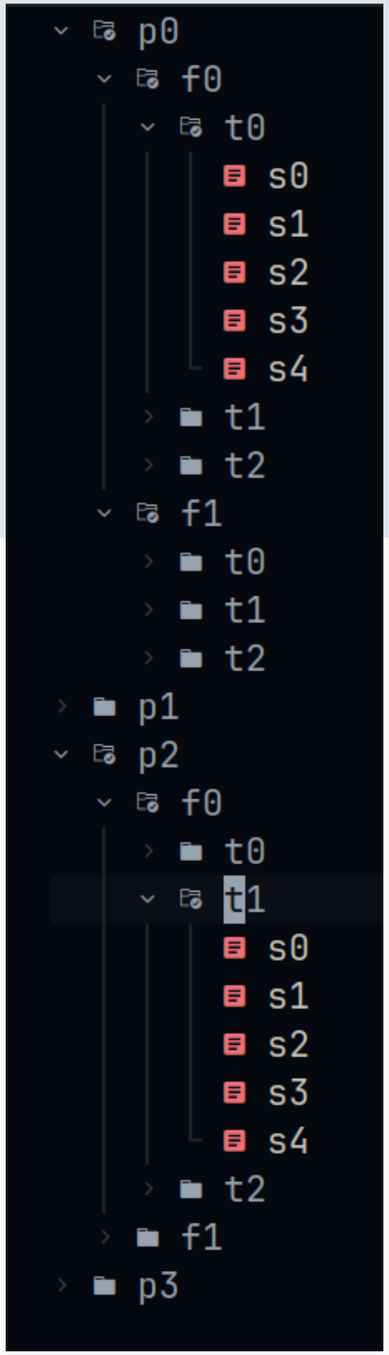
Universidad Nacional de San Agustín

Sistema Gestor de Base de Datos

Alberto Gabriel Torres Ara

BASE DE DATOS 2



Estructura del disco

```
all23tor@agtavictus:~/projects/UNSA/BBDD2/disco$ ./build/disco
El disco aún no existe, se procederá a su creación

Cantidad de platos: 4
Cantidad de pistas por superficie: 3
Cantidad de sectores por pista: 5
Tamaño de cada sector en bytes: 100000
Número de sectores por bloque: 8
"/home/all23tor/projects/UNSA/BBDD2/disco/disk"
└ "/home/all23tor/projects/UNSA/BBDD2/disco/disk/p0"
   └ "/home/all23tor/projects/UNSA/BBDD2/disco/disk/p0/f0"
      └ "/home/all23tor/projects/UNSA/BBDD2/disco/disk/p0/f0/t0"
         └ "/home/all23tor/projects/UNSA/BBDD2/disco/disk/p0/f0/t0/s0"
         └ "/home/all23tor/projects/UNSA/BBDD2/disco/disk/p0/f0/t0/s1"
         └ "/home/all23tor/projects/UNSA/BBDD2/disco/disk/p0/f0/t0/s2"
         └ "/home/all23tor/projects/UNSA/BBDD2/disco/disk/p0/f0/t0/s3"
         └ "/home/all23tor/projects/UNSA/BBDD2/disco/disk/p0/f0/t0/s4"
      └ "/home/all23tor/projects/UNSA/BBDD2/disco/disk/p0/f0/t1"
```

Si el disco ya existe, leer la información del disco del primer sector

```
inline void read_disk_info() {
  auto disk = fs::current_path() / "disk";
  auto first_path = disk / "p0" / "f0" / "t0" / "s0";
  ip::file_mapping first_file(first_path.c_str(), ip::mode_t::read_only);
  ip::mapped_region first_map(first_file, ip::mode_t::read_only);
  auto data = reinterpret_cast<char*>(first_map.get_address());
  globalDiskInfo = reinterpret_cast<DiskInfo&>(*data);
}
```

```
static inline struct DiskInfo {
  int plates;
  int tracks;
  int sectors;
  int bytes;
  int block_size;
} globalDiskInfo = {-1, -1, -1, -1, -1};
```

Si el disco aún no existe, crearlo y escribir la información en el primer sector

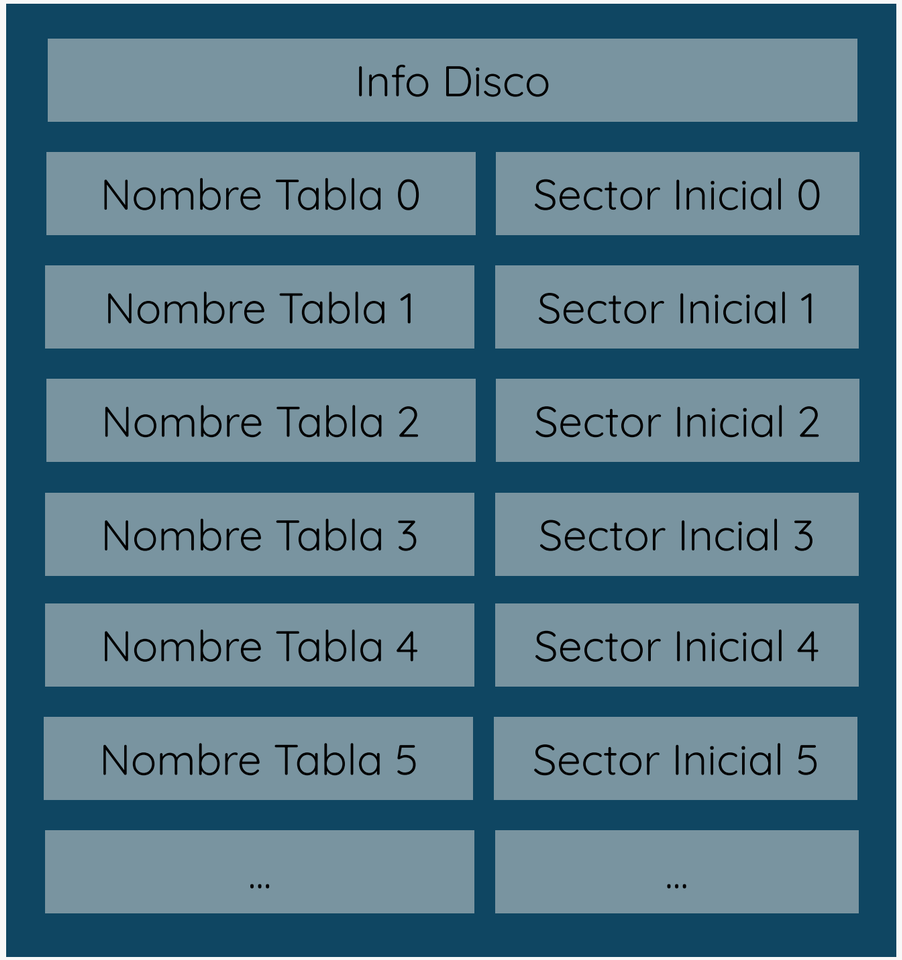
```
  auto first_path = disk_path / "p0" / "f0" / "t0" / "s0";
  ip::file_mapping first_file(first_path.c_str(), ip::mode_t::read_write);
  ip::mapped_region first_map(first_file, ip::mode_t::read_write);
  auto data = reinterpret_cast<char*>(first_map.get_address());
  globalDiskInfo = diskInfo;
  reinterpret_cast<DiskInfo&>(*data) = globalDiskInfo;

  fs::create_directory(blocks_path);
}
```

Almacenamiento

Registros de longitud fija

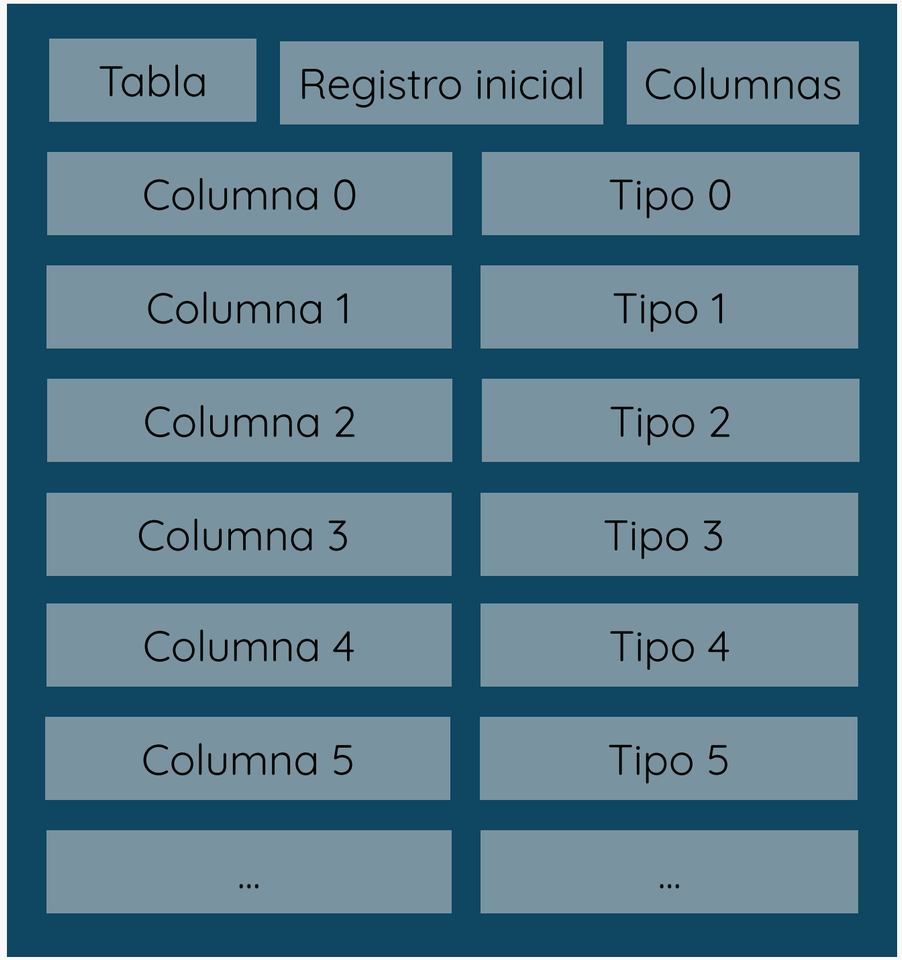
```
90 Address* write_header(const std::string& table_name,
91                       const std::vector<Db::Column>& columns) {
92   auto first_data = CowBlock::load_writeable_sector({0}) + sizeof(DiskInfo);
93   auto tables = reinterpret_cast<Table*>(first_data);
94   int table_idx = 0;
95
96   while (tables[table_idx].name[0] != '\0')
97     table_idx++;
98
99   auto& table = tables[table_idx];
100  for (int i = 0; i < table_name.size(); i++)
101    table.name[i] = table_name[i];
102  for (int i = table_name.size(); i < table.name.size(); i++)
103    table.name[i] = '\0';
104
105  auto header_sector = request_empty_sector();
106  auto header_data = CowBlock::load_writeable_sector(header_sector);
107  table.sector = header_sector;
108
109  auto records_start = reinterpret_cast<Address*>(header_data);
110  *records_start = NullAddress;
111  header_data += sizeof(Address);
112
```

Esquema



Almacenamiento

Información en el esquema



Registros de longitud fija

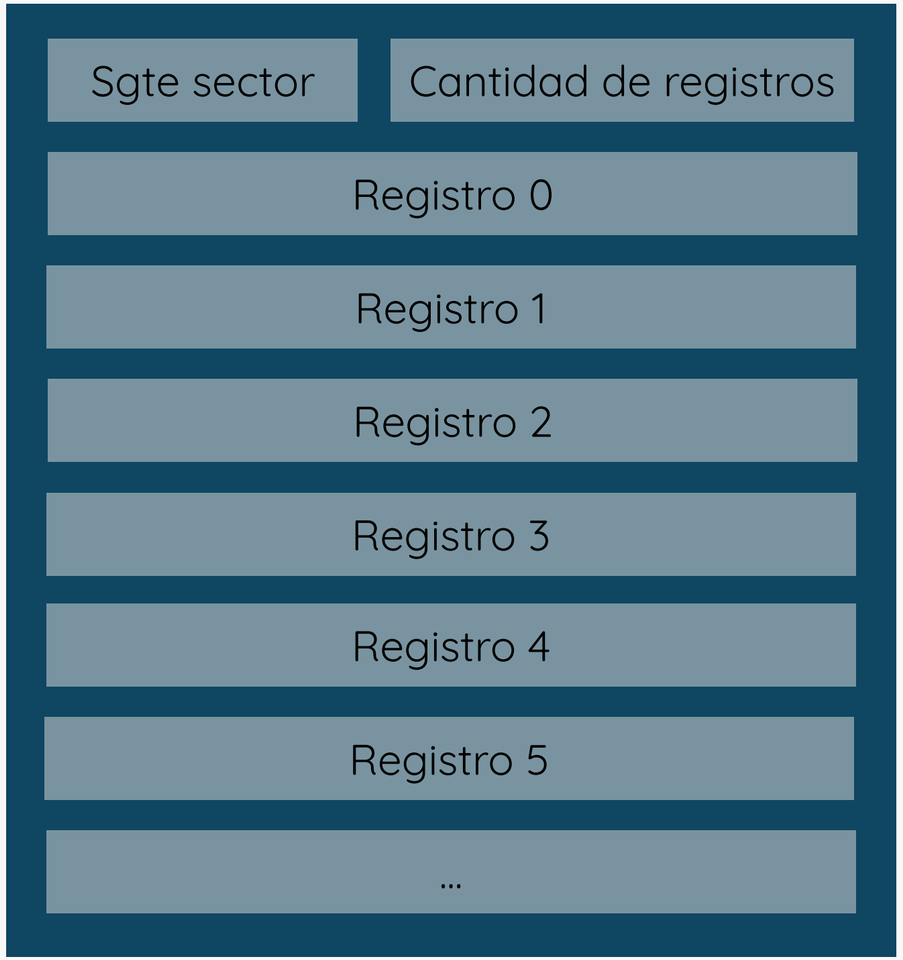
```
105   auto header_sector = request_empty_sector();
106   auto header_data = CowBlock::load_writeable_sector(header_sector);
107   table.sector = header_sector;
108
109   auto records_start = reinterpret_cast<Address*>(header_data);
110   *records_start = NullAddress;
111   header_data += sizeof(Address);
112
113   int column_size = columns.size();
114   for (char c : pun_cast(column_size))
115   |  *(header_data++) = c;
116
117   for (auto& column : columns)
118   |  for (char c : pun_cast(column))
119   |  |  *(header_data++) = c;
120
121   return records_start;
122 }
```

Almacenamiento

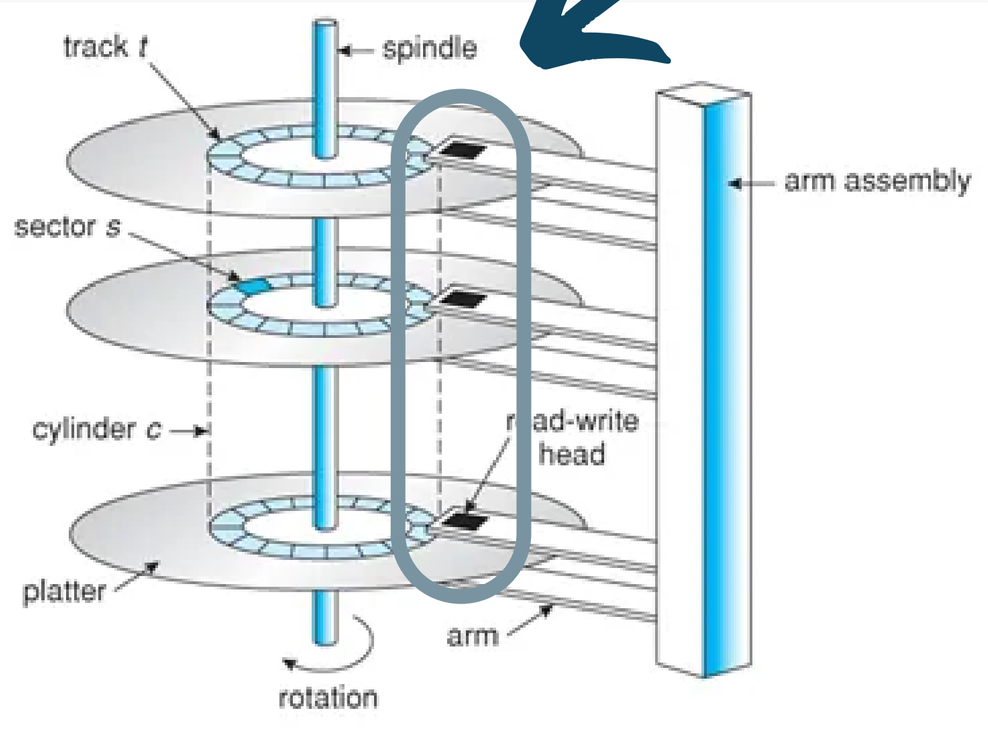
Registros de longitud fija

```
int recods_per_sector =
    (globalDiskInfo.bytes - sizeof(Address) - sizeof(int)) / record_size;
*records_start = request_empty_sector();
auto record_data = CowBlock::load_writeable_sector(*records_start);
auto next_sector = reinterpret_cast<Address*>(record_data);
record_data += sizeof(Address);
auto record_count = reinterpret_cast<int*>(record_data);
record_data += sizeof(int);
*next_sector = NullAddress;
*record_count = 0;

while (std::getline(file, line)) {
  if (*record_count == recods_per_sector) {
    *next_sector = request_empty_sector();
    record_data = CowBlock::load_writeable_sector(*next_sector);
    next_sector = reinterpret_cast<Address*>(record_data);
    record_data += sizeof(Address);
    record_count = reinterpret_cast<int*>(record_data);
    record_data += sizeof(int);
    *next_sector = NullAddress;
    *record_count = 0;
  }
```

Sector en uso



Bloques

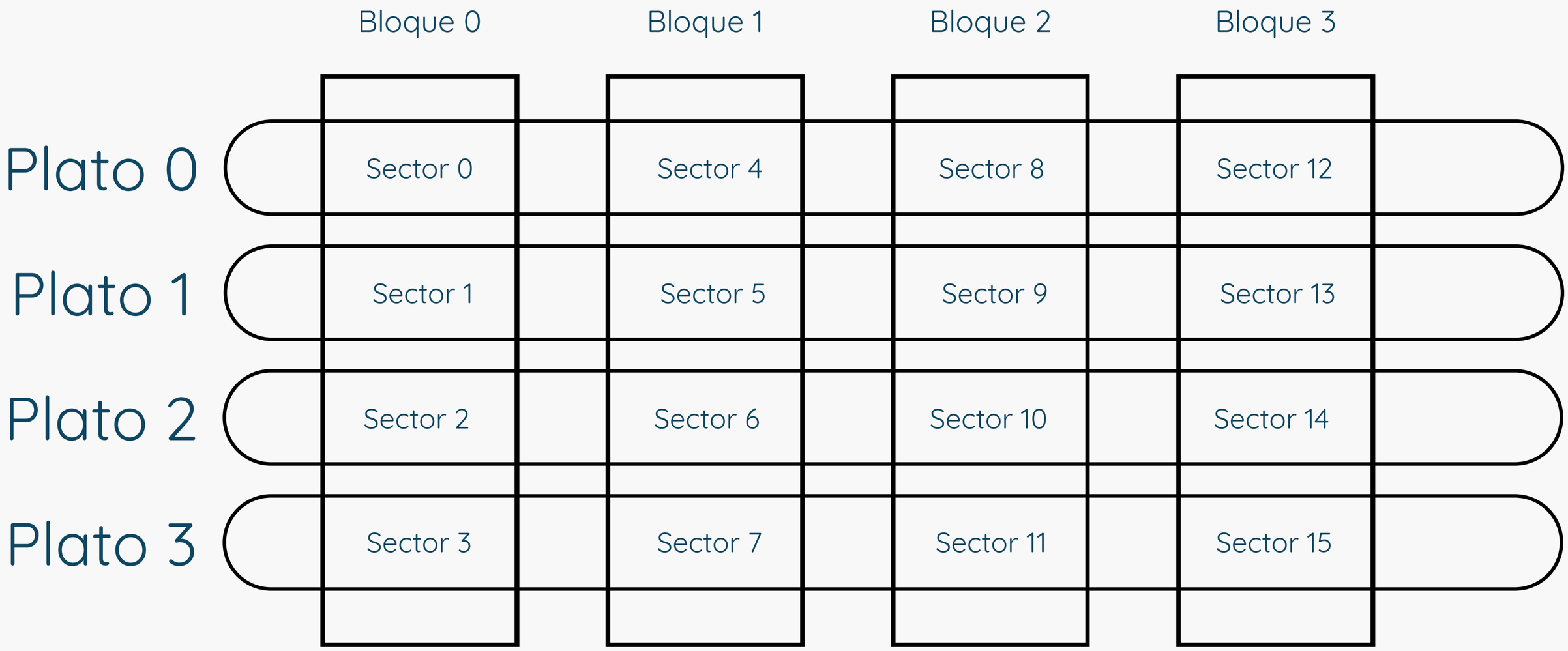


Bloques en vertical

Minimiza el movimiento del brazo

Facilita la lógica de solicitar sectores vacíos

Direccionamiento de bloques



De lugar físico a dirección lógica y viceversa

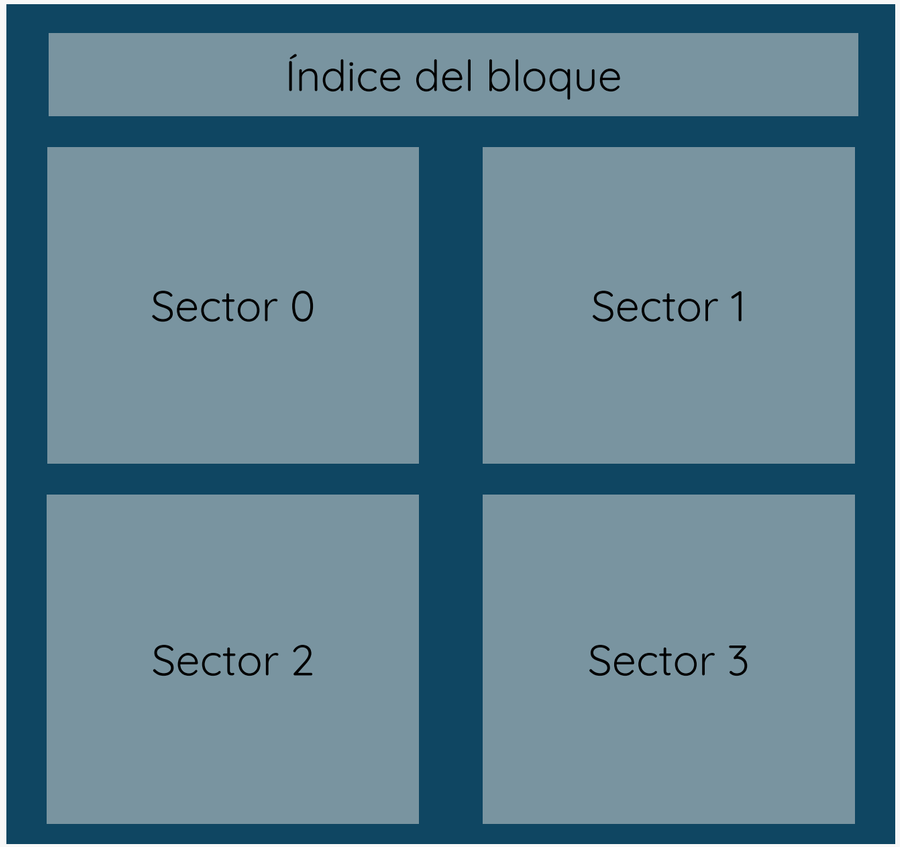
```
struct Address {
  int address;

  bool operator==(const Address&) const = default;
  static Address from_tree(int plate, int surface, int track, int sector) {
    return {plate +
            globalDiskInfo.plates *
                (sector + globalDiskInfo.sectors *
                              (track + globalDiskInfo.tracks * surface))};
  }
```

```
fs::path to_path() const {
  int address = this->address;
  auto plate = address % globalDiskInfo.plates;
  address /= globalDiskInfo.plates;
  auto sector = address % globalDiskInfo.sectors;
  address /= globalDiskInfo.sectors;
  auto track = address % globalDiskInfo.tracks;
  address /= globalDiskInfo.tracks;
  auto surface = address % 2;

  return disk_path / ('p' + std::to_string(plate)) /
        ('f' + std::to_string(surface)) / ('t' + std::to_string(track)) /
        ('s' + std::to_string(sector));
}
```

Bloque Copy on Write

```
class CowBlock {
  mutable int written = false;
  ip::mapped_region block_map;

  static inline std::set<CowBlock, std::less<>> loaded_blocks;
```

Estructura



De índice de bloque a direcciones de sectores

```
int block_idx = *reinterpret_cast<const int*>((data() - sizeof(int)));
for (int sector = 0; sector < globalDiskInfo.block_size; sector++) {
  auto sector_data = data() + sector * globalDiskInfo.bytes;
  Address sector_address = {block_idx * globalDiskInfo.block_size + sector};
```

Solo sobreescribimos los sectores si escribimos en el bloque

```
~CowBlock() {
  if (!written)
    return;
```

Guardamos los bloques cargados para evitar volver a cargar el mismo sector

```
static const char* load_sector(Address sector_address) {
  int block = sector_address.address / globalDiskInfo.block_size;
  auto it = loaded_blocks.find(block);
  if (it == loaded_blocks.end())
    std::tie(it, std::ignore) = loaded_blocks.insert(CowBlock(block));
  return it->data() + globalDiskInfo.bytes * (sector_address.address %
                                              globalDiskInfo.block_size);
}
```

Referencias

1. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2011). Database system concepts (6th ed.).
2. McGraw-Hill.Garcia-Molina, H., Ullman, J. D., & Widom, J. (2008). Database systems: The complete book (2nd ed.). Pearson Education.
3. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Patrones de diseño: Elementos de software orientado a objetos reutilizable (1.ª ed.). Addison-Wesley Iberoamericana.

Muchas gracias