BTreePlus

User Guide & API Reference

Version 1.2.7 • November 2025

Introduction

BTreePlus is a high-performance B+Tree index engine for .NET Standard 2.0. It provides deterministic memory usage, predictable I/O patterns, and strong durability guarantees for embedded and standalone applications where a full database is unnecessary.

What BTreePlus Is

- A true B+Tree implementation (values in leaves only)
- Zero-allocation lookups using Span<byte>
- File-backed or in-memory storage
- Thread-safe with automatic or manual commits
- Fixed-size keys and values for predictable performance

What BTreePlus Is NOT

- Not a database or query engine
- Not an ORM or object storage system
- Not designed for variable-length keys/values
- Not a replacement for SQL databases

> ***Target Users:*** *Developers building POS terminals, kiosks, industrial controllers, edge devices, local-first apps, message queues, or custom storage engines.*

Installation

.NET CLI

```
dotnet add package BTreePlus --version 1.2.7
```

Package Manager

```
Install-Package BTreePlus -Version 1.2.7
```

PackageReference

```
<PackageReference Include="BTreePlus" Version="1.2.7" />
```

Quick Start

In-Memory Tree

```
using mmh;

var bt = BTree.CreateMemory(
    keyBytes: 10,
    dataBytes: 4,
    pageSize: 1,
    max_recs: 10_000_000,
    balance: false);

byte[] key = Encoding.UTF8.GetBytes("1234567890");
byte[] value = BitConverter.GetBytes(1234);

bt.Insert(key, value);
bt.Commit();
bt.Close();
```

File-Backed Tree

```
using mmh;

var bt = BTree.CreateOrOpen("data.btp",
    keyBytes: 10,
    dataBytes: 4,
    pageSize: 4,
    enableCache: true,
    max_recs: 32_000_000,
    balance: true);

byte[] key = Encoding.UTF8.GetBytes("1234567890");
byte[] value = BitConverter.GetBytes(1234);

bt.Insert(key, value);
bt.Commit();
bt.Close();

// Reopen later
var bt2 = BTree.Open("data.btp", enableCache: true, balance: true);
```

Core API Reference

Creation & Opening

CreateMemory()

```
BTree CreateMemory(
    int keyBytes,
    int dataBytes,
    ushort pageSize = 1,
    long max_recs = 10_000_000,
    bool balance = false)
```

Creates an in-memory B+Tree. Data is not persisted.

CreateOrOpen()

```
BTree CreateOrOpen(
    string path,
    int keyBytes,
    int dataBytes,
    ushort pageSize = 4,
    bool enableCache = false,
    long max_recs = 32_000_000,
    bool balance = false)
```

Creates a new file or opens existing. Parameters must match for existing files.

Open()

```
BTree Open(
    string path,
    bool enableCache = false,
    long max_recs = 32_000_000,
    bool balance = false)
```

Opens an existing file. Reads parameters from file header.

Parameters Explained

Parameter	Type	Description
keyBytes	int	Fixed key length (all keys must be this size)
dataBytes	int	Fixed value length (all values must be this size)
pageSize	ushort	Page size in multiples of 512 bytes. Range: 1-128. Default: 1 for memory, 4 for disk
enableCache	bool	Enable write-back cache. Only for file-backed trees. Recommended: true
balance	bool	Enable rebalancing on delete. Default: false. Recommended: true for production

max_recs	long	Estimated max records for PAT sizing. Default: 10M for memory, 32M for disk

Data Operations

Insert()

```
bool Insert(ReadOnlySpan<byte> key, ReadOnlySpan<byte> data, bool bLock = true)

// Examples:
byte[] key = Encoding.UTF8.GetBytes("1234567890");
byte[] value = BitConverter.GetBytes(42);
bool success = bt.Insert(key, value);

// Stack-allocated (zero heap allocation):
Span<byte> key = stackalloc byte[10];
Span<byte> value = stackalloc byte[4];
Encoding.UTF8.GetBytes("1234567890").CopyTo(key);
BitConverter.GetBytes(42).CopyTo(value);
bool success = bt.Insert(key, value);
```

Returns: true if inserted, false if key already exists (duplicates not allowed).
Thread Safety: Pass bLock=true (default) for automatic locking.

Find()

```
bool Find(ReadOnlySpan<byte> key, Span<byte> data, bool bLock = true)

// Example:
Span<byte> key = stackalloc byte[10];
Span<byte> value = stackalloc byte[4];
Encoding.UTF8.GetBytes("1234567890").CopyTo(key);

if (bt.Find(key, value))
{
    int result = BitConverter.ToInt32(value);
    Console.WriteLine($"Found: {result}");
}
```

Returns: true if found (value written to data span), false if not found.
Performance: Zero allocations - value is written directly to your buffer.

Erase()

```
bool Erase(ReadOnlySpan<byte> key, bool bLock = true)

// Example:
Span<byte> key = stackalloc byte[10];
Encoding.UTF8.GetBytes("1234567890").CopyTo(key);
bool removed = bt.Erase(key);
```

Returns: true if deleted, false if key not found.
Rebalancing: If balance=true, automatically merges/redistributes underfull pages.

Iteration & Range Queries

Bof() / Next()

```
void Bof()  // Move to beginning of tree
bool Next(out ReadOnlySpan<byte> key, Span<byte> data)

// Example - iterate all records:
bt.Bof();
ReadOnlySpan<byte> key;
Span<byte> value = stackalloc byte[4];

while (bt.Next(out key, value))
{
    string keyStr = Encoding.UTF8.GetString(key);
    int val = BitConverter.ToInt32(value);
    Console.WriteLine($"{keyStr} => {val}");
}
```

FindRange()

```
List<(byte[] key, byte[] data)> FindRange(
    ReadOnlySpan<byte> fromKey,
    ReadOnlySpan<byte> toKey,
    bool acquireLock = true)

// Example - range query:
byte[] from = Encoding.UTF8.GetBytes("A000000000");
byte[] to = Encoding.UTF8.GetBytes("B000000000");

var results = bt.FindRange(from, to);
foreach (var (key, data) in results)
{
    Console.WriteLine($"{Encoding.UTF8.GetString(key)} => {BitConverter.ToInt32(data)}");
}
```

FindRangeValues()

```
List<T> FindRangeValues<T, TProj>(
    ReadOnlySpan<byte> fromKey,
    ReadOnlySpan<byte> toKey,
    TProj projector,
    int initialCapacity = 0,
    bool acquireLock = true)
    where TProj : struct, IValueProjector<T>

// Example - fast integer extraction:
var intResults = bt.FindRangeInt(from, to);
foreach (int value in intResults)
{
    Console.WriteLine(value);
}
```

High-performance range query that projects values directly without allocating key/data pairs.

Bulk Operations

BulkLoad()

```
void BulkLoad(
    IEnumerable<(byte[] key, byte[] data)> records,
    double leafFillFactor = 0.90)

// Example - fast initial load:
var records = new List<(byte[], byte[])>();
for (int i = 0; i < 1_000_000; i++)
{
    byte[] key = Encoding.UTF8.GetBytes(i.ToString("D10"));
    byte[] value = BitConverter.GetBytes(i);
    records.Add((key, value));
}

bt.BulkLoad(records);  // 2-5M records/sec
```

Important: Tree must be empty. Records are automatically sorted. Much faster than sequential inserts for initial data loading (2-5M records/sec vs ~400K records/sec).

Durability & Commits

Commit()

```
void Commit()

// Example:
bt.Insert(key1, value1);
bt.Insert(key2, value2);
bt.Commit();  // Flush all changes to disk
```

Flushes all dirty pages, PAT, and header to disk. Data is fully durable after Commit() returns. Auto-commit happens every 1 second by default for file-backed trees.

Close()

```
void Close()

// Example:
bt.Close();  // Commits and releases resources
```

Commits any pending changes and closes the file. Always call Close() when done.

Count

```
int Count { get; }

// Example:
int totalRecords = bt.Count;
```

Returns total number of records across all leaf pages. This is a counted operation.

Dictionary Wrapper API

BTreePlus includes a type-safe dictionary wrapper that implements IDictionary<TKey, TValue>. Use BTreeDictionaryFactory to create common types.

Long to Long Dictionary

```
using mmh;

var bt = BTree.CreateMemory(8, 8);  // 8-byte keys, 8-byte values
var dict = BTreeDictionaryFactory.CreateLongToLong(bt, bigEndian: true);

dict[12345L] = 67890L;
dict.Add(11111L, 22222L);

if (dict.TryGetValue(12345L, out long value))
{
    Console.WriteLine($"Value: {value}");
}

foreach (var kv in dict)
{
    Console.WriteLine($"{kv.Key} => {kv.Value}");
}
```

Int to Int Dictionary

```
var bt = BTree.CreateMemory(4, 4);
var dict = BTreeDictionaryFactory.CreateIntToInt(bt, bigEndian: true);

dict[100] = 200;
int value = dict[100];
```

String Key Dictionaries

```
// ASCII strings (fixed width, right-aligned)
var bt = BTree.CreateMemory(20, 4);
var dict = BTreeDictionaryFactory.CreateAsciiStringKey<int>(
    bt,
    value => BitConverter.GetBytes(value),
    span => BitConverter.ToInt32(span),
    rightAlign: true);

dict["product123"] = 42;

// UTF-8 strings
var dict2 = BTreeDictionaryFactory.CreateUtf8StringKey<int>(
    bt,
    value => BitConverter.GetBytes(value),
    span => BitConverter.ToInt32(span),
    rightAlign: true);
```

Note: *String keys are fixed-width. Shorter strings are padded. Right-align is recommended for lexicographic sorting of numeric strings.*

Sharding (Advanced)

For multi-million record workloads, use ShardManager to distribute data across multiple B+Trees using FNV-1a hashing.

```
using mmh;

var shards = new ShardManager(
    basePath: "data/shard",
    keyBytes: 10,
    valBytes: 4,
    pageSize: 16,
    shardCount: 8,           // Power of 2 recommended
    enableCache: true,
    max_recs: 32_000_000);

// Single inserts (automatically routed)
shards.Insert(key, value);

// Batch inserts (sorted by shard, parallel writes)
var batch = new (byte[], byte[])[10000];
// ... populate batch ...
shards.InsertBatch(batch);
```

Performance: Sharding eliminates lock contention and enables parallel writes. 8 shards can achieve ~8x throughput on multi-core systems.

Performance Tuning Guide

Choosing pageSize

pageSize	Physical Size	Use Case
1	512 bytes	Embedded systems, minimal memory
4	2 KB	General workloads, default for disk
8	4 KB	Balanced performance
16	8 KB	Good for most workloads
32	16 KB	High throughput
64	32 KB	NVMe/SSD workloads
128	64 KB	Maximum (very high throughput)

Formula: Physical page size = pageSize × 512 bytes
Example: pageSize=4 → 2 KB pages, pageSize=16 → 8 KB pages, pageSize=128 → 64 KB pages

Cache Configuration

Enable cache for file-backed trees:

```
var bt = BTree.CreateOrOpen("data.btp", ..., enableCache: true);

// Cache reduces disk I/O by ~90%
// Without cache: ~400K inserts/sec
// With cache: 1-3M inserts/sec
```

Key Design Guidelines

1. **Use fixed-width keys** - pad strings to keyBytes length

2. **Right-align numeric strings** - ensures proper sorting

3. **Big-endian integers** - sorts correctly as byte arrays

4. **Minimize keyBytes** - smaller keys = more records per page

```
// Good: right-aligned numeric string
byte[] key = new byte[10];
string id = "12345";
Encoding.ASCII.GetBytes(id).CopyTo(key, 10 - id.Length);
// Result: [0x00, 0x00, 0x00, 0x00, 0x00, 0x31, 0x32, 0x33, 0x34, 0x35]

// Good: big-endian int64
```

```
byte[] key = new byte[8];
BinaryPrimitives.WriteInt64BigEndian(key, 12345);
```

Common Usage Patterns

Pattern 1: Sequential ID Assignment

```
// Auto-incrementing IDs
long nextId = bt.Count + 1;
byte[] key = new byte[8];
BinaryPrimitives.WriteInt64BigEndian(key, nextId);
bt.Insert(key, data);
```

Pattern 2: Time-Series Data

```
// Unix timestamp as key
long timestamp = DateTimeOffset.UtcNow.ToUnixTimeSeconds();
byte[] key = new byte[8];
BinaryPrimitives.WriteInt64BigEndian(key, timestamp);
bt.Insert(key, sensorData);

// Query last hour
long hourAgo = timestamp - 3600;
byte[] from = new byte[8];
byte[] to = new byte[8];
BinaryPrimitives.WriteInt64BigEndian(from, hourAgo);
BinaryPrimitives.WriteInt64BigEndian(to, timestamp);
var recent = bt.FindRange(from, to);
```

Pattern 3: Composite Keys

```
// Category + ID composite key
byte[] key = new byte[12];
Encoding.ASCII.GetBytes("CAT").CopyTo(key, 0);  // 3 bytes category
BinaryPrimitives.WriteInt64BigEndian(key.AsSpan(4), productId);  // 8 bytes ID

// Query all in category
byte[] from = new byte[12];
byte[] to = new byte[12];
Encoding.ASCII.GetBytes("CAT").CopyTo(from, 0);
Encoding.ASCII.GetBytes("CAT").CopyTo(to, 0);
Array.Fill(to, (byte)0xFF, 4, 8);  // Max out ID portion
var categoryItems = bt.FindRange(from, to);
```

Troubleshooting

Key/Value Length Mismatch

Error: ArgumentException on Insert/Find

Cause: Key or value doesn't match keyBytes/dataBytes

Fix: Always pad/truncate to exact length

```
// Wrong - variable length
byte[] key = Encoding.UTF8.GetBytes("abc");  // Only 3 bytes!

// Correct - fixed length
byte[] key = new byte[10];
Encoding.UTF8.GetBytes("abc").CopyTo(key, 0);  // Padded with zeros
```

File Corruption After Crash

Symptom: Cannot open file after power loss

Cause: Partial write to disk

Prevention: BTreePlus guarantees consistency after Commit(). Uncommitted data may be lost, but file remains openable.

Slow Performance

Check 1: Is cache enabled for file-backed trees?

Check 2: Are you using pageSize=16 or higher for disk?

Check 3: For initial load, use BulkLoad() instead of Insert()

Check 4: Consider sharding for multi-million record workloads

Out of Memory

Cause: Cache grew too large

Fix: Reduce pageSize or call Commit() more frequently

```
// Force periodic commits
for (int i = 0; i < records.Length; i++)
{
    bt.Insert(records[i].key, records[i].value);
    if (i % 100000 == 0)
        bt.Commit();  // Flush every 100K records
}
```

Best Practices

DO:

✓ Always Close() the tree when done

✓ Use cache for file-backed trees

✓ Consider BulkLoad() for pre-sorted initial data (faster)

✓ Use Span<byte> for zero-allocation lookups

✓ Call Commit() explicitly for critical data

✓ Use pageSize 16-32 for most workloads

✓ Right-align numeric strings in keys

DON'T:

✗ Don't use variable-length keys/values

✗ Don't forget to Close() before exit

✗ Don't disable cache for file-backed trees

✗ Don't assume data persists without Commit()

✗ Don't use pageSize > 128 (capped at 128)

Support & Licensing

Community Version

BTreePlus 1.2.7 is available under the MIT license for community use. It includes:

• Core insert/find operations

• Single-key operations

• Basic range queries

• File and memory storage

Enterprise Features

For production deployments requiring advanced features:

• Erase (delete) with rebalancing

• Fast range scans with projections

• Maintenance and optimization tools

• Commercial support and integration services

• Embedded device licensing

Contact: btplus@mmhsys.com

Appendix: Technical Specifications

Specification	Value
Target Framework	.NET Standard 2.0
Minimum Runtime	.NET Core 2.0, .NET Framework 4.6.1
Page Size Range	1-128 (in multiples of 512 bytes)
Physical Page Size	pageSize × 512 bytes (512 bytes to 64 KB)
Max Tree Height	Depends on fanout (typically 3-5 levels for millions of records)
Thread Safety	Yes (with bLock=true)
Max Records	Limited by PAT size and max_recs parameter
File Format	Binary (Header + PAT + Pages)
Durability Model	Write-ahead PAT, committed after Commit()

Performance Benchmarks

Hardware: Intel i9-13900, Samsung 990 PRO NVMe, Linux, .NET 9.0

Operation	Mode	Throughput
Insert 3M records	FileStream (cache ON)	~3,157,895 ops/sec
Insert 3M records	FileStream (cache OFF)	~418,410 ops/sec
BulkLoad 3M records	FileStream (cache ON)	2-5M ops/sec
Find (cached)	Memory hit	~10M ops/sec
Range scan 100K	Sequential leaf traversal	~500K records/sec