Buscador Top Tracks (C + Linux)

Índices en disco, búsquedas por `track_id` y por texto, y servidor FIFO

Pablo Bueno Jhoan Smith Yanez Forero

21 de octubre de 2025

1. Resumen

Este documento describe la arquitectura y el funcionamiento de una solución en C para consultar un dataset CSV masivo (~3 GB) del "Top 200 de Spotify". El sistema evita cargar el archivo completo en memoria utilizando dos estructuras auxiliares en disco:

- **Índice por `track_id` (`tracks.idx`)**: *tabla hash de direccionamiento abierto con **linear probing***, implementada con hashing FNV-1a de 64 bits. Permite acceso O(1) promedio.
- **Índice invertido por texto (`nameidx/`)**: tokeniza `track_name` y `artist` (tras normalizar tildes/mayúsculas) y almacena listas de offsets por token.

Las consultas por ID o por palabras recuperan directamente las filas desde offsets del CSV, trabajando siempre en **modo lectura**. Además, se proveen un servidor/cliente FIFO de ejemplo y un programa principal con menú.

2. Dataset y parsers

2.1. CSV *streaming* con offsets

El CSV `merged_data.csv` se recorre en *streaming*. Para cada línea se memoriza el **offset** (posición del inicio de línea) mediante `ftello()`. Las consultas posteriores pueden hacer `fseeko()` al offset exacto sin escanear el archivo.

2.2. Parser que respeta comillas

Se implementó un parser ligero que soporta campos entrecomillados y comillas escapadas . Evita dividir por comas internas del campo. Fragmento representativo:

```c
size_t parse_csv_line(const char *line, char **out, size_t max_fields
    ){
     size_t n=0, L=strlen(line), bi=0; int inq=0;
     char *buf=(char*)malloc(L+1);
     for(size_t i=0;i<L;i++){
          char c=line[i];
          if(c=='"'){
               if(inq && i+1<L && line[i+1]=='"'){ buf[bi++]='"'; i++; }
               else inq=!inq;
          } else if(c==',' && !inq){
               buf[bi]='\0'; out[n++]=strdup(buf); bi=0;
          } else if(c!='\r' && c!='\n') buf[bi++]=c;
     }
     buf[bi]='\0'; out[n++]=strdup(buf);
     free(buf); return n;
}
```

3. Índice por `track_id`: `tracks.idx`

3.1. Formato de archivo y estructuras

- **Header** (`IdxHeader`): magia `ÏDX1TRK"`, capacidad (potencia de 2), columna clave, versión.
- **Slots** (`Slot`): pares `(hash, offset)`. El `hash==0` representa slot vacío.

```
typedef struct {
    char     magic[8];        // "IDX1TRK"
    uint64_t capacity;        // #slots (2^k)
    uint32_t key_col;         // columna de track_id
    uint32_t version;
    uint64_t reserved[3];
} __attribute__((packed)) IdxHeader;

typedef struct {
    uint64_t hash;            // 0 = vac o
    uint64_t offset;          // offset en CSV (inicio de l nea)
} __attribute__((packed)) Slot;
```

3.2. Hash FNV-1a y normalización de estado vacío

Para el hash se utiliza FNV-1a 64 bits por su buena distribución y coste bajo. Reservamos `hash==0` como marca de slot libre; si el valor calculado da 0, lo remapeamos a 1 (evitando confundir "ocupado" con "vacío").

```
static uint64_t fnv1a64(const char *s){
    const uint64_t OFF=14695981039346656037ULL, PR=1099511628211ULL;
    uint64_t h=OFF;
    for(const unsigned char *p=(const unsigned char*)s; *p; ++p){ h
        ^=*p; h*=PR; }
    if (h==0) h=1;                // 0 reservado para "slot vac o"
    return h;
}
```

3.3. Direccionamiento abierto con *linear probing*

La tabla hash es un arreglo de `Slot` de tamaño `capacity` (potencia de 2). El índice inicial se obtiene con `i = hash & (capacity-1)`. Si el slot está ocupado y el hash no coincide, se avanza secuencialmente con **sondeo lineal** hasta encontrar un hueco o el hash objetivo. Esto reduce estructuras auxiliares y mantiene accesos contiguos en memoria (buen *cache locality*).

Inserción (construcción del índice).

```
void insert(Slot *slots, uint64_t cap, uint64_t h, uint64_t off){
    uint64_t i = h & (cap-1);
    while (slots[i].hash != 0){       // ocupado
        i = (i + 1) & (cap-1);        // linear probing
    }
    slots[i].hash = h;
    slots[i].offset = off;
}
```

Búsqueda (tiempo de ejecución).

```
int find(const Slot *slots, uint64_t cap, uint64_t h, uint64_t *
   out_off){
    uint64_t i = h & (cap-1), start = i;
```

```
    for(;;){
        uint64_t sh = slots[i].hash;
        if (sh == 0) return 0;                  // no est
        if (sh == h){ *out_off = slots[i].offset; return 1; }
        i = (i + 1) & (cap-1);
        if (i == start) return 0;               // dio la vuelta
    }
}
```

3.4. Gestión de colisiones y falsos positivos

- **Colisiones de hash**: se resuelven con **linear probing**. El arreglo se dimensiona como potencia de 2 con *load factor* razonable (capacidad $\geq$ 2–4$\times$ número de claves) para mantener clústeres pequeños.
- **Falsos positivos**: aunque el `hash` coincide, **siempre** se valida la clave real: tras saltar al `offset` del CSV, se parsea la línea y se compara el campo `track_id`. Si difiere, se continúa el *probe*.
- **Slot vacío** bien definido: reservando `hash==0`, el final del probe es inequívoco.

3.5. Construcción del archivo `tracks.idx`

Durante el build se recorre el CSV: se toma el `offset` con `ftello()`, se extrae `track_id`, se calcula `hash` y se inserta en el arreglo de slots en memoria. Finalmente se escribe a disco: `IdxHeader` seguido del array `Slot[]` (`O(capacity)`).

3.6. Lookup por ID en ejecución

El binario (o el módulo en `p1-dataProgram.c`) mapea (`mmap`) `tracks.idx` y abre el CSV en `r`. Calcula hash, realiza sondeo lineal hasta hallar coincidencia y luego valida la columna `track_id` en la línea leída desde el CSV con `fseeko()` al offset.

4. Índice invertido por texto: `nameidx/`

4.1. Normalización y tokenización

Se normaliza a minúsculas y se remueven tildes frecuentes en UTF-8 (á/é/í/ó/ú/ü/ñ). Luego se tokeniza por separadores no alfanuméricos y se aplica **unique por línea** para evitar múltiples apariciones del mismo token en una fila.

4.2. Estructura y compactación por buckets

El índice se divide en **256 buckets** (`h & 0xFF`). En construcción se escriben pares `(hash, offset)` en `bXX.tmp`. Por bucket se ordena por `(hash, offset)` y se compacta a `bXX.idx` en bloques contiguos:

`[hash][df:uint32][pad:uint32][df * offset:uint64]`

donde `df` es el tamaño de la posting list para ese token. Se eliminan offsets duplicados.

4.3. Consulta por 1–3 palabras (AND)

Para cada término se carga su posting list (escaneo secuencial del archivo del bucket), y luego se intersecan listas ordenadas en `O(n)`. Los offsets resultantes se usan para leer directamente las líneas del CSV.

5. Servidor y cliente FIFO (opcional)

5.1. Protocolo y robustez

```
GET_ID <track_id> <fifo_respuesta>\n
QUIT\n
```

El servidor mapea el índice, abre el CSV y atiende solicitudes desde una FIFO. Se valida siempre el `track_id` tras saltar al offset. Para robustez: verificación del tipo FIFO, recreación si falta, y `write_full()` para escrituras completas.

6. Programa principal con menú: `p1-dataProgram.c`

6.1. Flujo de uso

1. Solicita rutas de `merged_data.csv`, `tracks.idx` y `nameidx/`.
2. Carga el header del CSV para detectar índices de columnas y emitir salida compacta.
3. Menú: (1) ID exacto; (2–3) palabras (AND) en nombre/artista; (4) ejecutar; (5) salir.

6.2. Salida compacta

Para evitar ruido (p.ej., `available_markets`), el programa imprime sólo: `track_id | track_name | artist | date | region`. Esto se logra parseando la línea y seleccionando únicamente esas columnas.

7. Makefile y entrega

7.1. Objetivos principales

- `make`: compila `p1-dataProgram`.
- `make indexes`: genera `tracks.idx` y `nameidx/`.
- `make fetch-data`: descarga el CSV desde OneDrive/SharePoint con URL directa.
- `make clean`: limpia binarios.
- `make dist LASTNAME1=... LASTNAME2=...`: empaqueta para entrega.

8. Rendimiento, memoria y validaciones

- **Por ID**: O(1) promedio; un único salto al CSV. RAM $<$10 MB más el mapeo del índice.
- **Por texto**: Depende del tamaño de postings; intersección lineal y pocos saltos al CSV.
- **Sincronía**: los offsets son válidos para el CSV con el que se construyó el índice; si el CSV cambia, reconstruir.
- **Validación**: siempre se compara el valor real de `track_id` tras el hash para evitar falsos positivos.

9. Conclusión

El enfoque "CSV + índices en disco" permite consultas rápidas sobre archivos de varios GB con uso mínimo de memoria. El índice hash por `track_id` brinda acceso O(1) y maneja colisiones mediante *linear probing* con validación de clave; el índice invertido habilita búsquedas por nombre/artista con intersección eficiente, manteniendo todo el pipeline en C, portable y sin dependencias externas.

Repositorio: https://github.com/Nutor30P/hash_table