BANDIRMA ONYEDİ EYLÜL UNIVERSITY

FACULTY OF ENGINEERING AND NATURAL SCIENCES

DEPARTMENT OF COMPUTER ENGINEERING

BMI3241 — FILE ORGANIZATION

Term Project

CardCatalog: Access Optimization and Performance Analysis on Large-Scale Datasets

Group Members

Yunus Emre Ayhan — 2111504053

Ahmet Başar Alüzüm — 2211504017

Mustafa Yusuf Yaman — 2111504044

Course Instructor: Asst. Prof. Dr. Arzum Karataş

Bandırma, June 2026

Abstract

This report presents CardCatalog, the core data-management unit of a library automation system designed to answer title, author, and category queries over one million book records in well under a millisecond while supporting full create, read, update, and delete (CRUD) operations. The design begins from a direct measurement of the dataset: the one million records contain only 78 distinct titles, 59 distinct authors, and 15 distinct categories, and their identifiers form a dense, contiguous sequence. These two properties — very low field cardinality with large result sets, and sequential identifiers — determine the entire architecture. The data is stored column-wise as dictionary-encoded byte arrays (a struct-of-arrays layout); the identifier is not stored at all but recovered as offset = id − 1; and a dense bitmap inverted index gives each field value its own bit vector. Exact-match queries return a cached bit vector, counts are answered in O(1) from cached cardinalities, multi-field (compound) queries reduce to word-wise bitmap AND, and prefix queries combine a binary search over a sorted dictionary with bitmap OR. Deletion uses a tombstone bitmap, and an optional binary snapshot restarts the system in a few milliseconds without re-parsing the source. Loading is performed by a memory-mapped, two-pass, multithreaded parser. We measured time and memory at the 10K, 100K, and 1M scales required by the assignment. At one million records the system loads in 41 ms, occupies about 28 MB of resident memory, answers point look-ups and counts in roughly 25 ns, and completes compound queries in under 9 µs. The scalability analysis shows that point access and counting stay constant as the data grows, set-returning queries scale with the size of the result rather than the dataset, and compound queries remain near-constant in microseconds because of word-wise bitmap intersection.

Contents

Abstract **1**
List of Abbreviations **3**
1. Introduction **4**
1.1 Problem Definition and Objective 4
1.2 Contributions 4
1.3 Report Organization 4
2. Background and Related Work **4**
2.1 File Organization Principles 4
2.2 Indexing Alternatives and Comparative Analysis 5
2.3 Dataset Profile and Design Drivers 6
3. System Design and Implementation **6**
3.1 Storage Layout: Columnar Storage, Dictionary Encoding, and Implicit Identifiers 7
3.2 Secondary Index: Dense Bitmap Inverted Index 7
3.3 Loading Pipeline: Memory-Mapped, Two-Pass Parallel Parsing 8
3.4 Query Processing 8
3.5 CRUD, Tombstone Deletion, and Compaction 9
3.6 Persistence: Binary Snapshot 9
4. Performance Evaluation and Scalability Analysis **10**
4.1 Methodology 10
4.2 Results 10
4.3 Scalability Analysis (10K ↔ 1M) 11
5. Engineering Challenges **12**
6. Conclusion **13**
References **13**
Appendix A — Task Chart **15**
Appendix B — Build and Usage **15**

List of Abbreviations

Abbreviation	Meaning
AVX2	Advanced Vector Extensions 2 (SIMD instruction set)
CRUD	Create, Read, Update, Delete
CSV	Comma-Separated Values
mmap	Memory-mapped file (POSIX)
popcount	Population count — number of set bits in a word
RLE	Run-Length Encoding
RSS	Resident Set Size (process memory reported by the OS)
SIMD	Single Instruction, Multiple Data
SoA	Structure of Arrays (columnar layout)
STL	Standard Template Library (C++)
WAH	Word-Aligned Hybrid (bitmap compression)

1. Introduction

1.1 Problem Definition and Objective

The assignment asks for the core data-management unit of a library automation system whose primary data source, books_dataset.txt, holds one million book records. Each record carries a unique id, a title, an author, and a category; the supplied data also includes a publication year, which we keep. The system must answer title, author, and category queries within milliseconds and support full CRUD on one million records. We were further required to measure time and memory at three scales — 10K, 100K, and 1M — and to analyse how the chosen indexing architecture behaves under these order-of-magnitude increases in data volume. We set an aggressive target: the highest achievable query speed at the smallest reasonable memory footprint. The resulting system answers every query type in microseconds, uses about 28 MB of memory at one million records, and restarts from a binary snapshot in a few milliseconds.

1.2 Contributions

In summary, the work contributes a measurement-driven justification for a dense bitmap inverted index on this workload; an implicit primary index that stores no identifier and computes each record's position arithmetically; a memory-mapped, two-pass parallel loader; constant-time point access and counting; and an honest performance and scalability study at the three required scales, including a subtle subset-loading defect that we found and corrected.

1.3 Report Organization

Section 2 reviews the relevant file-organization principles and indexing alternatives and explains, directly from the dataset profile, why a bitmap inverted index is the right structure here. Section 3 describes the storage layout, the index, the loading pipeline, query processing, CRUD, and persistence. Section 4 reports time and memory at the 10K, 100K, and 1M scales and analyses scalability. Section 5 discusses the engineering challenges we encountered, and Section 6 concludes.

2. Background and Related Work

2.1 File Organization Principles

A data-management unit is shaped by how records are placed on storage and how they are located again [1]. A primary index maps a key to a record's position, while a secondary index

maps non-key attribute values to the records that contain them. When keys are dense and sequential, a record's position can be computed directly from its key rather than searched for — a form of relative or implicit addressing that removes the need for a separate key-to-position structure [1]. Secondary access on low-cardinality attributes is classically supported by an inverted file, which lists, for each attribute value, the records that hold it [8]. The remaining design freedom is how those inverted lists are represented.

2.2 Indexing Alternatives and Comparative Analysis

The assignment explicitly leaves the choice of indexing mechanism to the team, so we compared four families. Simple (sequential) indexing needs no extra structure but answers a secondary-attribute query by scanning all records, which is unacceptable at one million rows. Hashing gives excellent exact-match look-up, but a multi-field (compound) query must intersect large posting lists, and when each list holds tens of thousands of records that intersection dominates the cost. B-trees and B+-trees are the standard answer for ordered access and range or prefix queries [1]; however, for equality on low-cardinality attributes they maintain far more structure than this workload needs. The bitmap inverted index instead represents each attribute value as a bit vector over record positions [2]. Introduced for read-mostly decision-support workloads [3], bitmap indexes excel precisely when attribute cardinality is low and result sets are large: a value's records are a single bit vector, a count is the population count of that vector, an intersection of two predicates is a word-wise AND, and a union is a word-wise OR. These bitwise operations exploit bit-level parallelism and modern SIMD instructions, so they run in time proportional to the number of machine words rather than the number of matching records [2], [9].

A natural follow-up question is compression. When bit vectors are sparse, run-length schemes such as the Word-Aligned Hybrid (WAH) code [4] or the packed-array format of Roaring [5], [6] reduce both space and operation time. Our result sets are not sparse: a single category already covers roughly 6.7 % of all records, so its bitmap is dense. At that density, run-length and packed-array compression save little space while adding indirection on the hot AND/OR path; an uncompressed (dense) bitmap keeps the word-wise operations maximal in speed and introduces no external dependency. We therefore chose dense bitmaps deliberately. We estimate that a Roaring representation would shrink the index to roughly 6 MB, but at a genuine cost in code complexity and a measurable cost on intersection throughput for dense data — a trade-off we judged not worthwhile for this workload [5], [6].

2.3 Dataset Profile and Design Drivers

We began by measuring the data rather than assuming its shape; the result is summarised in Table 1.

Table 1. Profile of books_dataset.txt (one million records).

Property	Value
Total records	1,000,000
Distinct titles	78
Distinct authors	59
Distinct categories	15 (≈ 64,000–65,000 records each)
Identifier range	1 … 1,000,000 (unique and contiguous)
Publication year range	1900 … 2025
Rows with commas inside the title	25,602

Two observations dictated the architecture. First, one million records spread over only 152 distinct field values (78 + 59 + 15) means every query returns tens of thousands of records and every attribute is extremely low-cardinality — the canonical case for bitmap indexing [2], [3]. Second, identifiers run from 1 to N with no gaps, so a record's storage position is simply offset = id − 1; no key-to-position map is needed, and the identifier need not be stored at all, saving both memory and an indirection on every look-up [1]. A practical complication is that 25,602 rows contain unquoted commas inside the title, for example “The Lion, the Witch, and the Wardrobe”. Because the last three fields are always present, we anchor parsing from the end of the line — the year follows the last comma, the category and author occupy the next two comma-separated slots, and everything between the first comma and the third-from-last comma is the title — which resolves rows with five, six, or seven commas under a single rule.

3. System Design and Implementation

Figure 1 shows the overall data flow, from the memory-mapped source file through the columnar store and the inverted index to the query layer.

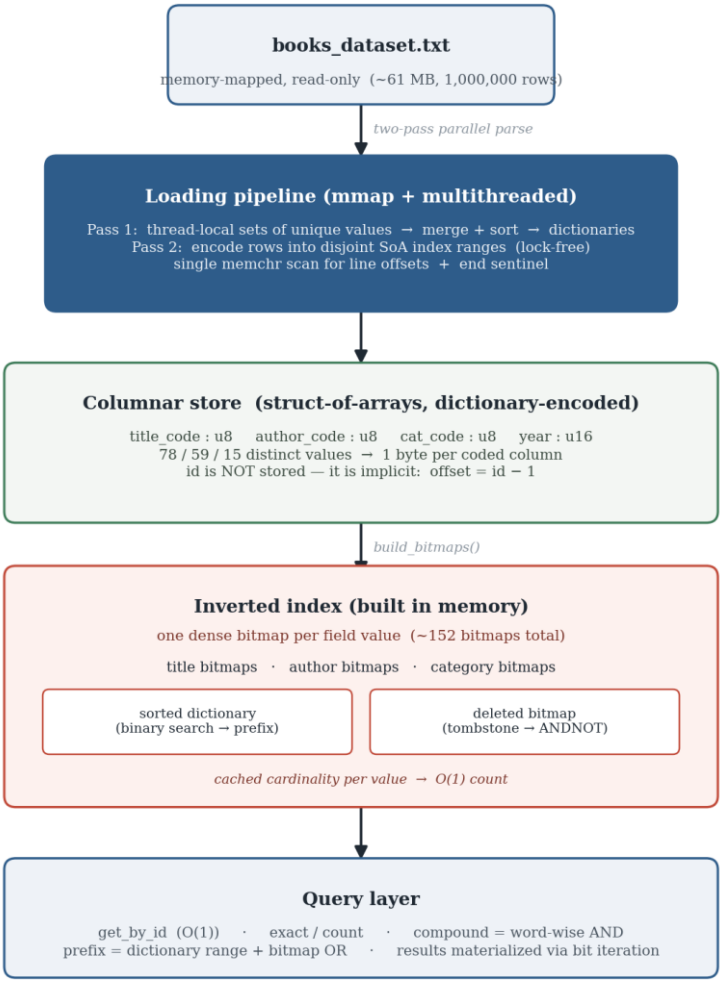

Figure 1. CardCatalog data flow: memory-mapped two-pass loading, dictionary-encoded columnar storage, a dense bitmap inverted index, and the query layer.

3.1 Storage Layout: Columnar Storage, Dictionary Encoding, and Implicit Identifiers

Records are stored column-wise (a struct-of-arrays layout) so that a scan over one attribute touches only that attribute's array, which is cache-efficient [7]. Each low-cardinality field is dictionary-encoded: the 78 titles, 59 authors, and 15 categories are mapped to single-byte codes, so each coded column uses one byte per record and the publication year uses two. The identifier is not stored; it is recovered as id = offset + 1. In total the columns occupy about 5 MB at one million records. Dictionary encoding both compresses the columns and turns equality comparisons into integer rather than string comparisons [7].

3.2 Secondary Index: Dense Bitmap Inverted Index

For every field value we keep one dense bit vector over record positions — about 152 vectors in all. A bit set at position i means that record i holds that value. The vectors are stored as arrays

of 64-bit words; AND, OR, and AND-NOT are implemented as word-wise loops that the compiler vectorises to AVX2 under -O3 -march=native, and population count compiles to the hardware POPCNT instruction [9]. Alongside the bitmaps we keep, per field, a dictionary sorted by value to support prefix queries through binary search, and a single deleted bitmap shared across the table to mark removed records. We also cache each value's cardinality so that a count needs neither a scan nor a population count when no deletions are pending.

3.3 Loading Pipeline: Memory-Mapped, Two-Pass Parallel Parsing

The source file is mapped read-only with mmap (with a plain-read fallback on Windows) and advised for sequential access, so the operating system streams pages on demand instead of copying the whole file [10]. Line boundaries are found in a single memchr scan and — importantly — an explicit end sentinel is recorded after the last scanned line; Section 5 explains why this matters. Parsing then runs in two parallel passes. In the first pass each thread builds thread-local sets of the unique values it sees; these sets are merged and sorted to assign deterministic codes and to build the dictionaries. In the second pass each thread writes codes into a disjoint range of the column arrays, so no two threads touch the same index and no locking is required. The inverted index is then built from the finished columns, with the three fields processed concurrently.

3.4 Query Processing

The operations and their costs are summarised in Table 2. An exact-match query maps the value to its code and returns the corresponding bit vector, with the deleted bitmap subtracted (AND-NOT) when deletions are pending. A count returns the cached cardinality in O(1) when there are no tombstones, and otherwise computes the population count of the value's vector with the deleted bits removed, without materialising a copy. A compound query intersects the per-field bit vectors with word-wise AND, skipping any field left blank, then subtracts deletions. A prefix query performs a binary search over the sorted dictionary to find the first matching value and ORs together the bit vectors of all values that share the prefix. In every case the result is a bit vector; the matching records are produced by iterating its set bits, each of which is a position that decodes to a full record in constant time.

Table 2. Operations supported on one million records and their complexity.

Operation	Method	Complexity
Create	append at offset n; encode values; set three bits; grow columns	amortised O(1)

Operation	Method	Complexity
Read by id	offset = id − 1; bounds and deleted check; decode codes	O(1)
Read by field	value → code; return bit vector (minus deleted)	O(1) + result
Count	cached cardinality when no tombstones	O(1)
Compound (AND)	word-wise AND of the per-field bit vectors	O(n / 64) words
Prefix	binary search in sorted dictionary + bitmap OR	O(log k + matches)
Update	clear old bit, set new bit, adjust code and cardinality	O(1)
Delete	set bit in the deleted bitmap (tombstone)	O(1)
Compaction	rebuild live columns, renumber, rebuild bitmaps	O(n)

3.5 CRUD, Tombstone Deletion, and Compaction

Insertion appends a record at the next free position: each field value is encoded (a previously unseen value extends the dictionary and adds an empty bit vector), the three bits are set, the columns grow by one element, and the cached cardinalities are incremented — amortised O(1). Reading by identifier is a single bounds-and-deleted check followed by a constant-time decode. Update clears the bit of the old value, sets the bit of the new value, adjusts the codes and cardinalities, and is O(1). Deletion is a tombstone: it sets one bit in the deleted bitmap and increments a counter, in O(1), leaving the data in place. Compaction physically removes tombstoned records by rebuilding the columns over the live records, renumbering them so that the offset = id − 1 invariant holds again, and rebuilding the bitmaps — an O(n) operation run only when desired.

3.6 Persistence: Binary Snapshot

To avoid re-parsing the source on every start, the system can write a compact binary snapshot containing the coded columns, the year column, the dictionaries, and the deleted bitmap, prefixed by a small header. Loading a snapshot reads these arrays directly and rebuilds the bitmaps and sorted dictionaries in memory, with no CSV parsing. At one million records a snapshot is written in about 1.2 ms and loaded in about 6.3 ms.

4. Performance Evaluation and Scalability Analysis

4.1 Methodology

We measured the first 10,000, 100,000, and 1,000,000 rows of the dataset. For each scale we recorded load time (memory mapping, parallel parsing, and index construction), the total resident set size (RSS) reported by the operating system, and the size of the index and columns alone. Query latency is reported as the median of 200 repetitions. All measurements use the release build (-O3 -march=native -flto -pthread); resident memory is read after the temporary mapping is released, so that the mapped source file does not inflate the figure.

4.2 Results

Table 3 reports the measurements, and Figures 2 and 3 visualise them. Loading one million records, including index construction, takes 41 ms, and the system settles at about 28 MB of resident memory, of which roughly 23 MB is the index and columns. Point look-ups and counts complete in about 25 ns. A single-field query such as query_category takes 5 µs at one million records, a compound query 8.8 µs, and a prefix query that matches 358,313 records 90 µs. Every query type stays in the microsecond range, comfortably beating the millisecond target.

Table 3. Time, memory, and median query latency at 10K / 100K / 1M records.

Measurement	10K	100K	1M
Load (mmap + parallel + index)	4.0 ms	5.3 ms	41 ms
Resident memory (RSS)	4.7 MB	6.9 MB	28 MB
Index + columns only	0.23 MB	2.3 MB	23 MB
get_by_id	0.047 µs	0.024 µs	0.025 µs
count (cached)	0.033 µs	0.026 µs	0.024 µs
query_category	0.16 µs	0.44 µs	5.0 µs
query_author	0.20 µs	0.60 µs	5.0 µs
compound (word-wise AND)	0.29 µs	0.93 µs	8.8 µs
prefix “The” (358,313 matches @1M)	1.1 µs	9.4 µs	90 µs
snapshot save / load	—	—	1.2 / 6.3 ms

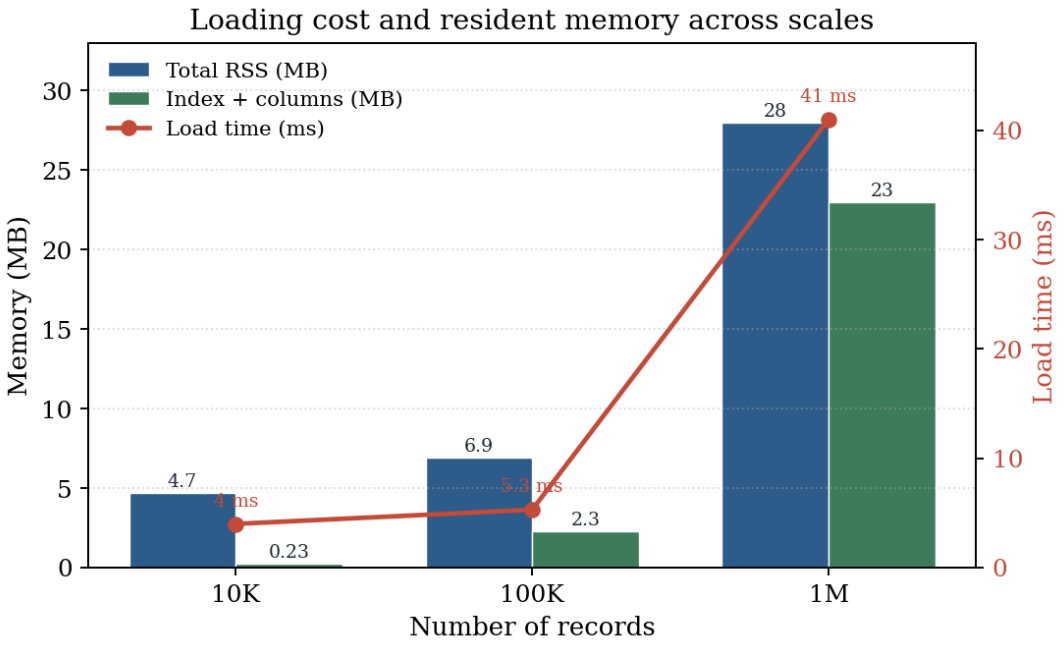

Figure 2. Loading time and resident memory as the dataset grows from 10K to 1M records; both grow roughly linearly.

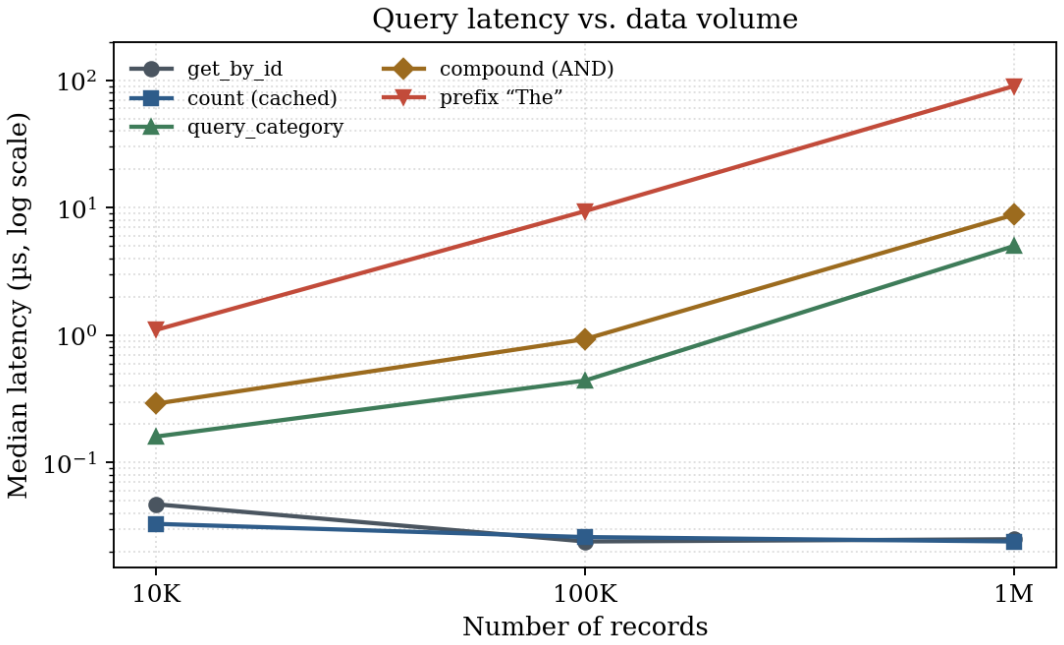

Figure 3. Median query latency versus data volume (log scale). Point access and counting stay flat; only set-returning queries grow.

4.3 Scalability Analysis (10K ↔ 1M)

Table 4 summarises how each cost behaves as the data grows by two orders of magnitude. Loading time and memory grow roughly linearly, because the work and the storage per record are constant. Point access by identifier is constant — about 25 ns at every scale — because it is a single computed memory access with no search. Counting is likewise constant, since it reads

a cached cardinality and does not even need a population count. Set-returning queries grow with the size of the result rather than the size of the dataset: their cost is the cost of scanning the result bit vector and materialising matches, which is why query_category rises from 0.16 µs at 10K to 5 µs at 1M while point access does not move. Compound queries remain near-constant in microseconds because their dominant cost is a word-wise AND over n / 64 machine words, which modern hardware processes at roughly ten billion words per second. Figure 3 makes the distinction visible: the point-access and count lines are flat near the bottom of the log scale, the single-field and compound lines rise gently, and only the prefix query — which unions many bit vectors and returns hundreds of thousands of records — rises steeply. The central message is that the index removes the dataset size from the cost of a query; what remains is the unavoidable cost of producing the answer.

Table 4. How each cost scales from 10K to 1M records.

Metric	Behaviour (10K → 1M)	Reason
Load time and memory	≈ linear	constant work and storage per record
get_by_id	constant (≈ 25 ns)	single computed memory access, no search
count	constant	cached cardinality; no popcount needed
Field query (materialised)	scales with result size	cost is scanning the result bit vector
Compound (AND)	near-constant µs	word-wise AND over n / 64 words (AVX2)

5. Engineering Challenges

The most instructive problem appeared only when loading a subset of the data. While scanning line offsets, the scan stopped as soon as it had collected the requested number of lines but did not record where the last line ended. With no end sentinel, the last record's view extended to the end of the mapped file (about 60 MB), which both touched far more memory than necessary — inflating resident memory to 178 MB instead of 28 MB — and corrupted the final record (at 10K it produced a spurious 79th title). The defect was invisible when loading the whole file, because there the last line genuinely ends at end-of-file. We caught it through an isolated test in which the number of distinct titles came out as 79 rather than 78, and fixed it by recording an explicit end sentinel after the scan. The lesson is that correctness must be demonstrated on the subset path, not only on the full-data path.

Three further issues are worth noting. Sharing the dictionary safely across threads risked a data race during parallel parsing; the two-pass design avoids it by first building thread-local value sets, then merging and sorting them into a deterministic dictionary, after which the second pass writes to disjoint index ranges without locks. Measuring memory accurately required releasing the memory-mapped file before reading resident size, since the mapped source otherwise inflates the figure during loading. Keeping the cached cardinalities consistent required updating them by hand on every insert, update, and delete, and correcting counts in the presence of tombstones by subtracting the deleted bitmap. Finally, the single-byte field codes assume at most 255 distinct values per field; this holds for the given data (78 / 59 / 15), and the loader raises an explicit error rather than overflowing silently should the limit ever be exceeded.

6. Conclusion

CardCatalog meets the assignment's functional requirement — full CRUD over one million records — and exceeds its performance requirement by answering queries in microseconds rather than milliseconds. The design follows directly from two measured properties of the data: very low field cardinality with large result sets, which makes a dense bitmap inverted index the natural secondary structure, and contiguous identifiers, which let us use an implicit primary index that stores no key at all. Columnar, dictionary-encoded storage keeps the footprint near 28 MB at one million records, word-wise bitmap operations keep compound queries near-constant in microseconds, and a binary snapshot restarts the system in milliseconds. The scalability study confirms the intended behaviour across the 10K, 100K, and 1M scales: point access and counting are constant, set-returning queries scale with the answer rather than the dataset, and memory grows linearly. The main limitations are the single-byte code ceiling and the reliance on contiguous identifiers, both of which have clear remedies — wider codes and renumbering on compaction. As required by the assignment, the theoretical design and the comparative 10K/100K/1M analysis were carried out jointly by all group members, and every member is prepared to defend the entire system.

References

[1] M. J. Folk, B. Zoellick, and G. Riccardi, *File Structures: An Object-Oriented Approach with C++*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1998.

[2] C.-Y. Chan and Y. E. Ioannidis, “Bitmap index design and evaluation,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Seattle, WA, USA, Jun. 1998, pp. 355–366, doi: 10.1145/276305.276336.

[3] P. O'Neil and D. Quass, "Improved query performance with variant indexes," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Tucson, AZ, USA, May 1997, pp. 38–49.

[4] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 1–38, Mar. 2006, doi: 10.1145/1132863.1132864.

[5] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with Roaring bitmaps," *Softw., Pract. Exper.*, vol. 46, no. 5, pp. 709–719, May 2016, doi: 10.1002/spe.2325.

[6] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser, "Consistently faster and smaller compressed bitmaps with Roaring," *Softw., Pract. Exper.*, vol. 46, no. 11, pp. 1547–1569, Nov. 2016, doi: 10.1002/spe.2402.

[7] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, "The design and implementation of modern column-oriented database systems," *Found. Trends Databases*, vol. 5, no. 3, pp. 197–280, 2013, doi: 10.1561/1900000024.

[8] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Comput. Surv.*, vol. 38, no. 2, Art. no. 6, Jul. 2006, doi: 10.1145/1132956.1132959.

[9] W. Muła, N. Kurz, and D. Lemire, "Faster population counts using AVX2 instructions," *Comput. J.*, vol. 61, no. 1, pp. 111–120, Jan. 2018, doi: 10.1093/comjnl/bxx046.

[10] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ, USA: Wiley, 2018.

Appendix A — Task Chart

Table 5 lists the development phases and their deliverables. The split of coding work is representative; per the assignment, the theoretical design and the comparative 10K/100K/1M analysis were carried out collectively by all members.

Table 5. Development phases, deliverables, and leads.

Phase	Task	Output	Lead(s)
1	Dataset analysis (cardinality, contiguous ids, comma problem)	analysis notes	All (joint)
2	Dense bitmap layer (set / AND / OR / ANDNOT / popcount / iterate)	bitmap.hpp	Yunus Emre Ayhan
3	Columnar SoA + dictionary encoding + implicit id	columns + get_by_id	Mustafa Yusuf Yaman
4	mmap + two-pass parallel loader	load_from_file	Yunus Emre Ayhan
5	Bitmap inverted index + exact / compound / prefix queries	query_*	Ahmet Başar Alüzüm
6	CRUD + tombstone + compaction	full CRUD	Mustafa Yusuf Yaman
7	Binary snapshot (persistence)	save / load_snapshot	Ahmet Başar Alüzüm
8	Benchmarking (10K/100K/1M) + subset-load bug fix	measurements	All; fix: Yunus Emre
9	Report and interview preparation	this document	All (joint)

Appendix B — Build and Usage

The project depends only on the C++20 standard library and POSIX mmap, with a plain-read fallback on Windows; it is built with make using the flags -O3 -march=native -flto -pthread, under which the bitmap loops vectorise to AVX2. The dataset books_dataset.txt is expected one directory above the build. The available targets are listed in Table 6.

Table 6. Build and run targets.

Command	Action
make	build the release binary
make demo	load, run sample queries and CRUD, and exercise the snapshot path
make bench	measure time, memory, and query latency at 10K / 100K / 1M
make repl	interactive shell, including prefix queries