# HIGH-PERFORMANCE STORAGE WITH ROCKSDB

## The Definitive Guide to Building High-Performance Embedded Storage Systems

- Maximize throughput and minimize latency
- Deep dive into LSM tree and compaction
- Design, tune, and optimize for real-world workloads
- Practical patterns and best practices

MATTHEW WILLIAMS

# High-Performance Storage with RocksDB

## The Definitive Guide to Building High-Performance Embedded Storage Systems

Steve Publications

This book is available at
https://leanpub.com/high-performancestoragewithrocksdb

This version was published on 2026-08-03

# Contents

# The Definitive Guide to Building High-Performance Embedded Storage Systems

RocksDB is one of the most widely used storage engines in modern infrastructure, powering everything from distributed SQL databases to streaming state backends. This book takes you from first principles to production mastery: how LSM-trees work, how RocksDB implements them, how to configure it for your workload, and how to keep it running reliably at scale. You will learn the architecture deeply enough to diagnose performance problems, tune compaction strategies, manage memory budgets, design key spaces, implement custom merge operators, and integrate RocksDB into real systems. Every chapter includes working code examples and practical guidance drawn from both the official documentation and production deployments across the industry.

# Introduction

In 2012, engineers at Facebook needed a storage engine that could handle millions of operations per second on flash storage without the latency overhead of network calls or the lock contention of traditional databases. They took Google's LevelDB as a starting point and built something new: RocksDB. Within months it was managing nearly a petabyte of data across Facebook's infrastructure. Today, more than a decade later, RocksDB is embedded in some of the most critical systems on the internet: distributed SQL databases like CockroachDB and TiKV, stream processing frameworks like Apache Flink and Kafka Streams, storage clusters like Ceph, and countless internal services at companies including Meta, LinkedIn, Yahoo, Uber, and Netflix.

Why has RocksDB become so ubiquitous? The answer lies in a combination of architectural choices that match the realities of modern hardware. Flash storage excels at sequential writes but struggles with random updates. Multi-core CPUs demand workloads that can parallelize. RocksDB embraces these constraints through its log-structured merge-tree design, turning random writes into sequential appends and spreading compaction work across cores. The result is a storage engine optimized for write-heavy workloads on SSDs, with the flexibility to tune itself toward reads when needed.

But RocksDB is not a simple drop-in replacement for a traditional database. It does not speak SQL. It does not provide distributed consistency out of the box. It is an embedded library that lives inside your process, giving you raw access to a sorted key-value store with extensive knobs for controlling every aspect of its behavior. This power comes at a cost: configuring RocksDB correctly requires understanding how it works internally. Set the wrong compaction strategy and your SSDs wear out faster. Misconfigure memory limits and your application runs out of RAM. Ignore bloom filters and read latency explodes.

This book exists to help you navigate that complexity. It is organized as a journey from fundamentals to mastery, with each chapter building on the previous ones so you develop a coherent mental model rather than a scattered collection of tips. We begin by explaining what RocksDB is and why its architecture matters. Then we walk through installation and basic

API usage so you can start writing code immediately. From there we dive deep into the internals: memtables, SST files, WAL, compaction, caching, bloom filters, compression. We cover advanced features like column families, transactions, merge operators, and iterators. Finally we address production concerns: tuning, monitoring, backup, security, and real-world use cases from companies that run RocksDB at scale.

By the end of this book you will be able to answer questions like these:

- Should I use leveled compaction or universal compaction for my workload?
- How much memory should I allocate to the block cache versus memtables?
- What bloom filter configuration minimizes read amplification without wasting RAM?
- How do I design a key space that supports efficient range scans?
- When should I use column families versus separate database instances?
- How do I detect and diagnose write stalls or compaction bottlenecks?
- What is the right backup strategy for my reliability requirements?

These are not theoretical questions. Engineers running RocksDB in production face them every day, and getting them wrong has real consequences: degraded performance, wasted infrastructure, data loss. This book gives you the knowledge to get them right.

The code examples throughout use modern C++ because that is RocksDB's native interface and where you get the most control. We also cover language bindings for Go, Java, Python, and Rust so you can apply these concepts regardless of your stack. Every example is designed to compile and run without modification, with explanations of significant lines so you understand not just what the code does but why it is written that way.

Let's get started.

# Chapter 1: What Is RocksDB and Why It Matters

## The Key-Value Store That Powers Modern Infrastructure

RocksDB is an embeddable persistent key-value store written in C++. It stores arbitrary byte arrays as keys and values, sorted lexicographically by key unless you provide a custom comparator. You open it with a directory path, write to it with Put operations, read from it with Get operations, and iterate over ranges with iterators. That is the surface API. The complexity and power lie underneath.

What makes RocksDB different from other key-value stores is that it is embedded rather than client-server. There is no separate RocksDB process to manage. Instead you link the RocksDB library into your application binary, and the database lives in the same address space. This design eliminates network latency between your application and its storage layer. At Facebook, where RocksDB originated, this was a critical requirement: their services needed sub-microsecond access to data for tasks like spam detection and graph search, and even a fast local TCP round trip would have been too slow.

The trade-off for embeddability is that you are responsible for everything RocksDB does not provide. There is no built-in replication. No distributed consensus. No SQL query layer. If you need those capabilities you build them on top of RocksDB, which is exactly what projects like CockroachDB, TiKV, and YugabyteDB have done. This makes RocksDB a building block rather than a complete solution, which is precisely why it has been so widely adopted: it gives infrastructure builders the raw storage performance they need while leaving higher-level design decisions to them.

## From LevelDB to RocksDB: A History of Evolution

To understand RocksDB you need to understand what came before it. In 2011 Google open-sourced LevelDB, a key-value store developed by Sanjay

Ghemawat and Jeff Dean that used a log-structured merge-tree architecture. LevelDB was designed for Chrome's IndexedDB implementation and ran as an embedded library within the browser process. It was fast and simple but had limitations that made it unsuitable for server workloads:

- Single-threaded compaction could not keep up with high write rates on modern hardware
- No support for multiple column families or flexible data partitioning
- Limited configurability for different workload patterns
- Write stalls under sustained load as background operations fell behind

Facebook's database engineering team evaluated LevelDB in 2012 and found it promising but insufficient for their needs. They forked it and began building RocksDB with a focus on server-grade performance on flash storage. The early development was aggressive: within months the codebase had diverged significantly from LevelDB, adding multi-threaded compaction, pluggable memtable implementations, column families, merge operators, and extensive tuning knobs. By November 2013, when RocksDB was open-sourced, most of the code was original rather than inherited from LevelDB.

The name "RocksDB" reflects its design goal: a database that could fully utilize the IOPS offered by rock-solid state drives. Facebook's internal benchmarks showed it performing up to ten times faster than LevelDB for pure random write workloads and thirty percent faster for reads. These numbers were not just academic: RocksDB quickly became critical infrastructure, managing nearly a petabyte of data across services including Newsfeed, ZippyDB (a distributed key-value store), Dragon (a graph query engine), and LogDevice (a distributed log storage system).

The evolution did not stop there. Over the following decade RocksDB accumulated features that addressed real production needs: universal compaction for write-heavy workloads, FIFO compaction for cache-like use cases, optimistic and pessimistic transactions, backup and checkpoint APIs, encryption at rest support, ribbon filters as an alternative to bloom filters, blob storage for large values, and dynamic level sizing. Each addition was driven by a concrete problem encountered in production rather than theoretical considerations.

## LSM-Trees: The Core Innovation Behind Performance

The defining architectural choice of RocksDB is its use of a log-structured merge-tree, or LSM-tree. This data structure, originally described by Patrick O'Neil et al. in 1996, is fundamentally different from the B-tree that powers most traditional databases like MySQL and PostgreSQL. Understanding this difference is essential to understanding when and why to use RocksDB.

A B-tree keeps data sorted on disk in a balanced tree structure. When you insert or update a record, the B-tree locates the correct leaf page, modifies it in place, and potentially splits pages to maintain balance. This design gives excellent random read performance: finding a key requires logarithmic time with very few disk seeks. But writes are expensive because they involve random I/O: locating the page, reading it into memory, modifying it, and writing it back. On spinning disks this is devastatingly slow. On SSDs it is better but still suboptimal because SSDs wear out with excessive program-erase cycles on individual blocks.

An LSM-tree takes a different approach. It never updates data in place. Instead every write is an append: new key-value pairs are added to the end of an in-memory buffer called a memtable. When the memtable fills up, it is flushed to disk as an immutable sorted file called an SST (sorted string table). Over time many SST files accumulate, potentially containing multiple versions of the same key if a key was updated or deleted after an earlier version was flushed. A background process called compaction periodically merges these files together, removing duplicate and obsolete entries so that reads do not need to check too many files.

This design has profound implications for performance characteristics:

- **Writes are fast**: Appending to a memtable is essentially a linked-list insertion with O(log n) complexity using a skip list. Flushing to disk is a sequential write, which SSDs handle efficiently.
- **Reads can be slower**: To find a key you must check the active memtable, any immutable memtables waiting to flush, all L0 SST files (which may overlap), and then lower levels sequentially until you find the key or determine it does not exist.
- **Space is reclaimed over time**: Compaction removes old versions of keys and deleted entries, but there is always some "space amplification" because multiple files may temporarily contain overlapping data.

- **SSD wear is reduced**: Sequential writes distribute wear evenly across flash blocks rather than concentrating on hot pages.

The LSM-tree is not universally superior to the B-tree. For read-heavy workloads with mostly point lookups, a well-tuned B-tree can outperform an LSM-tree because it avoids the overhead of checking multiple levels. But for write-intensive workloads, especially those involving time-series data, event logs, or frequent updates, the LSM-tree's sequential write pattern is hard to beat. This is why RocksDB has become the storage engine of choice for systems that ingest large volumes of data: distributed databases, stream processors, blockchain nodes, monitoring platforms, and caching layers.

## Where RocksDB Lives Today: Ecosystem and Adoption

RocksDB's influence extends far beyond Facebook. It has become the de facto embedded storage engine for systems that need high write throughput on commodity hardware. Here is where you will find it in production:

**Distributed databases**: CockroachDB uses RocksDB as its per-node storage engine (though it has been migrating to a Go-based fork called Pebble). TiKV, the distributed key-value store underlying TiDB, uses RocksDB for both user data and Raft logs. YugabyteDB's DocDB layer customizes RocksDB for document-oriented storage with MVCC support.

**Stream processing**: Apache Flink uses RocksDB as its state backend for maintaining operator state across failures. Kafka Streams embeds RocksDB to persist state stores for joins, aggregations, and windowing operations. Both frameworks rely on RocksDB's ability to handle continuous writes while supporting point lookups and range scans.

**Storage systems**: Ceph's BlueStore backend uses RocksDB to manage metadata including object mappings and placement group logs, typically on a dedicated SSD separate from the primary data devices. Alluxio uses RocksDB for file system metadata at scale, managing billions of files.

**Blockchain and ledgers**: Multiple blockchain projects use RocksDB as their state database because it handles the continuous write stream of new blocks efficiently while supporting fast lookups by account or contract address.

**Internal services**: LinkedIn uses RocksDB in its FollowFeed service for real-time content delivery. Yahoo deployed it in Sherpa, a distributed data store.

Pinterest uses it for caching and data storage. Companies like Uber, Netflix, Airbnb, and Twitter have all used RocksDB in production systems, often with custom configurations tuned to their specific workloads.

The ecosystem around RocksDB includes official language bindings for Java (RocksJava), Python, Perl, and Node.js, plus community-maintained bindings for Go (gorocksdb), Rust (rust-rocksdb), Ruby, PHP, Haskell, and others. There are also commercial distributions and forks: MyRocks (MySQL on RocksDB by Percona and MariaDB), MongoRocks (MongoDB on RocksDB), Speedb (a performance-optimized fork by Stream), and Pebble (CockroachDB's Go-based successor).

## When to Choose RocksDB (and When Not To)

RocksDB is powerful but not universally appropriate. Choosing it requires understanding both its strengths and its limitations.

**Choose RocksDB when:**

- You need high write throughput on SSDs, especially for time-series data, event logs, or continuous ingestion workloads
- You want an embedded storage engine that eliminates network latency to your database
- Your workload is write-heavy or has bursty write patterns that would cause lock contention in a B-tree database
- You are building a distributed system and need a reliable per-node storage layer
- You need fine-grained control over storage behavior: compaction strategies, compression algorithms, memory budgets, bloom filter policies
- You want to build custom data structures on top of a sorted key-value store (secondary indexes, caches, queues, state machines)

**Do not choose RocksDB when:**

- You need SQL queries with joins and complex filtering without building that layer yourself
- You need distributed consistency, replication, or sharding out of the box
- Your workload is read-heavy with predominantly point lookups on relatively static data (a B-tree database like SQLite may be simpler and faster)

- You have very limited memory: RocksDB needs RAM for memtables and block caches to perform well
- You need a simple embedded database for a small application: SQLite's maturity, portability, and zero-configuration operation make it a better choice
- Your data fits entirely in memory: an in-memory data structure will be faster and simpler

RocksDB is a tool for engineers who need raw storage performance and are willing to invest time in understanding and tuning it. It rewards deep knowledge with exceptional performance but punishes ignorance with confusing behavior, wasted resources, or data loss. The rest of this book is devoted to giving you that knowledge.

# Chapter 2: Getting Started with RocksDB

## Building and Installing RocksDB from Source

RocksDB is distributed as source code on GitHub at https://github.com/facebook/rocksdb. There are no official binary packages for most platforms, so building from source is the standard installation method. The project supports both Makefile-based builds (primary method for Unix-like systems) and CMake (useful for Windows and cross-platform projects).

**System requirements:**

- GCC >= 11 or Clang >= 10 with C++20 support
- Standard build tools: make, git, cmake (optional but recommended)
- Optional compression libraries: snappy, lz4, zlib, zstd, bzip2
- For command-line tools: gflags >= 2.2.0

On Ubuntu or Debian-based systems, install dependencies with:

```
sudo apt-get update
sudo apt-get install -y build-essential git cmake \
    libsnappy-dev zlib1g-dev libbz2-dev liblz4-dev \
    libzstd-dev libgflags-dev libgoogle-perftools-dev
```

On macOS with Homebrew:

```
xcode-select --install
brew install rocksdb
```

This installs RocksDB system-wide, but for production use you typically want to build from source with specific options. Here is the standard process:

```
git clone https://github.com/facebook/rocksdb.git
cd rocksdb

# Build static library in release mode (production)
make static_lib -j$(nproc)

# Or build shared library if you prefer dynamic linking
make shared_lib -j$(nproc)

# Install to a prefix directory
mkdir -p /opt/rocksdb
sudo cp -r include /opt/rocksdb/
sudo cp librocksdb.a /opt/rocksdb/lib/ 2>/dev/null || true
sudo cp librocksdb.so* /opt/rocksdb/lib/ 2>/dev/null || true
```

Critical note: do not use `make all` or `make check` for production builds. These compile RocksDB in debug mode with extensive assertions enabled, which dramatically reduces performance. Always use `make static_lib` or `make shared_lib` for release builds.

**CMake build (alternative):**

```
cd rocksdb
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release \
      -DWITH_SNAPPY=ON \
      -DWITH_LZ4=ON \
      -DWITH_ZSTD=ON \
      -DWITH_GFLAGS=ON \
      ..
make -j$(nproc)
make install  # Installs to /usr/local by default
```

The CMake approach is particularly useful on Windows where you would use Visual Studio:

```
cmake -G "Visual Studio 17 2022" -A x64 ^
      -DCMAKE_BUILD_TYPE=Release ^
      -DWITH_SNAPPY=ON ^
      ..
msbuild rocksdb.sln /p:Configuration=Release /m
```

**Linking your application:**

When compiling code that uses RocksDB, link against the library and its dependencies. For a static build:

```
g++ -std=c++20 -O2 -I/opt/rocksdb/include myapp.cpp \
    -L/opt/rocksdb/lib -lrocksdb -lsnappy -lz -lbz2 -llz4 -lzstd \
    -pthread -o myapp
```

For a shared build, you may also need to set `LD_LIBRARY_PATH` at runtime:

```
export LD_LIBRARY_PATH=/opt/rocksdb/lib:$LD_LIBRARY_PATH
./myapp
```

## Your First Database: Opening, Writing, Reading, Closing

Let us write a minimal RocksDB application that demonstrates the basic lifecycle. This example opens a database (creating it if necessary), writes a key-value pair, reads it back, and closes cleanly.

```
#include <cassert>
#include <iostream>
#include <string>
#include "rocksdb/db.h"
#include "rocksdb/options.h"
#include "rocksdb/status.h"

int main() {
    // Define the directory where RocksDB will store its files
    std::string db_path = "/tmp/my_rocksdb";

    // Configure database options
    rocksdb::Options options;
    options.create_if_missing = true;  // Create DB directory if it does not
    ↪  exist
    options.error_if_exists = false;   // Do not fail if DB already exists

    // Pointer to hold the database handle
    rocksdb::DB* db = nullptr;

    // Open the database
    rocksdb::Status status = rocksdb::DB::Open(options, db_path, &db);
    assert(status.ok());  // In production, handle errors properly instead of
    ↪  asserting
    std::cout << "Database opened successfully at: " << db_path << std::endl;

    // Write a key-value pair using Put
    status = db->Put(rocksdb::WriteOptions(), "hello", "world");
    assert(status.ok());
```

```
    std::cout << "Wrote key 'hello' with value 'world'" << std::endl;

    // Read the value back using Get
    std::string value;
    status = db->Get(rocksdb::ReadOptions(), "hello", &value);
    if (status.ok()) {
        std::cout << "Read value for 'hello': " << value << std::endl;
        assert(value == "world");
    } else if (status.IsNotFound()) {
        std::cerr << "Key 'hello' not found" << std::endl;
    } else {
        std::cerr << "Error reading key: " << status.ToString() << std::endl;
    }

    // Try reading a non-existent key
    status = db->Get(rocksdb::ReadOptions(), "nonexistent", &value);
    if (status.IsNotFound()) {
        std::cout << "Key 'nonexistent' correctly reported as not found" <<
        ↪  std::endl;
    }

    // Delete the key
    status = db->Delete(rocksdb::WriteOptions(), "hello");
    assert(status.ok());
    std::cout << "Deleted key 'hello'" << std::endl;

    // Verify deletion
    status = db->Get(rocksdb::ReadOptions(), "hello", &value);
    assert(status.IsNotFound());
    std::cout << "Confirmed key 'hello' is deleted" << std::endl;

    // Close the database and release resources
    delete db;
    std::cout << "Database closed successfully" << std::endl;

    return 0;
}
```

Let us walk through the significant parts of this code:

- `rocksdb::Options` is the configuration object that controls how the database behaves. Setting `create_if_missing = true` tells RocksDB to create the directory and initialize the database files if they do not exist. Without this, opening a non-existent path returns an error.
- `rocksdb::DB::Open()` takes the options, the path, and a pointer to a `DB*` that will be set on success. It returns a `rocksdb::Status` object indicating success or failure. Always check status in production code.

- `db->Put()` writes a key-value pair. The first argument is a `WriteOptions` object that controls write behavior (we use defaults here). Keys and values are passed as byte ranges; C++ std::string works because RocksDB's Slice type can construct from it.
- `db->Get()` reads a value by key. If the key exists, it writes the value into the provided string buffer and returns OK. If the key does not exist, it returns a NotFound status rather than an empty string, so you must check the status to distinguish between "key exists with empty value" and "key does not exist."
- `db->Delete()` removes a key. Like Put, it takes WriteOptions. Deletion in RocksDB is implemented as a tombstone marker, not immediate physical removal; compaction eventually purges the deleted entry from disk files.
- The database must be explicitly closed by calling `delete db`. This flushes any remaining data and releases file handles. Failing to close properly can corrupt the database or leak resources.

Compile this example:

```
g++ -std=c++20 -O2 -I/opt/rocksdb/include first_db.cpp \
    -L/opt/rocksdb/lib -lrocksdb -lsnappy -lz -lbz2 -llz4 -lzstd \
    -pthread -o first_db
./first_db
```

Expected output:

```
Database opened successfully at: /tmp/my_rocksdb
Wrote key 'hello' with value 'world'
Read value for 'hello': world
Key 'nonexistent' correctly reported as not found
Deleted key 'hello'
Confirmed key 'hello' is deleted
Database closed successfully
```

After running, you will find several files in `/tmp/my_rocksdb`: CURRENT, LOCK, MANIFEST, OPTIONS, log files, and potentially SST files if enough data was written to trigger a flush. These are RocksDB's internal files; do not modify them manually.

## Understanding the Basic C++ API Surface

RocksDB's C++ API is organized around several core types defined in the `include/rocksdb/` directory. Here is the essential surface you need to know:

**rocksdb::DB**: The main database handle. All operations go through this object. Key methods include:

- `Open()`, `Close()` (via delete), `DestroyDB()`: lifecycle management
- `Put()`, `Get()`, `Delete()`, `Merge()`: basic data operations
- `NewIterator()`: create an iterator for range scans
- `Write()`: write a batch of operations atomically
- `GetProperty()`: query runtime statistics and state

**rocksdb::Options**: Configuration for opening a database. This is the master options struct that combines global settings (`DBOptions`) and per-column-family settings (`ColumnFamilyOptions`). Key fields include:

- `create_if_missing`, `error_if_missing`, `paranoid_checks`: open behavior
- `max_open_files`: file descriptor limit (-1 for unlimited)
- `block_cache`: cache for uncompressed data blocks
- `compression`: default compression algorithm
- `write_buffer_size`: size of each memtable (default 64 MB)

**rocksdb::ReadOptions**: Controls read behavior. Important fields:

- `verify_checksums`: validate data integrity on reads (default true)
- `fill_cache`: whether to cache blocks in block cache (default true)
- `snapshot`: read from a specific point-in-time snapshot

**rocksdb::WriteOptions**: Controls write behavior. Important fields:

- `sync`: if true, fsync after each write for durability at cost of performance (default false)
- `disableWAL`: skip writing to WAL for faster but less durable writes
- `no_slowdown`: allow writes even when flush/compaction is falling behind

**rocksdb::Status**: Represents success or failure. Methods:

- `ok()`: true if operation succeeded
- `IsNotFound()`, `IsCorruption()`, `IsIOError()`, etc.: specific error types
- `ToString()`: human-readable error message

**rocksdb::Slice**: A lightweight reference to a byte range, similar to Go's slice or Java's ByteString. It does not own the data it points to, so be careful about lifetimes: do not create a Slice pointing to stack memory that will go out of scope.

Here is a slightly more realistic example showing proper error handling and resource management:

```
#include <iostream>
#include <memory>
#include "rocksdb/db.h"
#include "rocksdb/options.h"
#include "rocksdb/status.h"

int main() {
    std::string db_path = "/tmp/my_rocksdb_safe";

    // Configure options with some production-minded defaults
    rocksdb::Options options;
    options.create_if_missing = true;
    options.max_open_files = -1;  // Use as many file descriptors as needed
    options.compression = rocksdb::CompressionType::kLZ4Compression;

    std::unique_ptr<rocksdb::DB> db;
    rocksdb::Status status = rocksdb::DB::Open(options, db_path, &db);
    if (!status.ok()) {
        std::cerr << "Failed to open database: " << status.ToString() <<
        ↪  std::endl;
        return 1;
    }

    // Write multiple key-value pairs
    const char* keys[] = {"user:1:name", "user:1:email", "user:2:name",
    ↪  "user:2:email"};
    const char* values[] = {"Alice", "alice@example.com", "Bob",
    ↪  "bob@example.com"};

    for (int i = 0; i < 4; i++) {
        status = db->Put(rocksdb::WriteOptions(), keys[i], values[i]);
        if (!status.ok()) {
```

```
            std::cerr << "Failed to write key '" << keys[i] << "': "
                      << status.ToString() << std::endl;
            return 1;
        }
    }

    // Read and display all keys
    for (int i = 0; i < 4; i++) {
        std::string value;
        status = db->Get(rocksdb::ReadOptions(), keys[i], &value);
        if (status.ok()) {
            std::cout << keys[i] << " = " << value << std::endl;
        } else if (status.IsNotFound()) {
            std::cout << keys[i] << " not found" << std::endl;
        } else {
            std::cerr << "Error reading '" << keys[i] << "': "
                      << status.ToString() << std::endl;
        }
    }

    // unique_ptr automatically deletes db when it goes out of scope
    return 0;
}
```

This version uses `std::unique_ptr<rocksdb::DB>` for automatic cleanup, checks every status return value, and demonstrates a common key naming pattern (namespace🔑field) that we will explore further in Chapter 4.

## Integrating RocksDB into Modern C++ Projects

In real projects you need to integrate RocksDB with your build system and application architecture cleanly. Here are patterns that work well in practice.

**CMakeLists.txt example:**

```
cmake_minimum_required(VERSION 3.20)
project(my_rocksdb_app VERSION 1.0 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Find RocksDB (if installed via package manager or CMake install)
find_package(rocksdb REQUIRED)

# Or if using a local build, specify paths directly:
# set(ROCKSDB_INCLUDE_DIR "/opt/rocksdb/include")
# set(ROCKSDB_LIBRARY "/opt/rocksdb/lib/librocksdb.a")

add_executable(my_app src/main.cpp src/database.cpp)

target_include_directories(my_app PRIVATE ${ROCKSDB_INCLUDE_DIRS})
target_link_libraries(my_app PRIVATE ${ROCKSDB_LIBRARIES} pthread)
```

**Encapsulating RocksDB in a class:**

For larger applications, wrap RocksDB operations in a well-defined interface rather than scattering raw calls throughout your codebase:

```
#include <memory>
#include <string>
#include <optional>
#include "rocksdb/db.h"
#include "rocksdb/options.h"

class KeyValueStore {
public:
    explicit KeyValueStore(const std::string& path)
        : path_(path), db_(nullptr) {
        rocksdb::Options options;
        options.create_if_missing = true;
        options.compression = rocksdb::CompressionType::kLZ4Compression;
        options.max_open_files = -1;

        auto status = rocksdb::DB::Open(options, path_, &db_);
        if (!status.ok()) {
            throw std::runtime_status("Failed to open RocksDB: " +
            ↪  status.ToString());
        }
    }

    ~KeyValueStore() {
        if (db_) {
            delete db_;
        }
```

```
    }

    // Non-copyable, movable
    KeyValueStore(const KeyValueStore&) = delete;
    KeyValueStore& operator=(const KeyValueStore&) = delete;
    KeyValueStore(KeyValueStore&& other) noexcept
        : path_(std::move(other.path_)), db_(other.db_) {
        other.db_ = nullptr;
    }

    bool Put(const std::string& key, const std::string& value) {
        auto status = db_->Put(rocksdb::WriteOptions(), key, value);
        return status.ok();
    }

    std::optional<std::string> Get(const std::string& key) {
        std::string value;
        auto status = db_->Get(rocksdb::ReadOptions(), key, &value);
        if (status.ok()) {
            return value;
        }
        return std::nullopt;
    }

    bool Delete(const std::string& key) {
        auto status = db_->Delete(rocksdb::WriteOptions(), key);
        return status.ok();
    }

private:
    std::string path_;
    rocksdb::DB* db_;
};
```

This wrapper provides a clean API using modern C++ idioms (std::optional for nullable returns), handles the database lifecycle automatically, and prevents accidental copies that would double-delete the DB handle. It also centralizes configuration so you can tune options in one place.

## Quick Start with Alternative Language Bindings

While C++ gives you the most control and best performance, many teams prefer to use RocksDB from their primary language. Here are brief examples for popular bindings.

**Go (gorocksdb):**

```go
package main

import (
    "fmt"
    "github.com/tecbot/gorocksdb"
)

func main() {
    opts := gorocksdb.NewDefaultOptions()
    opts.SetCreateIfMissing(true)

    db, err := gorocksdb.OpenDb(opts, "/tmp/go_rocksdb")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    writeOpts := gorocksdb.NewDefaultWriteOptions()
    readOpts := gorocksdb.NewDefaultReadOptions()

    err = db.Put(writeOpts, []byte("hello"), []byte("world"))
    if err != nil {
        panic(err)
    }

    value, err := db.Get(readOpts, []byte("hello"))
    if err != nil {
        panic(err)
    }
    defer value.Free()
    fmt.Println(string(value.Data()))
}
```

### Java (RocksJava - official binding):

```java
import org.rocksdb.*;

public class RocksDBExample {
    static {
        RocksDB.loadLibrary();
    }

    public static void main(String[] args) throws Exception {
        Options options = new Options()
            .setCreateIfMissing(true);

        try (RocksDB db = RocksDB.open(options, "/tmp/java_rocksdb")) {
            db.put("hello".getBytes(), "world".getBytes());
            byte[] value = db.get("hello".getBytes());
```

```
            System.out.println(new String(value));
        }
    }
}
```

**Rust (rust-rocksdb):**

```rust
use rocksdb::{DB, Options};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let mut opts = Options::default();
    opts.create_if_missing(true);

    let db = DB::open(&opts, "/tmp/rust_rocksdb")?;
    db.put(b"hello", b"world")?;

    if let Some(value) = db.get(b"hello")? {
        println!("{}", String::from_utf8_lossy(&value));
    }

    Ok(())
}
```

**Python (rocksdb3, based on rust-rocksdb):**

```python
import rocksdb

db = rocksdb.DB("/tmp/python_rocksdb", rocksdb.Options(create_if_missing=True))
db.put(b"hello", b"world")
print(db.get(b"hello"))  # b'world'
db.close()
```

These bindings generally wrap the same underlying C++ library, so performance characteristics are similar. The main differences are in ergonomics, feature completeness (some bindings lag behind the core API), and overhead from foreign function calls. For latency-critical paths, native C++ remains the best choice.

Language binding status as of this writing:

- Java (RocksJava): Officially maintained by Meta, closely tracks core features
- Go (gorocksdb): Community-maintained, widely used in production

- Rust (rust-rocksdb): Actively maintained by PingCAP, production-grade
- Python: Multiple options exist; python-rocksdb is unmaintained, rocksdb3 (via PyO3) is the modern choice

Always check the binding's repository for the latest status and compatibility information before committing to it in production.

# Chapter 3: Core Architecture and Data Flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## The Write Path: From Application Call to Disk

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## MemTables and Immutable MemTables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## SST Files and the Level Hierarchy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## The Read Path: Lookup, Bloom Filters, and Caching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Write-Ahead Log and Crash Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Chapter 4: Data Models and Column Families

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Keys, Values, and Encoding Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Column Families: Isolation, Configuration, and Use Cases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Designing Key Spaces for Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Prefixes, Ranges, and Secondary Index Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Versioning and Multi-Tenancy with Column Families

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Chapter 5: Advanced Write Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Write Batches: Atomicity and Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Merge Operators: Semantics and Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Conditional Writes and Compare-and-Swap Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Single-Delete vs Delete: Handling Tombstones Correctly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Optimizing Write Throughput in Production Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Chapter 6: Reading Data: Iterators, Snapshots, and Transactions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Point Lookups and Read Options

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Iterators: Range Scans, Prefix Seeks, and Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Snapshots: Consistent Views Without Locking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Optimistic Transactions in RocksDB

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Isolation Levels and Concurrency Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Chapter 7: Compaction: The Heart of RocksDB

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Why Compaction Exists: Trade-offs Explained

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Level-Based Compaction (LEVEL)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Universal Compaction: FIFO, Size-Tiered, and Windowed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## FIFO Compaction for Log-Like Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Choosing and Tuning Compaction for Your Workload

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Chapter 8: Memory Management and Caching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## The Cache Hierarchy: Block Cache, Table Cache, and More

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## MemTable Memory Budgeting and Auto-Tuning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Shared vs Per-Instance Caches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Subcompaction and Parallelism Controls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Diagnosing and Fixing Memory Pressure Issues

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Chapter 9: Compression, Bloom Filters, and SST File Format

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## The SST File Format: Anatomy of a Table

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Compression Algorithms: Choosing the Right One

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Bloom Filters: Standard, Partitioned, and Full

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Prefix Bloom Filters for Range Queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Tuning Block Size and Filter Policies Together

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Chapter 10: Configuration and Options Reference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## DatabaseOptions: Global Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## ColumnFamilyOptions: Per-Column Tuning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Env and IO Control: Managing System Resources

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Runtime Options vs Build-Time Options

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## A Practical Configuration Checklist for Production

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Chapter 11: Performance Tuning and Benchmarking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Understanding RocksDB Benchmarks and Metrics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Using the Built-in rocksdbtool and ldb Utilities

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Profiling Hot Paths: CPU, IO, and Lock Contention

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Tuning for Read-Heavy Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Tuning for Write-Heavy and Mixed Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Chapter 12: Monitoring, Debugging, and Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Runtime Stats, Histograms, and Property API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Integrating with Prometheus and Grafana

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Reading and Interpreting ROCKSDB_LOG_LEVEL Output

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Debugging Slow Queries and Stalls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Handling Corruption: Detection and Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Chapter 13: Backup, Restore, and Checkpoints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Full Backups with BackupEngine

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Incremental Backups and Log Shipping Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Checkpoints for Instant Snapshots and Migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Point-in-Time Recovery Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Cross-Machine Restore and Version Compatibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Chapter 14: Concurrency, Replication, and Distributed Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Concurrent Access Patterns and Thread Safety

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Using Multiple Instances Safely on One Machine

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Replication Patterns Built on Top of RocksDB

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Integrating with Raft, Paxos, and Distributed Consensus

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## RocksDB in Microservices and Cloud-Native Architectures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Chapter 15: Real-World Production Use Cases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Time-Series Data and Metrics Storage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Blockchain and Ledger Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Caching Layers and Edge Databases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## IoT Device Storage and Embedded Analytics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Lessons Learned from Production Deployments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Chapter 16: Comparisons, Alternatives, and Migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## RocksDB vs LevelDB: What Changed and Why It Matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## RocksDB vs WiredTiger: MongoDB's Choice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## RocksDB vs SQLite: Different Problems, Different Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## RocksDB in the LSM Family: Cassandra, ScyllaDB, Badger

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Migrating to RocksDB: Strategies and Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Chapter 17: Security, Limitations, and Future Directions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Security Considerations: Encryption, Access Control, Sandboxing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Known Limitations and Anti-Patterns to Avoid

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## The RocksDB Roadmap and Upcoming Features

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## When to Look Beyond RocksDB

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

## Final Recommendations for Production Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.

# References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancestoragewithrocksdb.