:

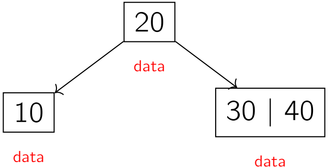
- **Excellent for range queries** (scan linked leaves)
- More keys per internal node (no data overhead)
- Sequential access via leaf chain
- Consistent performance (always reach leaf)

Disadvantages:

- Duplicate keys use extra space
- Always traverse to leaf (even if key in internal node)

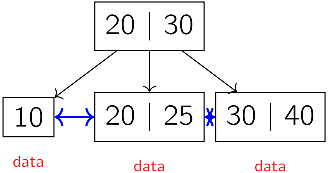
Visual Comparison

B-Tree (order 3):



All nodes contain data

B+ Tree (order 3):



Only leaves contain data, leaves are linked

Detailed Comparison Table

Feature	B-Tree	B+ Tree
Data location	All nodes	Leaf only
Internal nodes	Keys + data	Keys only
Key duplication	No	Yes
Leaf linkage	No	Yes
Keys per internal	Fewer	More
Range queries	$O(\log n + k)$	$O(\log n + k/b)$
Point queries	Faster	Always to leaf
Sequential access	Poor	Excellent
Use case	General	DB/Filesystem

Why Databases Prefer B+ Trees

Five Key Reasons:

1. **Range Queries:**
   - Common in SQL: `SELECT * WHERE age BETWEEN 20 AND 30`
   - Efficient leaf chain scanning
2. **Sequential Scans:**
   - Full table scans via leaf chain
   - No need to traverse internal nodes repeatedly
3. **Higher Fanout:**
   - More keys per internal node → shorter tree
   - Internal nodes don't store data
4. **Predictable Performance:**
   - Always same depth to leaf
   - Consistent query response times
5. **Easier Concurrency:**
   - Lock leaves independently

Range Scans and Storage Locality

Range Query in B+ Tree

Algorithm:

1. **Find start key**: $O(\log_m n)$ - traverse to leaf
2. **Scan leaves**: Follow linked list until end key
3. **Total cost**: $O(\log_m n + k)$ where k = results

Example Query:

```
SELECT * FROM users WHERE age BETWEEN 25 AND 35
```

Execution Steps:

- Step 1: Search for `age=25` $\rightarrow$ reach leaf L_1
- Step 2: Scan $L_1 \rightarrow L_2 \rightarrow L_3$ until `age > 35`
- Step 3: Return all records found

Disk I/Os:

- 1 (root) + 1 (internal) + 1 (leaf) + k/b (scan)
- Much better than k separate point queries!

Range Query in B-Tree

Less Efficient - Must Use In-Order Traversal

Problem:

- No leaf linkage
- Must jump between internal and leaf nodes
- Random access pattern

Complexity:

- $O(\log_m n + k \log_m n)$ - revisit internal nodes
- Much worse than B+ Tree for large ranges

Example:

- Range query with 1000 results
- B+ Tree: $3 + 10 = 13$ I/Os (assuming 100 keys/block)
- B-Tree: $3 + 1000 \times 3 = 3003$ I/Os

Storage Locality Benefits

Sequential Disk Access:

- Leaves stored contiguously on disk
- Operating system prefetches adjacent blocks
- Minimizes seek time (critical for HDDs)

Cache-Friendly:

- Scanning leaves keeps data in cache
- No random jumps between levels
- High cache hit rate

Performance Impact:

- **Random access**: 10ms per seek (HDD)
- **Sequential access**: 100MB/s throughput
- Locality can provide **100x speedup** for range queries

Bulk Loading

Building B+ Tree Bottom-Up

Algorithm:

1. Sort all keys
2. Create leaves left-to-right
3. Build internal levels bottom-up

Advantages:

- **100% space utilization** (vs 67% for incremental insert)
- Optimal storage locality
- Much faster than individual inserts
- Speed: 100,000+ ops/sec vs 1,000-10,000 for random inserts

Use Cases:

- Initial database load
- Index rebuilding
- Data warehouse ETL

Bulk Loading Implementation

```python
def bulk_load(sorted_keys):
    # Create leaf level
    leaves = []
    for i in range(0, len(sorted_keys), LEAF_SIZE):
        leaf = create_leaf(sorted_keys[i:i+LEAF_SIZE])
        leaves.append(leaf)

    # Link leaves
    for i in range(len(leaves)-1):
        leaves[i].next = leaves[i+1]
        leaves[i+1].prev = leaves[i]

    # Build internal levels bottom-up
    return build_internal_levels(leaves)

def build_internal_levels(nodes):
    while len(nodes) > 1:
        parents = []
        for i in range(0, len(nodes), ORDER):
            parent = create_internal_node(nodes[i:i+ORDER])
            parents.append(parent)
        nodes = parents
    return nodes[0]  # Root
```

Optimization: Prefix Compression

Store Only Distinguishing Prefix in Internal Nodes

Example:

- Full keys: ["apple", "application", "apply"]
- Compressed: ["app", "appl"]
- Savings: 50% space in internal nodes

Benefits:

- Higher fanout (more keys per node)
- Shorter tree height
- Fewer disk I/Os

Implementation:

```
def compress_key(left_key, right_key):
    # Find shortest prefix that distinguishes keys
    for i in range(min(len(left_key), len(right_key))):
        if left_key[i] != right_key[i]:
            return left_key[:i+1]
    return left_key  # One is prefix of other
```

B+ Tree Leaf Node Implementation

```python
class BPlusTreeLeaf:
    def __init__(self, order):
        self.order = order
        self.keys = []           # Sorted keys
        self.values = []         # Corresponding values/data
        self.next = None         # Next leaf (right sibling)
        self.prev = None         # Previous leaf (left sibling)
        self.parent = None       # Parent node
        self.n = 0               # Current number of keys

    def range_scan(self, start_key, end_key):
        """Efficient range query using leaf chain"""
        results = []
        current = self

        # Find starting position in first leaf
        start_idx = bisect_left(current.keys, start_key)

        # Scan leaves until end_key
        while current:
            for i in range(start_idx, current.n):
                if current.keys[i] > end_key:
                    return results
                results.append((current.keys[i], current.values[i]))
            current = current.next
            start_idx = 0  # Start from beginning in subsequent leaves

        return results
```

Database and Filesystem Applications

Database Indexes

Primary Index: B+ Tree on primary key

- Leaf nodes contain actual data rows (clustered index)
- Example: MySQL InnoDB primary key index
- Data physically sorted by primary key

Secondary Index: B+ Tree on non-primary key

- Leaf nodes contain pointers to primary key
- Example: Index on email column
- Requires two lookups: secondary index $\rightarrow$ primary index

Composite Index: Multi-column B+ Tree

- Keys are tuples: `(last_name, first_name)`
- Supports queries on prefix: `WHERE last_name = 'Smith'`
- Left-to-right column ordering matters

Database Systems Using B+ Trees

Database	Index Type	Details
MySQL InnoDB	B+ Tree	Clustered PK, secondary indexes
PostgreSQL	B-Tree*	Actually B+ Tree, default type
SQLite	B+ Tree	Table and index storage
Oracle	B+ Tree	Index-organized tables
SQL Server	B+ Tree	Clustered and non-clustered

Note: PostgreSQL calls it "B-Tree" but implements B+ Tree variant

Example: MySQL InnoDB

```sql
-- Clustered index (B+ Tree on primary key)
CREATE TABLE users (
    id INT PRIMARY KEY,          -- B+ Tree root
    name VARCHAR(100),
    email VARCHAR(100),
    age INT,
    INDEX idx_email (email),     -- Secondary B+ Tree
    INDEX idx_age_name (age, name)  -- Composite B+ Tree
);

-- Range query (efficient - uses B+ Tree leaf chain)
SELECT * FROM users WHERE id BETWEEN 1000 AND 2000;

-- Composite index query (uses idx_age_name)
SELECT * FROM users WHERE age = 25 AND name LIKE 'J%';

-- Index-only scan (covering index)
SELECT age, name FROM users WHERE age BETWEEN 20 AND 30;
-- All data in B+ Tree leaves, no table lookup needed
```

Example: PostgreSQL B-Tree Index

```sql
-- Create index
CREATE INDEX idx_users_age ON users(age);

-- Explain query plan
EXPLAIN SELECT * FROM users WHERE age > 25;
-- Output:
-- Index Scan using idx_users_age
--   Index Cond: (age > 25)

-- Composite index for multiple columns
CREATE INDEX idx_users_city_age ON users(city, age);

-- Query using composite index
SELECT * FROM users WHERE city = 'Seoul' AND age BETWEEN 20 AND 30;
-- Uses idx_users_city_age for both conditions

-- Index-only scan (no table access)
EXPLAIN SELECT age FROM users WHERE age > 25;
-- Output:
-- Index Only Scan using idx_users_age
--   Index Cond: (age > 25)
```

Filesystem Applications

File Allocation:

- B+ Tree maps file blocks to disk blocks
- Fast random access within files
- Efficient sparse file support

Directory Structure:

- B+ Tree for large directories (thousands of files)
- Efficient filename lookups
- Example: `/usr/bin` with 10,000 files

Filesystem	B-Tree Usage
ext4	HTree (B-Tree) for directories
XFS	B+ Trees for free space, inodes
Btrfs	B-Trees for all metadata
NTFS	B+ Trees for file records (MFT)
HFS+	B-Trees for catalog file

Filesystem Example: Directory Lookup

Without B-Tree:

- Directory: `/usr/bin` (10,000 files)
- Lookup: find "python3"
- Method: Linear scan through directory entries
- Complexity: $O(n)$ - 10,000 comparisons

With B-Tree (ext4 HTree):

- Same directory with B-Tree index
- Lookup: "python3"
- Method: B-Tree search
- Complexity: $O(\log n) \approx 4$ disk accesses

Performance Impact:

- **2500x faster** for large directories
- Critical for directories with many files
- Example: `/var/mail`, `/tmp`

Performance Characteristics

Insert Performance:

- Random inserts: 1,000-10,000 ops/sec
- Bulk inserts: 100,000+ ops/sec (bulk loading)

Query Performance:

- Point query: 1-3 disk I/Os (typical depth)
- Range query: 1-3 + k/b I/Os (k results, b per block)

Space Overhead:

- 50-75% space utilization (minimum 50%)
- Internal nodes: 1-2% of total space
- Leaf nodes: 98-99% of total space

Real-World Example:

- MySQL InnoDB with 10M rows
- Index size: $\approx$ 500MB
- Query time: 1-5ms (warm cache), 10-50ms (cold)

Advanced Features

Write-Ahead Logging (WAL):

- Log changes before applying to B+ Tree
- Enables crash recovery
- Used in PostgreSQL, SQLite

MVCC (Multi-Version Concurrency Control):

- Multiple versions of same row
- Readers don't block writers
- Used in PostgreSQL, InnoDB

Compression:

- Prefix compression (internal nodes)
- Page compression (entire blocks)
- Higher fanout, better performance

Partitioning:

- Distribute B+ Tree across multiple disks

Summary

Key Takeaways

B-Trees and B+ Trees:

- **Purpose**: Disk-friendly balanced trees for large datasets
- **Key idea**: High branching factor $\to$ shallow tree $\to$ few I/Os
- **Order** m: Typically 100-1000 for disk-based systems

Operations:

- Search, Insert, Delete: $O(\log_m n)$ time, $O(\log_m n)$ I/Os
- Split on overflow, merge on underflow
- Maintains balance automatically

B+ Tree Advantages:

- All data in leaves $\to$ better range queries
- Leaf linkage $\to$ efficient sequential scans
- More keys per internal node $\to$ shorter tree
- **Standard for databases and filesystems**

When to Use B-Trees

Use B-Trees/B+ Trees when:

- Data doesn't fit in memory (disk-based storage)
- Need efficient range queries
- Building database indexes
- Implementing filesystems
- Sequential access patterns common

Don't use when:

- Data fits in memory (use hash tables, AVL/Red-Black trees)
- Only point queries (hash tables may be faster)
- Frequent updates to same keys (consider LSM-trees)

Modern Alternatives:

- **LSM-Trees**: Write-optimized (Cassandra, RocksDB)
- **Tries**: String-specific (Redis)
- **Skip Lists**: Simpler implementation (Redis, LevelDB)

Practice Problems

Problem 1: Height Calculation

- Given: 1 billion keys, order $m = 500$
- Question: What is the maximum tree height?
- Hint: Use $\log_{\lceil m/2 \rceil}((n+1)/2)$

Problem 2: Insertion Trace

- Insert keys [10, 20, 30, 40, 50] into empty B-Tree (order 3)
- Draw tree after each insertion
- Show all split operations

Problem 3: B+ Tree Range Query

- Given: B+ Tree with 1M keys, order 100
- Query: `SELECT * WHERE id BETWEEN 1000 AND 2000`
- Calculate: Number of disk I/Os

Resources

Academic Papers:

- Bayer & McCreight (1972): "Organization and Maintenance of Large Ordered Indexes"
- Comer (1979): "The Ubiquitous B-Tree"

Books:

- "Database System Concepts" (Silberschatz et al.)
- "Introduction to Algorithms" (CLRS) - Chapter 18

Online Resources:

- MySQL InnoDB documentation
- PostgreSQL B-Tree implementation details
- Visualization: `https://www.cs.usfca.edu/g̃alles/visualization/`

Implementation Projects:

- Build your own B+ Tree in Python/C++
- Implement database index using B+ Tree