Building Databases with BASON

A Graduate Course in Database Internals

Overview

This course builds four database storage engines from shared components, all unified by a single binary encoding format: BASON (Binary Adaptation of Structured Object Notation). Students implement progressively more complex systems, reusing code from earlier assignments as building blocks for later ones.

The four target systems are:

- **BASONCask** — an append-only log-structured store (inspired by Bitcask)
- **BASONLevel** — an LSM-tree store with leveled compaction (inspired by LevelDB/RocksDB)
- **BASONLite** — a page-based B-tree store with transactions (inspired by SQLite)
- **BASONRedis** — an in-memory data structure server with persistence (inspired by Redis)

Each system is a different answer to the same question: how do you durably store and retrieve BASON records? The architectural differences between them illuminate the fundamental tradeoffs in database design: write amplification vs read amplification, memory vs disk, simplicity vs throughput, random access vs sequential access.

Prerequisite Knowledge

Students should be comfortable with C++ (including move semantics, RAII, and smart pointers), basic data structures (hash maps, balanced trees, skip lists), file I/O and system calls (open, read, write, fsync), and concurrency primitives (mutexes, condition variables). Familiarity with existing database systems at the user level is assumed; this course teaches the internals.

Language

All assignments should be implemented in C++ (C++17 or later). Students may use a different language if they can demonstrate binary compatibility with the BASON format specification and interoperability with the provided test harnesses. Regardless of language choice, all on-disk formats must conform exactly to the BASON specification; a BASON file produced by any student's implementation must be readable by any other.

Format Specification

All assignments reference the BASON format specification (see companion document: *BASON: Binary Adaptation of Structured Object Notation, Version 0.1*). Students should read the full specification before beginning Assignment 1. Key sections are referenced by number throughout.

Guiding Principle

The interface between every component is BASON. The codec speaks BASON. The WAL stores BASON records. The SST files contain sorted BASON records. The memtable indexes BASON records. The wire protocol carries BASON records. There is no separate serialization layer, no internal format distinct from the external one. This is a deliberate design constraint: it forces architectural clarity and means every component can be tested, inspected, and debugged with the same tooling.

Assignment Structure

Assignments are numbered and ordered by dependency. Part I covers shared components (Assignments 1–4). Part II covers the four stores (Assignments 5–8). Each assignment specifies deliverables, a public API, correctness criteria, and performance targets. Assignments build on previous ones — the WAL uses the codec, the SST writer uses the codec, the stores use the WAL and SST layers.

```
Part I: Shared Components

  Assignment 1: BASON Codec
  Assignment 2: Write-Ahead Log
  Assignment 3: SST Files (BASON Tables)
  Assignment 4: Memtable and Merge Iterator

Part II: Storage Engines

  Assignment 5: BASONCask
  Assignment 6: BASONLevel
  Assignment 7: BASONLite
  Assignment 8: BASONRedis
```

Part I: Shared Components

Assignment 1: BASON Codec

Objective

Implement an encoder and decoder for the BASON binary format, a RON64 number codec, a path manipulation library, and a strictness validator. This is the foundation upon which every subsequent assignment builds.

Background

BASON is a TLV (Tag-Length-Value) encoding for JSON-compatible data. Each record begins with a single-byte tag drawn from the set `{B, A, S, O, N, b, a, s, o, n}`, where uppercase indicates long form and lowercase indicates short form. The tag identifies both the value type (Boolean, Array, String, Object, Number) and the length encoding scheme.

Refer to the BASON specification, Sections 2–4, for the complete encoding rules. Section 7 defines the strictness bitmask.

Deliverables

1.1. Record Encoder/Decoder

```cpp
// Core types
enum class BasonType { Boolean, Array, String, Object, Number };

struct BasonRecord {
    BasonType type;
    std::string key;
    std::string value;           // leaf types
    std::vector<BasonRecord> children; // container types (Array, Object)
};

// Encode a record to bytes. The encoder must choose short form when
// both key and value fit in 15 bytes, long form otherwise.
std::vector<uint8_t> bason_encode(const BasonRecord& record);

// Decode a record from bytes. Returns the record and the number of
// bytes consumed. Throws on malformed input.
std::pair<BasonRecord, size_t> bason_decode(
    const uint8_t* data, size_t len);

// Decode all records from a byte buffer (a stream of concatenated
// records).
std::vector<BasonRecord> bason_decode_all(
    const uint8_t* data, size_t len);
```

1.2. RON64 Codec

```cpp
// Encode a non-negative integer to a RON64 string (big-endian digit
// order, as specified in BASON specification Section 3).
std::string ron64_encode(uint64_t value);

// Decode a RON64 string to an integer. Throws on invalid characters.
uint64_t ron64_decode(const std::string& s);
```

1.3. Path Utilities

```cpp
// Join path segments with '/' separator.
std::string path_join(const std::vector<std::string>& segments);

// Split a path into segments.
std::vector<std::string> path_split(const std::string& path);

// Return the parent path ("a/b/c" → "a/b"). Empty string for
// top-level keys.
std::string path_parent(const std::string& path);

// Return the last segment ("a/b/c" → "c").
std::string path_basename(const std::string& path);
```

1.4. Strictness Validator

```cpp
// Validate a record against a strictness bitmask (BASON specification
// Section 7). Returns true if the record conforms to all enabled
// rules. For container types, validation is recursive.
bool bason_validate(const BasonRecord& record, uint16_t strictness);
```

1.5. JSON Round-trip

```cpp
// Convert JSON text to a BASON byte stream (nested mode).
std::vector<uint8_t> json_to_bason(const std::string& json);

// Convert a BASON byte stream to JSON text.
std::string bason_to_json(const uint8_t* data, size_t len);
```

1.6. Flat/Nested Conversion

```cpp
// Convert a nested BASON record into a sequence of flat path-keyed
// leaf records.
std::vector<BasonRecord> bason_flatten(const BasonRecord& root);

// Convert a sequence of flat path-keyed leaf records into a nested
// BASON record tree.
BasonRecord bason_unflatten(const std::vector<BasonRecord>& records);
```

1.7. Command-line Tool: `basondump`

Build a command-line utility that reads a file containing BASON records
and prints them in human-readable form. It should support:

- Hex dump with annotated fields (tag, lengths, key, value)
- JSON output mode
- Strictness validation against a specified bitmask
- Flat-to-nested and nested-to-flat conversion

This tool will be invaluable for debugging every subsequent assignment.

Correctness Criteria

- Round-trip: for any valid JSON input, `bason_to_json(json_to_bason(j))` must produce semantically equivalent JSON.
- Round-trip: for any valid BASON input at Standard strictness, `bason_encode(bason_decode(b))` must produce identical bytes.
- Short/long form: the encoder must produce the shortest valid encoding (short form when both key ≤ 15 bytes and value ≤ 15 bytes).
- Strictness: the validator must accept all records at Permissive and reject known-bad records at the appropriate strictness level.
- `bason_unflatten(bason_flatten(r))` must produce a record semantically equivalent to `r` for any nested record.

Performance Targets

- Encode/decode throughput: ≥ 500 MB/s on a single core for leaf records (measured on records with 8-byte keys and 64-byte values).
- This is a codec, not a storage engine — allocation and copying dominate. Students should profile and minimize copies.

Testing Guidance

Write a fuzz test that generates random JSON documents, converts to BASON and back, and checks round-trip equivalence. Run it for at least 10 million iterations. Separately, write property-based tests for RON64 (encode/decode round-trip, ordering preservation, minimal encoding).

Assignment 2: Write-Ahead Log

Objective

Implement an append-only, crash-safe write-ahead log that stores BASON records, using cumulative BLAKE3 hashing for integrity and file-offset-based addressing.

Background

The WAL is the durability backbone of every store in this course. It accepts BASON records, appends them to segment files, and provides crash recovery by replaying from the last valid hash checkpoint. Every record is identified by its cumulative byte offset from the beginning of the log (across all segment files), which serves as the log sequence number (LSN). There is no separate sequence counter.

There is only one mutation operation: merge. A record with a non-null value is an upsert. A Boolean record with an empty value (the BASON null encoding) is a tombstone. The WAL does not distinguish between inserts, updates, and deletes — the record itself carries the semantics.

WAL Record Format

```
+---------------+----------+
| BASON record  | padding  |
| TLV ...       | 0-7B     |
+---------------+----------+
```

Records are padded to 8-byte alignment. The padding bytes must be zero.

Hash Checkpoint Format

```
+---------------+--------------+
| magic 'H'     | BLAKE3 hash  |
| 1B           | 32B         |
+---------------+--------------+
```

The byte `0x48` ('H') is not a valid BASON tag, so it unambiguously marks a checkpoint. The BLAKE3 hash covers all bytes from the start of the segment file up to and including the `H` byte, but excluding the 32-byte digest itself.

A hash checkpoint is written at transaction boundaries. All records
between the previous checkpoint and this one are atomic — either all
are valid or all are discarded on recovery.

Segment File Format

```
Header (20 bytes):
+------------------+----------+---------+----------------+
| magic "BASONWAL" | version  | flags   | start_offset   |
| 8B               | 2B LE    | 2B LE   | 8B LE          |
+------------------+----------+---------+----------------+

Followed by:
  (record | hash_checkpoint)*
```

The `start_offset` field records the global byte offset of the first record in this segment.

Segment files are named by their start offset, zero-padded to 20
decimal digits:

```
wal/00000000000000000000.wal
wal/00000000000004194304.wal
```

Deliverables

2.1. WAL Writer

```
cpp
```

```cpp
class WalWriter {
public:
    // Open or create a WAL in the given directory.
    static WalWriter open(const std::string& dir);

    // Append a BASON record. Returns the global byte offset of the
    // record (its LSN). The record is buffered in memory until
    // checkpoint() or sync() is called.
    uint64_t append(const BasonRecord& record);

    // Write a BLAKE3 hash checkpoint. All records since the last
    // checkpoint become part of an atomic group.
    void checkpoint();

    // Flush buffers and fsync to disk.
    void sync();

    // Rotate to a new segment file. Called when the current segment
    // exceeds a size threshold.
    void rotate(uint64_t max_segment_size);
};
```

2.2. WAL Reader

```
cpp
```

```
class WalReader {
public:
    // Open an existing WAL directory for reading.
    static WalReader open(const std::string& dir);

    // Recover after a crash. Returns the offset just past the last
    // valid hash checkpoint. Records beyond this point are discarded
    // (the segment file is truncated).
    uint64_t recover();

    // Create an iterator starting from a given global offset. Used
    // for replay during recovery, replication, and reads.
    WalIterator scan(uint64_t from_offset);
};

class WalIterator {
public:
    bool valid() const;
    void next();
    uint64_t offset() const;      // global byte offset of current record
    const BasonRecord& record() const;
};
```

2.3. WAL Garbage Collection

```cpp
// Delete segment files whose highest record offset is below
// the given threshold. Called after a flush or compaction makes
// // the data in those segments redundant.
void wal_truncate_before(const std::string& dir, uint64_t offset);
```

Correctness Criteria

- **Durability**: after `checkpoint()` + `sync()` return, the records survive process crash, kernel panic, and power loss (assuming the storage device honors fsync).
- **Atomicity**: records between two hash checkpoints are all-or-nothing. A crash mid-transaction discards all records since the last checkpoint.
- **Ordering**: `recover()` returns records in the exact order they were appended.
- **Integrity**: any corruption (bit flip, truncation, partial write) within a segment is detected by hash checkpoint verification.

Crash Testing

This assignment requires a crash test harness. The recommended approach:

1. Fork a child process that performs writes to the WAL.
2. At random intervals, kill the child with SIGKILL.
3. In the parent, call `recover()` and verify that all recovered records are consistent (no partial records, no gaps, no corruption).
4. Repeat for at least 10,000 crash cycles.

Students must also test with `fsync` disabled to observe the difference between buffered and durable writes, and explain in their report why the results differ.

Performance Targets

- Sequential append throughput: ≥ 200 MB/s with periodic sync (e.g., every 1000 records).
- Recovery speed: ≥ 500 MB/s (dominated by BLAKE3 hashing speed and sequential read I/O).

Assignment 3: SST Files (BASON Tables)

Objective

Implement a sorted, immutable file format for BASON records with block structure, sparse index, and bloom filter. These files are the on-disk building blocks for BASONLevel and are reusable in BASONLite.

Background

An SST (Sorted String Table) file contains BASON records sorted by key. Records are grouped into fixed-size data blocks. A sparse index maps the first key of each block to its file offset. A bloom filter enables fast negative lookups. The file is written once and never modified.

The file structure borrows from LevelDB's table format but uses BASON encoding throughout — data blocks contain concatenated BASON records, the index block contains BASON records mapping keys to offsets, and the footer is a fixed-size structure pointing to the index and filter blocks.

File Layout

```
+-------------------+
| Data block 0      |   Sorted BASON records, concatenated
+-------------------+
| Data block 1      |
+------------------+
| ...              |
+------------------+
| Data block N     |
+------------------+
| Filter block     |   Bloom filter over all keys
+------------------+
| Index block      |   Sparse index: first key of each data block
+------------------+
| Footer (48 bytes) |
+------------------+
```

Data block: concatenated BASON records (flat mode, sorted by key). Each block is approximately `block_size` bytes (default 4096). A block boundary falls after the first record that causes the block to reach or exceed `block_size`. Blocks are optionally compressed (Snappy or zstd).

Filter block: a bloom filter with configurable bits-per-key (default 10). Encodes all keys present in the file.

Index block: a sequence of BASON records where the key is the first key of each data block and the value is a BASON Number containing the byte offset of that block within the file.

Footer (48 bytes):

```
+------------------+----------------+------------------+---------+
| filter_offset    | filter_size    | index_offset     | index_sz|
| 8B LE            | 8B LE          | 8B LE            | 8B LE   |
+------------------+----------------+------------------+---------+
| record_count     | magic "BASONSST"                            |
| 8B LE            | 8B                                          |
+------------------+---------------------------------------------+
```

Deliverables

3.1. SST Writer

```cpp
```

```
class SstWriter {
public:
    struct Options {
        size_t block_size = 4096;
        int bloom_bits_per_key = 10;
        // Compression type (none, snappy, zstd)
    };

    static SstWriter open(const std::string& path, Options opts = {});

    // Add a record. Keys must be added in strictly ascending sorted
    // order. Throws if order is violated.
    void add(const std::string& key, const BasonRecord& record);

    // Finalize: write filter block, index block, footer. Flush and
    // close the file. Returns metadata (file size, record count,
    // first key, last key).
    SstMetadata finish();
};
```

3.2. SST Reader

cpp

```
class SstReader {
public:
    static SstReader open(const std::string& path);

    // Point lookup. Returns nullopt if key is not found. Uses bloom
    // filter to short-circuit when possible.
    std::optional<BasonRecord> get(const std::string& key);

    // Range scan from start (inclusive) to end (exclusive).
    // Empty start means beginning of file. Empty end means end of
    // file.
    SstIterator scan(const std::string& start = "",
                     const std::string& end = "");

    // Check bloom filter without reading data. Returns false if the
    // key is definitely not present. Returns true if the key might
    // be present.
    bool may_contain(const std::string& key);

    // File metadata.
    SstMetadata metadata() const;
};

class SstIterator {
public:
    bool valid() const;
    void next();
    const std::string& key() const;
    const BasonRecord& record() const;
};
```

Correctness Criteria

- An SST file written and read back must produce identical records in identical order.
- `get()` must return the correct record for every key in the file and `nullopt` for every key not in the file.
- The bloom filter must never produce false negatives (it returns true for every key that is present). False positive rate at 10 bits-per-key should be approximately 1%.
- The file must be readable by the `basondump` tool from Assignment 1 (at least the data blocks, which are concatenated BASON records).

Performance Targets

- Write throughput: ≥ 100 MB/s (sequential writes, 4KB blocks).

- Point lookup: ≤ 2 disk reads for a file with 1 million records (one for the index block, one for the data block; footer and bloom filter are cached after first access).
- Range scan: ≥ 200 MB/s sequential read throughput.

Testing Guidance

Write a test that generates N random key-value pairs (N from 100 to 10 million), writes them sorted to an SST file, then verifies every key with `get()` and performs random range scans. Verify that the bloom filter false positive rate is within expected bounds. Test with and without compression.

Assignment 4: Memtable and Merge Iterator

Objective

Implement an in-memory sorted structure for BASON records and a merge iterator that combines multiple sorted sources with offset-based conflict resolution.

Background

The memtable is the write buffer for BASONLevel and serves as a fast lookup structure for BASONRedis. It accepts BASON records tagged with WAL offsets, maintains them in sorted key order, and supports point lookups, range scans, and ordered iteration.

The merge iterator combines multiple sorted iterators (from memtables and SST files) into a single sorted stream, resolving duplicate keys by choosing the record with the highest WAL offset (most recent write). Tombstones (null records) with offsets below a configurable threshold are dropped, enabling garbage collection of deleted keys.

Deliverables

4.1. Memtable

Implement using a skip list. A skip list is preferred over a red-black tree or B-tree because it supports concurrent reads without locking during writes (with careful implementation) and has good cache behavior for sequential iteration.

```cpp
```

```cpp
class Memtable {
public:
    // Insert or update a record. The offset is the WAL offset (LSN)
    // of the record, used for conflict resolution.
    void put(const std::string& key, const BasonRecord& record,
             uint64_t offset);

    // Point lookup. Returns the record and its offset, or nullopt.
    std::optional<std::pair<BasonRecord, uint64_t>>
        get(const std::string& key) const;

    // Range scan.
    MemtableIterator scan(const std::string& start = "",
                          const std::string& end = "") const;

    // Total memory usage in bytes (keys + values + skip list
    // overhead).
    size_t memory_usage() const;

    // Number of entries.
    size_t count() const;

    // Freeze: return an immutable snapshot of this memtable and
    // reset the mutable state. The frozen memtable supports reads
    // but not writes.
    std::shared_ptr<const Memtable> freeze();
};
```

4.2. Merge Iterator

```
cpp
```

```cpp
class MergeIterator {
public:
    // Construct from a list of sorted iterators. Each iterator
    // must yield (key, record, offset) triples in ascending key
    // order. When multiple iterators yield the same key, the one
    // with the highest offset wins. Tombstones with offsets below
    // min_live_offset are suppressed.
    MergeIterator(
        std::vector<std::unique_ptr<SortedIterator>> sources,
        uint64_t min_live_offset = 0);

    bool valid() const;
    void next();
    const std::string& key() const;
    const BasonRecord& record() const;
    uint64_t offset() const;
};
```

The merge iterator must use a min-heap (priority queue) over the source iterators, keyed by `(key, -offset)` so that for duplicate keys, the highest offset surfaces first. After yielding a key, all sources with the same key must be advanced past it.

4.3. Sorted Iterator Interface

Define a common interface that both `MemtableIterator` and `SstIterator` conform to, so they can be passed to `MergeIterator`:

```cpp
class SortedIterator {
public:
    virtual ~SortedIterator() = default;
    virtual bool valid() const = 0;
    virtual void next() = 0;
    virtual const std::string& key() const = 0;
    virtual const BasonRecord& record() const = 0;
    virtual uint64_t offset() const = 0;
};
```

Correctness Criteria

- Memtable: `get()` returns the most recently `put()` record for a given key (highest offset).
- Memtable: iteration yields keys in strictly ascending sorted order.
- Memtable: `freeze()` produces a snapshot unaffected by subsequent mutations.
- Merge iterator: output is in strictly ascending key order.

- Merge iterator: for duplicate keys across sources, only the record with the highest offset is yielded.
- Merge iterator: tombstones with offsets below `min_live_offset` are not yielded.
- Merge iterator: tombstones with offsets at or above `min_live_offset` are yielded (the caller needs them to suppress older versions in lower levels).

Performance Targets

- Memtable insert: ≥ 1 million records/sec for 16-byte keys and 64-byte values.
- Memtable point lookup: ≥ 2 million lookups/sec.
- Merge iterator: ≥ 100 MB/s over 10 sources of 1 million records each.

Testing Guidance

For the memtable, write a concurrent test: one thread inserts records while another reads. Verify that reads always see a consistent snapshot (no torn records, no missing keys that were inserted before the read started). For the merge iterator, generate multiple sorted sequences with controlled overlaps and verify the merged output against a naive sort-and-deduplicate of the union.

Part II: Storage Engines

Assignment 5: BASONCask

Objective

Build an append-only log-structured key-value store with an in-memory hash index, inspired by Bitcask.

Architecture

BASONCask is the simplest database in the course. It has two components: the WAL (from Assignment 2) and a hash map. Every write appends a BASON record to the WAL. The hash map maps each key to the global offset of its most recent record in the WAL. A read looks up the offset in the hash map, seeks to that position in the WAL, and reads the record.

```
Write path:
  1. Append BASON record to WAL → get offset
  2. Update hash map: key → offset
  3. Checkpoint + sync per configured policy
  4. Acknowledge

Read path:
  1. Look up key in hash map → offset
  2. Seek to offset in WAL segment file
  3. Read and decode BASON record
  4. Return (or not-found if key absent from hash map)

Delete path:
  1. Append tombstone record (Boolean, empty value) → get offset
  2. Remove key from hash map
  3. Checkpoint + sync
```

Deliverables

5.1. BASONCask Store

```cpp
class BasonCask {
public:
    struct Options {
        std::string dir;
        uint64_t max_segment_size = 64 * 1024 * 1024;  // 64 MB
        bool sync_on_write = false;
    };

    static BasonCask open(Options opts);

    // Write a key-value pair.
    void put(const std::string& key, const BasonRecord& value);

    // Read a key. Returns nullopt if not found or deleted.
    std::optional<BasonRecord> get(const std::string& key);

    // Delete a key.
    void del(const std::string& key);

    // Close the database (checkpoint, sync, release resources).
    void close();
};
```

5.2. Startup Recovery

On open, scan all WAL segment files in offset order, replaying records to rebuild the hash map. After recovery, the hash map contains the offset of the latest version of every live key.

5.3. Log Compaction

Implement a background compaction process:

1. Iterate the hash map to find live keys and their offsets.
2. Write a new WAL segment containing only the latest record for each live key (skip tombstones and superseded versions).
3. Atomically swap: update the hash map offsets to point to the new segment, delete old segments.

Compaction must be safe to run concurrently with reads and writes. Writes go to the active (new) segment. Reads may hit old segments until the swap completes. The swap must be crash-safe (if the process crashes mid-compaction, recovery must produce a correct state).

Correctness Criteria

- After `put(k, v)` + sync, a crash and recovery must yield `get(k) == v`.
- After `del(k)` + sync, a crash and recovery must yield `get(k) == nullopt`.
- Compaction must not lose or corrupt any live key-value pair.
- Concurrent reads during compaction must return correct results.

Performance Targets

- Write: ≥ 100,000 puts/sec (batched, sync every 1000).
- Read (warm, key in OS page cache): ≤ 10 μs per get.
- Startup recovery: ≥ 200 MB/s (limited by sequential WAL scan).

Analysis Questions

Include a brief written report addressing:

1. What is the maximum database size BASONCask can handle, given that the hash map must fit in memory? Estimate the memory overhead per key.
2. Why is range scanning expensive in BASONCask? What would you need to add to support efficient range queries?
3. Compare the write amplification of BASONCask (with compaction) to a B-tree store. Under what workload does BASONCask win?

Assignment 6: BASONLevel

Objective

Build an LSM-tree key-value store with leveled compaction, using the WAL, memtable, SST files, and merge iterator from previous assignments.

Architecture

BASONLevel maintains a sorted, persistent key-value store. Writes go to the WAL and memtable. When the memtable reaches a size threshold, it is frozen and flushed to an SST file at level 0. Background compaction merges overlapping SST files from level N to level N+1, maintaining the invariant that files at levels ≥ 1 have non-overlapping key ranges.

```
Write path:
  1. Append BASON record to WAL → get offset
  2. Insert into active memtable with offset
  3. If memtable exceeds size threshold:
     a. Freeze memtable
     b. Flush frozen memtable to L0 SST file
     c. Record flush in MANIFEST
     d. Truncate WAL segments below flush offset

Read path:
  1. Check active memtable
  2. Check frozen memtable (if flush in progress)
  3. Check L0 files (newest first, may overlap)
  4. Check L1+ files (binary search by key range, no overlap)
  5. Return first match (highest offset wins)

Merge semantics:
  - Latest offset wins for duplicate keys
  - Tombstones suppress older versions
  - Tombstones are dropped during compaction when below the
    oldest live snapshot offset
```

MANIFEST

The MANIFEST tracks the current set of SST files, their levels, key ranges, and offsets. It is itself a BASON WAL (same format as Assignment 2) containing version edit records:

```
Each version edit is a BASON Object with keys:
  "add"    → Array of Objects: {level, file, first_key, last_key, size}
  "remove" → Array of Objects: {level, file}
  "flush"  → Number: the WAL offset at which the flush occurred
```

On startup, replay the MANIFEST to reconstruct the file set. The first record in a new MANIFEST is a full snapshot (lists all files); subsequent records are deltas. Periodic MANIFEST rotation writes a fresh snapshot and starts a new file.

Compaction

Implement leveled compaction:

- L0 files may overlap with each other. When the count of L0 files exceeds a threshold (default 4), compact all overlapping L0 files plus overlapping L1 files into new L1 files.
- For levels ≥ 1, when a level's total size exceeds its threshold (level_size_base × ratio^level), pick the file with the most overlap with the next level and compact it with the overlapping files at the next level.
- Compaction reads from input files using a merge iterator, writes to new SST files, records the version edit in the MANIFEST, then deletes input files.

Deliverables

6.1. BASONLevel Store

```
cpp
```

```cpp
class BasonLevel {
public:
    struct Options {
        std::string dir;
        size_t memtable_size = 4 * 1024 * 1024;    // 4 MB
        size_t l0_compaction_trigger = 4;
        size_t level_size_base = 64 * 1024 * 1024;  // 64 MB
        size_t level_size_ratio = 10;
        size_t max_levels = 7;
        size_t block_size = 4096;
        int bloom_bits_per_key = 10;
    };

    static BasonLevel open(Options opts);

    void put(const std::string& key, const BasonRecord& value);
    std::optional<BasonRecord> get(const std::string& key);
    void del(const std::string& key);

    // Range scan.
    std::unique_ptr<SortedIterator> scan(
        const std::string& start = "",
        const std::string& end = "");

    // Create a snapshot. Reads through this snapshot see a consistent
    // view at the time of creation. The snapshot pins tombstones
    // above its offset from being garbage collected.
    std::shared_ptr<Snapshot> snapshot();

    // Force compaction of a level (for testing).
    void compact_level(int level);

    void close();
};
```

6.2. Background Compaction Thread

Compaction runs in a background thread. Implement a scheduling loop that wakes on flush completion or periodic timer, checks compaction triggers, and runs compaction as needed. The compaction must not block reads or writes (except for brief mutex acquisitions to update the file set).

6.3. Metrics

Expose internal metrics for analysis:

```cpp
struct LevelMetrics {
    size_t num_files_per_level[MAX_LEVELS];
    size_t bytes_per_level[MAX_LEVELS];
    uint64_t total_compactions;
    uint64_t total_bytes_written;     // includes compaction rewrites
    uint64_t total_bytes_read;
    double write_amplification;       // total_bytes_written / user_bytes_written
    double read_amplification;        // disk_reads / user_reads
};
```

Correctness Criteria

- All correctness criteria from BASONCask apply.
- Range scans must yield keys in sorted order with no duplicates and no tombstones visible to the caller.
- Snapshots must provide a consistent point-in-time view.
- Compaction must not lose data, introduce duplicates, or disorder keys.
- Crash at any point during compaction must be recoverable (MANIFEST is the source of truth; orphaned SST files are cleaned up).

Performance Targets

- Write throughput: ≥ 10 MB/s sustained (including compaction overhead).
- Point lookup (data in cache): ≤ 50 μs.
- Point lookup (data on disk, 7 levels): ≤ 500 μs (with bloom filters reducing reads to ~1 per level).
- Range scan: ≥ 100 MB/s for sequential reads.

Analysis Questions

1. Measure write amplification under a uniform random write workload at 1 million, 10 million, and 100 million keys. Plot total bytes written (including compaction) vs user bytes written.
2. Measure the effect of bloom filter bits-per-key on read amplification. Test at 0, 5, 10, 15, and 20 bits per key.
3. Compare L0 compaction trigger values of 1, 4, 8, and 16. How does this affect write stalls, read amplification, and space amplification?

Assignment 7: BASONLite

Objective

Build a page-based B-tree database with ACID transactions, inspired by SQLite. This is the most complex assignment in the course.

Architecture

BASONLite stores data in a single database file divided into fixed-size pages. Tables are B+ trees keyed by a primary key (a BASON path string). Each leaf page contains sorted BASON records. Interior pages contain separator keys and child page pointers. The WAL provides crash recovery and atomic transactions.

```
Database file:
  Page 1: database header + schema catalog root
  Page 2+: B-tree pages (interior and leaf), overflow pages, freelist

WAL (separate file):
  Stores page images before they are written to the database file.
  Provides atomic commit and crash recovery.
```

Page Format

All pages are `page_size` bytes (default 4096). Each B-tree page:

```
Page header (12 bytes):
+------------+---------+----------+---------+----------------+
| page_type  | count   | free_ofs | frag    | right_child    |
| 1B         | 2B LE   | 2B LE    | 1B      | 4B LE          |
+------------+---------+----------+---------+----------------+
| (1B unused padding for alignment)                          |
+------------------------------------------------------------+

Followed by:
  Cell pointer array: count × 2B LE offsets into page
  ... free space ...
  Cells: BASON records packed from end of page toward start

page_type:
  0x01 = interior node (B-tree)
  0x02 = leaf node (B-tree)
  0x03 = overflow page
  0x04 = freelist trunk page
```

Leaf cell: a complete BASON record (flat mode, key + value).

Interior cell: a BASON record where the key is the separator and the value is a BASON Number encoding the left child page number. The `right_child` field in the header points to the rightmost child.

Overflow: when a cell's total size exceeds `page_size / 4`, the excess is stored in linked overflow pages. The cell stores the inline portion plus a 4-byte overflow page number.

Schema Catalog

Page 1 contains the database header (in the first 64 bytes) and the root of the schema catalog B-tree. The schema catalog maps table names to their root page numbers and column definitions, stored as BASON Object records.

WAL Integration

BASONLite uses the WAL from Assignment 2 in page-image mode:

- Before modifying a page, write the new page image to the WAL.
- On commit, write a hash checkpoint.
- A background checkpoint process copies committed page images from the WAL back to the database file.
- On crash, replay the WAL: for each committed transaction (delimited by hash checkpoints), copy page images to the database file.

The WAL offset serves as the transaction ID. Readers see a consistent snapshot by checking the WAL first: for any page, the most recent committed WAL image (at or before the reader's snapshot offset) takes precedence over the database file.

Deliverables

7.1. Page Manager

```cpp
```

```
class PageManager {
public:
    static PageManager open(const std::string& path,
                            size_t page_size = 4096);

    // Read a page by number (1-indexed).
    Page read_page(uint32_t page_no);

    // Write a page (through the WAL).
    void write_page(uint32_t page_no, const Page& page);

    // Allocate a new page (from freelist or by extending the file).
    uint32_t allocate_page();

    // Free a page (add to freelist).
    void free_page(uint32_t page_no);
};
```

7.2. B-tree

```cpp
class BTree {
public:
    BTree(PageManager& pm, uint32_t root_page);

    void insert(const std::string& key, const BasonRecord& value);
    std::optional<BasonRecord> find(const std::string& key);
    bool remove(const std::string& key);
    BTreeIterator scan(const std::string& start = "",
                       const std::string& end = "");

private:
    // Internal operations
    void split_child(uint32_t parent_page, int child_index);
    void merge_children(uint32_t parent_page, int child_index);
    void rebalance(uint32_t page_no);
};
```

7.3. Transaction Manager

```cpp
```

```
class Transaction {
public:
    void put(const std::string& table, const std::string& key,
             const BasonRecord& value);
    std::optional<BasonRecord> get(const std::string& table,
                                   const std::string& key);
    void del(const std::string& table, const std::string& key);
    void commit();
    void rollback();
};

class BasonLite {
public:
    static BasonLite open(const std::string& path);
    Transaction begin();
    void close();
};
```

7.4. Checkpoint Process

Implement WAL checkpointing: copy committed page images from the WAL to the database file, then truncate the WAL. This can run in the background or be triggered manually.

Correctness Criteria

- ACID transactions: committed data survives crashes; rolled-back data is never visible; concurrent readers see consistent snapshots.
- B-tree invariants: keys are sorted within each page, separator keys correctly partition the key space, all leaves are at the same depth.
- No page leaks: every allocated page is either reachable from the root or on the freelist.
- The database file is self-consistent after checkpoint (readable without the WAL).

Performance Targets

- Point lookup: O(log N) with ≤ 4 page reads for 10 million records (given 4KB pages with ~100 keys each).
- Insert throughput: ≥ 50,000 inserts/sec in a single transaction.
- The database file should be openable by the `basondump` tool (treating each page as a container of BASON records).

Analysis Questions

1. Compare the write amplification of BASONLite to BASONLevel for a workload of 1 million random inserts. Explain the difference.
2. What happens to performance as the database file becomes fragmented (many freelist pages interspersed with live pages)? Implement and measure a VACUUM operation that rewrites the file compactly.
3. What is the maximum transaction size before WAL space becomes a concern? How does this compare to SQLite's behavior?

Assignment 8: BASONRedis

Objective

Build an in-memory data structure server that uses BASON as its wire protocol, command encoding, and persistence format.

Architecture

BASONRedis keeps the entire dataset in memory as a hash map from key (path string) to BASON record. The record's type tag determines which operations are valid on it. Persistence is via the WAL (AOF-style) and periodic full snapshots (RDB-style).

The wire protocol is BASON over TCP: commands are BASON Array records, responses are BASON records. There is no separate protocol.

```
Client → Server:  BASON Array: ["SET", "user/42/name", "Alice"]
Server → Client:  BASON Boolean: "true"

Client → Server:  BASON Array: ["GET", "user/42/name"]
Server → Client:  BASON String: "Alice"
```

Type-Driven Commands

The BASON type tag of a stored value determines which commands are valid:

```
Boolean (b/B):
  GET, SET, EXISTS (null check)

Number (n/N):
  GET, SET, INCR, DECR, INCRBY, DECRBY

String (s/S):
  GET, SET, APPEND, STRLEN, GETRANGE

Array (a/A):
  LPUSH, RPUSH, LPOP, RPOP, LINDEX, LLEN, LRANGE

Object (o/O):
  HGET, HSET, HDEL, HKEYS, HVALS, HLEN, HGETALL
```

Universal commands (work on any type):

```
DEL, EXISTS, TYPE, KEYS (glob pattern), TTL, EXPIRE, PERSIST
```

Key Expiry

Keys may have an associated TTL (time-to-live). Implement expiry
using two mechanisms:

- **Lazy expiry**: on every read, check if the key has expired. If so, delete it and return not-found.
- **Active expiry**: a background thread periodically samples random keys and deletes expired ones. Adaptive frequency: if many keys are expired in a sample, increase the scan rate.

Store TTL metadata as a separate internal key (e.g., `__ttl/<user_key>` → Number with epoch timestamp in milliseconds).

Persistence

AOF (WAL-based): every mutation command appends a BASON record to the WAL. The record is the command itself (a BASON Array), allowing replay on restart. The WAL hash checkpoints serve as durability boundaries.

Snapshot: periodically, serialize the entire dataset as a single BASON nested-mode document (an Object whose children are all key-value pairs) and write it to a `.bason` file. On startup, if a snapshot exists, load it first, then replay the WAL from the snapshot's offset.

Deliverables

8.1. Data Store

```cpp
class BasonRedisStore {
public:
    // Core operations
    void set(const std::string& key, BasonRecord value);
    std::optional<BasonRecord> get(const std::string& key);
    bool del(const std::string& key);
    bool exists(const std::string& key);
    BasonType type(const std::string& key);

    // Number operations
    int64_t incr(const std::string& key);
    int64_t incrby(const std::string& key, int64_t delta);

    // String operations
    size_t append(const std::string& key, const std::string& suffix);

    // Array operations
    size_t lpush(const std::string& key, const BasonRecord& value);
    size_t rpush(const std::string& key, const BasonRecord& value);
    std::optional<BasonRecord> lpop(const std::string& key);
    std::optional<BasonRecord> rpop(const std::string& key);
    std::optional<BasonRecord> lindex(const std::string& key,
                                      int64_t index);
    size_t llen(const std::string& key);

    // Object operations
    void hset(const std::string& key, const std::string& field,
              const BasonRecord& value);
    std::optional<BasonRecord> hget(const std::string& key,
                                    const std::string& field);
    bool hdel(const std::string& key, const std::string& field);
    std::vector<std::string> hkeys(const std::string& key);

    // TTL
    void expire(const std::string& key, uint64_t ms);
    int64_t ttl(const std::string& key);  // -1 = no TTL, -2 = missing
    void persist(const std::string& key);

    // Pattern matching
    std::vector<std::string> keys(const std::string& glob_pattern);
};
```

8.2. Network Server

```cpp
class BasonRedisServer {
public:
    // Start the server on the given port. Accept TCP connections,
    // read BASON Array commands, dispatch to the store, and send
    // BASON responses.
    void listen(uint16_t port);

    // Graceful shutdown.
    void shutdown();
};
```

Implement a simple event loop (epoll/kqueue or `std::thread` per connection). Parse incoming bytes as BASON records, dispatch based on the command name (first Array element), execute, and encode the response as a BASON record.

8.3. Persistence Manager

```cpp
class PersistenceManager {
public:
    // AOF: append a command to the WAL.
    void log_command(const BasonRecord& command);

    // Snapshot: write current state to a .bason file.
    void snapshot(const BasonRedisStore& store,
                  const std::string& path);

    // Recovery: load snapshot + replay WAL.
    void recover(BasonRedisStore& store);
};
```

8.4. Client Library

```cpp
```

```
class BasonRedisClient {
public:
    void connect(const std::string& host, uint16_t port);

    // Send a command and receive a response. The command is a BASON
    // Array.
    BasonRecord execute(const std::vector<std::string>& args);

    // Convenience methods
    void set(const std::string& key, const std::string& value);
    std::optional<std::string> get(const std::string& key);
    int64_t incr(const std::string& key);
    // ... etc
};
```

Correctness Criteria

- All commands must produce correct results as defined by their type semantics.
- Type mismatch must return an error (e.g., INCR on a String value).
- After AOF replay, the store state must be identical to the state at the time of the last hash checkpoint.
- After snapshot + WAL replay, the store state must be identical to the state at shutdown.
- Expired keys must never be visible to clients.

Performance Targets

- Single-client throughput: ≥ 100,000 commands/sec for GET/SET.
- Multi-client (10 concurrent): ≥ 300,000 commands/sec aggregate.
- Snapshot write: ≥ 200 MB/s for serializing the in-memory state.
- Startup recovery: snapshot load ≥ 200 MB/s, WAL replay ≥ 500,000 commands/sec.

Analysis Questions

1. Compare BASONRedis's AOF persistence to BASONCask. Both are append-only logs — what are the structural differences and why?
2. What happens when the dataset approaches available memory? Propose and (optionally) implement an eviction policy (LRU, LFU, or random) with a configurable maximum memory limit.
3. Measure the latency impact of fsync frequency on SET commands. Compare sync-per-write, sync-per-100, and sync-per-second.

General Guidelines

Submission

Each assignment is submitted as a Git repository with:

- Source code under `src/`.
- Tests under `test/`.
- A `Makefile` or `CMakeLists.txt` that builds the project and runs tests.
- A brief report (`REPORT.md`) answering the analysis questions and describing design decisions.

Code Quality

- No memory leaks (test with AddressSanitizer).
- No data races (test with ThreadSanitizer for concurrent components).
- Error handling: all I/O errors must be detected and reported, not silently ignored.

Interoperability Testing

A key property of this course is format-level interoperability. At several checkpoints, students will exchange files and verify compatibility:

- After Assignment 1: exchange BASON-encoded files and verify cross-decoding.
- After Assignment 3: exchange SST files and verify that another student's reader can open your SST files.
- After Assignment 8: connect one student's BASONRedis client to another's server.

These exchanges verify that the BASON specification is implemented consistently and that the format truly serves as a universal interface.

Measurement and Reporting

Every assignment includes performance targets. Students must measure and report their actual numbers with a description of the test environment (CPU, disk type, OS). The targets are guidelines, not pass/fail thresholds — understanding why your implementation is faster or slower than the target is more valuable than hitting the number.

Building on Previous Work

Assignments in Part II explicitly reuse code from Part I. Students should design their Part I components with reuse in mind: clean interfaces, minimal coupling, and well-documented behavior at boundaries. A poorly designed codec or WAL will create compounding problems in every store built on top of it.

Suggested Timeline

For a 15-week semester:

Weeks	Assignment
1-2	Assignment 1: BASON Codec
3-4	Assignment 2: Write-Ahead Log
5-6	Assignment 3: SST Files
7	Assignment 4: Memtable and Merge Iterator
8-9	Assignment 5: BASONCask
10-11	Assignment 6: BASONLevel
12-13	Assignment 7: BASONLite
14-15	Assignment 8: BASONRedis

Assignments 7 and 8 are the most complex. Students who fall behind on Part I will struggle. Start early.

Afterword

By the end of this course, you will have built four working databases from scratch, totaling roughly 5,000–10,000 lines of code, all sharing a common format and reusable components. More importantly, you will have internalized the tradeoffs that define database design:

- BASONCask taught you that an append-only log is the simplest possible durable store, but that in-memory indexing limits your scale and random access to the log limits your read patterns.
- BASONLevel taught you that sorting data on disk enables efficient range queries and that the LSM tree is an elegant way to batch random writes into sequential I/O, at the cost of write and space amplification from compaction.
- BASONLite taught you that B-trees give you random access with logarithmic cost and that ACID transactions require careful coordination between a WAL and a page

store, but that in-place updates cause write amplification on every mutation.

- BASONRedis taught you that an in-memory store can be orders of magnitude faster than a disk-based one, that the wire protocol is a design decision with real performance consequences, and that persistence is a spectrum from "lose everything on crash" to "durable on every write."

The fact that all four systems use the same format at every layer — codec, WAL, storage, and wire protocol — is itself a lesson. A good abstraction reduces the number of concepts you need to hold in your head. BASON is a simple abstraction. The databases built on it are not. The gap between the two is where engineering lives.