
Zirka –
the Fastest
Deduplicator

```c
// zirka.c

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <stdbool.h>

#define CHUNK_SIZE 256
#define ORDER 32            // Flatter tree for better disk performance
#define MAX_KEYS (ORDER - 1)
#define NODE_NULL 0

typedef struct {
    uint8_t  keys[MAX_KEYS][CHUNK_SIZE];
    uint64_t offsets[MAX_KEYS];
    uint64_t children[ORDER];
    int32_t  n;
    int32_t  leaf;
} BTreeNode;

// Global State
FILE* fidx = NULL;
uint64_t root_addr = NODE_NULL;
uint64_t nodes_written = 0;
uint64_t total_keys = 0;
uint64_t duplicates_found = 0;

// --- CHECKSUM FOR SANITY CHECK ---
uint32_t calculate_hashsum(const uint8_t* data) {
    uint32_t hash = 0;
    for (int i = 0; i < CHUNK_SIZE; i++) {
        hash = (hash << 1) ^ data[i];
    }
    return hash;
}

// --- DISK I/O HELPERS ---
void read_node(uint64_t addr, BTreeNode* node) {
    if (addr == NODE_NULL) return;
```

```
    fseek(fidx, addr, SEEK_SET);
    fread(node, sizeof(BTreeNode), 1, fidx);
}

void write_node(uint64_t addr, BTreeNode* node) {
    fseek(fidx, addr, SEEK_SET);
    fwrite(node, sizeof(BTreeNode), 1, fidx);
}

uint64_t alloc_node() {
    fseek(fidx, 0, SEEK_END);
    uint64_t addr = ftell(fidx);
    BTreeNode empty = {0};
    fwrite(&empty, sizeof(BTreeNode), 1, fidx);
    nodes_written++;
    return addr;
}

int compare_chunks(const uint8_t* a, const uint8_t* b) {
    return memcmp(a, b, CHUNK_SIZE);
}

// --- B-TREE SPLITTING AND INSERTION ---
typedef struct {
    bool did_split;
    bool already_existed;
    uint8_t promoted_key[CHUNK_SIZE];
    uint64_t promoted_offset;
    uint64_t right_child_addr;
} SplitResult;

SplitResult split_generic(BTreeNode* node, const uint8_t* k, uint64_t offset, uint64_t new_child_addr) {
    SplitResult res = {0};
    uint8_t  t_keys[ORDER][CHUNK_SIZE];
    uint64_t t_offs[ORDER];
    uint64_t t_children[ORDER + 1];

    int i = 0, j = 0;
    bool inserted = false;
    for (i = 0; i < MAX_KEYS; i++) {
```

```
        if (!inserted && compare_chunks(k, node->keys[i]) < 0) {
            memcpy(t_keys[j], k, CHUNK_SIZE);
            t_offs[j] = offset;
            t_children[j] = node->children[i];
            t_children[j+1] = new_child_addr;
            j++; inserted = true;
        }
        memcpy(t_keys[j], node->keys[i], CHUNK_SIZE);
        t_offs[j] = node->offsets[i];
        t_children[j + (inserted ? 1 : 0)] = node->children[i+1];
        if (!inserted) t_children[j] = node->children[i];
        j++;
    }
    if (!inserted) {
        memcpy(t_keys[MAX_KEYS], k, CHUNK_SIZE);
        t_offs[MAX_KEYS] = offset;
        t_children[MAX_KEYS] = node->children[MAX_KEYS];
        t_children[ORDER] = new_child_addr;
    }

    int median = ORDER / 2;
    node->n = median;
    for (i = 0; i < median; i++) {
        memcpy(node->keys[i], t_keys[i], CHUNK_SIZE);
        node->offsets[i] = t_offs[i];
        node->children[i] = t_children[i];
    }
    node->children[median] = t_children[median];

    uint64_t r_addr = alloc_node();
    BTreeNode r = {0};
    r.leaf = node->leaf;
    r.n = MAX_KEYS - median;
    for (i = 0; i < r.n; i++) {
        memcpy(r.keys[i], t_keys[median + 1 + i], CHUNK_SIZE);
        r.offsets[i] = t_offs[median + 1 + i];
        r.children[i] = t_children[median + 1 + i];
    }
    r.children[r.n] = t_children[ORDER];
    write_node(r_addr, &r);
```

```
    res.did_split = true;
    memcpy(res.promoted_key, t_keys[median], CHUNK_SIZE);
    res.promoted_offset = t_offs[median];
    res.right_child_addr = r_addr;
    return res;
}

SplitResult insert_recursive(uint64_t node_addr, const uint8_t* k, uint64_t offset) {
    BTreeNode node;
    read_node(node_addr, &node);
    SplitResult res = {0};

    for(int i = 0; i < node.n; i++) {
        if(compare_chunks(k, node.keys[i]) == 0) {
            res.already_existed = true;
            return res;
        }
    }

    if (node.leaf) {
        if (node.n < MAX_KEYS) {
            int i = node.n - 1;
            while (i >= 0 && compare_chunks(k, node.keys[i]) < 0) {
                memcpy(node.keys[i+1], node.keys[i], CHUNK_SIZE);
                node.offsets[i+1] = node.offsets[i];
                i--;
            }
            memcpy(node.keys[i+1], k, CHUNK_SIZE);
            node.offsets[i+1] = offset;
            node.n++;
            write_node(node_addr, &node);
            return res;
        } else {
            res = split_generic(&node, k, offset, NODE_NULL);
            write_node(node_addr, &node);
            return res;
        }
    } else {
        int i = 0;
```

```
        while (i < node.n && compare_chunks(k, node.keys[i]) > 0) i++;
        SplitResult c_res = insert_recursive(node.children[i], k, offset);
        if (c_res.already_existed || !c_res.did_split) return c_res;

        if (node.n < MAX_KEYS) {
            int j = node.n - 1;
            while (j >= i) {
                memcpy(node.keys[j+1], node.keys[j], CHUNK_SIZE);
                node.offsets[j+1] = node.offsets[j];
                node.children[j+2] = node.children[j+1];
                j--;
            }
            memcpy(node.keys[i], c_res.promoted_key, CHUNK_SIZE);
            node.offsets[i] = c_res.promoted_offset;
            node.children[i+1] = c_res.right_child_addr;
            node.n++;
            write_node(node_addr, &node);
            return (SplitResult){0};
        } else {
            res = split_generic(&node, c_res.promoted_key, c_res.promoted_offset, c_res.right_child_addr);
            write_node(node_addr, &node);
            return res;
        }
    }
}

void insert_root(const uint8_t* k, uint64_t offset) {
    if (root_addr == NODE_NULL) {
        root_addr = alloc_node();
        BTreeNode r = {0}; r.leaf = 1; r.n = 1;
        memcpy(r.keys[0], k, CHUNK_SIZE); r.offsets[0] = offset;
        write_node(root_addr, &r);
        total_keys++;
    } else {
        SplitResult res = insert_recursive(root_addr, k, offset);
        if (!res.already_existed) {
            total_keys++;
            if (res.did_split) {
                uint64_t nr_addr = alloc_node();
                BTreeNode nr = {0}; nr.leaf = 0; nr.n = 1;
```

```
                memcpy(nr.keys[0], res.promoted_key, CHUNK_SIZE);
                nr.offsets[0] = res.promoted_offset;
                nr.children[0] = root_addr; nr.children[1] = res.right_child_addr;
                write_node(nr_addr, &nr);
                root_addr = nr_addr;
            }
        }
    }
}

bool find_and_insert(const uint8_t* k, uint64_t current_offset, uint64_t* orig_offset) {
    uint64_t search_addr = root_addr;
    while (search_addr != NODE_NULL) {
        BTreeNode node;
        read_node(search_addr, &node);
        int i = 0;
        while (i < node.n && compare_chunks(k, node.keys[i]) > 0) i++;
        if (i < node.n && compare_chunks(k, node.keys[i]) == 0) {
            *orig_offset = node.offsets[i];
            duplicates_found++;
            return true;
        }
        if (node.leaf) break;
        search_addr = node.children[i];
    }
    insert_root(k, current_offset);
    return false;
}

// --- MAIN ---
int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    FILE* fin = fopen(argv[1], "rb");
    if (!fin) { perror("Input file error"); return 1; }

    fseek(fin, 0, SEEK_END);
```

```
    double input_size_mb = ftell(fin) / 1048576.0;
    fseek(fin, 0, SEEK_SET);

    char out_name[512];
    snprintf(out_name, sizeof(out_name), "%s.deduplicatedfile", argv[1]);
    FILE* fout = fopen(out_name, "wb");

    fidx = fopen("dedup.index", "w+b");
    uint64_t h_ptr = 0;
    fwrite(&h_ptr, 8, 1, fidx);

    uint8_t window[CHUNK_SIZE];
    uint64_t pos = 0;
    uint8_t magic = 255;

    fprintf(stderr, "Deduplicating with %d bytes granularity...\n", CHUNK_SIZE);

    if (fread(window, 1, CHUNK_SIZE, fin) == CHUNK_SIZE) {
        insert_root(window, 0);
        fwrite(window, 1, CHUNK_SIZE, fout);

        int next_byte;
        while ((next_byte = fgetc(fin)) != EOF) {
            pos++;
            memmove(window, window + 1, CHUNK_SIZE - 1);
            window[CHUNK_SIZE - 1] = (uint8_t)next_byte;

            uint64_t first_seen_at = 0;
            if (find_and_insert(window, pos, &first_seen_at)) {
                uint32_t hash = calculate_hashsum(window);
                fputc(magic, fout);
                fwrite(&first_seen_at, 8, 1, fout);
                fwrite(&hash, 4, 1, fout);

                // JUMPING: Skip 255 bytes, but index them all
                uint8_t skip_buf[CHUNK_SIZE];
                size_t read = fread(skip_buf, 1, CHUNK_SIZE - 1, fin);
                for(size_t j = 0; j < read; j++) {
                    pos++;
                    memmove(window, window + 1, CHUNK_SIZE - 1);
```

```
                window[CHUNK_SIZE - 1] = skip_buf[j];
                insert_root(window, pos);
            }
        } else {
            fputc(window[CHUNK_SIZE - 1], fout);
        }

        if (pos % 100000 == 0) {
            fprintf(stderr, "\rUniques: %lu | Duplicates: %lu", total_keys, duplicates_found);
        }
    }
}

            fprintf(stderr, "\rUniques: %lu | Duplicates: %lu\n", total_keys, duplicates_found);

fseek(fidx, 0, SEEK_SET);
fwrite(&root_addr, 8, 1, fidx);
fseek(fidx, 0, SEEK_END);
double index_size_mb = ftell(fidx) / 1048576.0;

if (input_size_mb > 0) {
    printf("The need for index file is: %.2fMB / %.2fMB =~ %.1fx\n",
           index_size_mb, input_size_mb, index_size_mb / input_size_mb);
}

fclose(fin); fclose(fout); fclose(fidx);
return 0;
}
```

```c
// unzirka.c

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define CHUNK_SIZE 256

// Must match the encoder's hash function exactly
uint32_t calculate_hashsum(const uint8_t* data) {
    uint32_t hash = 0;
    for (int i = 0; i < CHUNK_SIZE; i++) {
        hash = (hash << 1) ^ data[i];
    }
    return hash;
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("Usage: %s <file.deduplicatedfile>\n", argv[0]);
        return 1;
    }

    FILE* fin = fopen(argv[1], "rb");
    if (!fin) { perror("Input error"); return 1; }

    char out_name[512];
    snprintf(out_name, sizeof(out_name), "%s.restored", argv[1]);

    // Open for read/write (w+b) to allow back-referencing reconstructed data
    FILE* fout = fopen(out_name, "w+b");
    if (!fout) { perror("Output error"); fclose(fin); return 1; }

    int c;
    uint8_t buffer[CHUNK_SIZE];
    uint64_t current_out_pos = 0;

    printf("Decoding: %s\n", argv[1]);

    while ((c = fgetc(fin)) != EOF) {
```

```
    if (c == 255) {
        uint64_t source_offset;
        uint32_t expected_hash;

        // Record where the pointer data starts in the input file
        long pointer_start = ftell(fin);

        // Try to read the 8-byte offset and 4-byte hash
        if (fread(&source_offset, 8, 1, fin) == 1 && fread(&expected_hash, 4, 1, fin) == 1) {

            // SANITY CHECK 1: Offset must be historical
            if (source_offset < current_out_pos) {
                long saved_pos = ftell(fout);

                // Go to historical data
                fseek(fout, source_offset, SEEK_SET);
                fread(buffer, 1, CHUNK_SIZE, fout);

                // SANITY CHECK 2: Does the data at that offset match the recorded hash?
                if (calculate_hashsum(buffer) == expected_hash) {
                    fseek(fout, saved_pos, SEEK_SET);
                    fwrite(buffer, 1, CHUNK_SIZE, fout);
                    current_out_pos += CHUNK_SIZE;
                    continue; // Success, pointer resolved
                }
            }
        }

        // If sanity checks fail, the '255' was just a literal byte.
        // Rewind input to just after the '255' and continue.
        fseek(fin, pointer_start, SEEK_SET);
        fseek(fout, 0, SEEK_END);
        fputc(255, fout);
        current_out_pos++;
    } else {
        // Literal byte
        fputc((uint8_t)c, fout);
        current_out_pos++;
    }
```

```
        if (current_out_pos % 1048576 == 0) {
            fprintf(stderr, "\rRestored: %lu MB", current_out_pos / 1048576);
        }
    }

    printf("\nDone. Final Size: %lu bytes\n", current_out_pos);
    fclose(fin);
    fclose(fout);
    return 0;
}
```

```
================================================================================
Zirka v.1 - THE B-TREE SLIDING DEDUPLICATOR (JUMPING EDITION) - README.TXT
================================================================================

THE HIGHLIGHT: ZERO-RAM ARCHITECTURE
Most compression tools are limited by your computer's RAM. If you want to
deduplicate 300GB, they usually require 300GB of RAM.
NOT THIS TOOL. By using a Disk-Based B-Tree, this encoder uses almost
ZERO RAM. It performs "Search & Rescue" operations directly on your
NVMe/SSD. If you can store it, you can dedup it.

THE FOUR EXTREMES (THE HIGHLIGHTS) [

1. THE FASTEST DECOMPRESSOR:
   Optimized specifically for deduplicated .tar files. The decoder doesn't
   waste time with B-Trees or complex math; it just copy-pastes data
   at lightning speeds.

2. THE LOWEST RAM FOOTPRINT:
   While other tools need gigabytes of physical memory (RAM), this tool
   stays at nearly 0MB RAM usage. It uses the hard drive as its brain.

3. THE HIGHEST SSD FOOTPRINT:
   This tool is a "Storage Gladiator." It trades external memory (SSD
   space) for raw power. The .index file is massive because it catalogs
   the input data at each position.

4. THE HIGHEST COMPRESSION RATIO:
   Because we use 256-byte granularity and an "Infinite Lookback" window,
   we can catch duplicates that ZIP or LZ4 class tools (limited to 32KB-64KB
   windows) would never even see.

THE FOUR EXTREMES (THE HIGHLIGHTS) ]

1. WHAT IS THIS?
---------------
A FREE open-source console tool (Linux) written in C by Kaze & Gemini AI.
A high-speed deduplicator that uses a B-Tree "Brain" to remember EVERY
256-byte sequence in a file. It is "Lossless," meaning the restored file
is a perfect bit-for-bit match of the original.
```

2. HOW THE ENCODER WORKS

The encoder slides a 256-byte "viewing window" across your file.

Step A: It checks the B-Tree on your disk: "Is this block a repeat?"
Step B:
   - IF UNIQUE: It writes the raw byte and saves the block to the disk-index.
   - IF DUPLICATE: It writes a 13-byte "Pointer" and jumps ahead 255 bytes.

3. THE "SANITY CHECK" POINTER

To prevent errors if the original file contains the "Magic Byte" (255),
we use a 13-byte secure pointer:
   [1 Byte Magic] + [8 Bytes Offset] + [4 Bytes Hashsum]

The Decoder will only follow a pointer if the data at the destination
perfectly matches the 4-byte Hashsum passcode.

4. THE DECODER

The decoder is a lightweight "Copy-Paster." It reads the instructions
and rebuilds the file by copying from the data it has already restored.

5. PERFORMANCE & STATS

- RAM Usage: Nearly 0. The disk does the heavy lifting.
- Index Size: Expect the .index to be significantly larger than the input.
- Lookback: Infinite. It can find a duplicate from byte 1 even if it's
               currently at byte 300,000,000,000.

```
$ sh zrk.sh Andrzej\ Sapkowski\ -\ The\ Witcher\ Series\ -\ 2007-2018.tar
Deduplicating with 256 bytes granularity...
Uniques: 5887604 | Duplicates: 104
The need for index file is: 2226.08MB / 5.63MB =~ 395.1x

Command being timed: "./zirka Andrzej Sapkowski - The Witcher Series - 2007-2018.tar"
Elapsed (wall clock) time (h:mm:ss or m:ss): 33:16.09
Maximum resident set size (kbytes): 1456
File system inputs: 127,970,104
```

```
File system outputs: 115,155,272

Command being timed: "./unzirka Andrzej Sapkowski - The Witcher Series - 2007-2018.tar"
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.12
Maximum resident set size (kbytes): 1336
File system inputs: 11,480
File system outputs: 11,544

    5,908,480 'Andrzej Sapkowski - The Witcher Series - 2007-2018.tar'
    5,883,364 'Andrzej Sapkowski - The Witcher Series - 2007-2018.tar.deduplicatedfile'
    5,908,480 'Andrzej Sapkowski - The Witcher Series - 2007-2018.tar.restored'
2,334,216,200 dedup.index

6. CREDITS
----------
Gemini AI is cool, it is the main "culprit" for this wondertool.
The 'Zirka' naming is to pay tribute to the beloved kidbook 'Дружков Ю. - Приключения Карандаша и Самоделкина (худ. И. Семёнов)'.
One of the two villains was called Zirka - a petty spy.

зирка
Bulgarian
Etymology: By surface analysis, зи́ра (zíra, “transparent substance”) + -ка (-ka)
Pronunciation: IPA: ['zirkɐ]
Noun
зи́рка • (zírka) m (dialectal)
slit, leak, gap for seeing through
Derived terms:
взи́рка (vzírka) (dialectal)
прозѝрка (prozírka) (dialectal)
зи́ркав (zírkav, “squeezed, squinted”) (of eyes)
Related terms:
зи́ркам (zírkam, “to see through”) (dialectal)
прозо́рец (prozórec, “window”)
https://en.wiktionary.org/wiki/%D0%B7%D0%B8%D1%80%D0%BA%D0%B0

РЕЧНИК НА БЪЛГАРСКИЯ ЕЗИК (ОНЛАЙН)
ЗЙРКА ж. Диал. Цепнатина или отвор в някаква преграда, покрив и под.; пролука, процеп, прозорка, прозирка. Дядо Яначко седеше поприведен в тъмното на одъра; през зирките на вратата светеше - оттатъка в оная, голямата стая. Ил. Волен, НС, 73. Той наблюдаваше през зирките на керемидите как зората се сипваше. Г. Караславов, Избр. съч. Х, 26. През зирките в дъсчената врата на плевнята се виждаше целият двор. В. Андреев, ПП, 8.
https://ibl.bas.bg/rbe/lang/bg/%D0%B7%D0%B8%D1%80%D0%BA%D0%B0/
```

7. DOWNLOAD

https://github.com/Sanmayce/Zirka

2026-Jan-25
Kaze (sanmayce@sanmayce.com)