



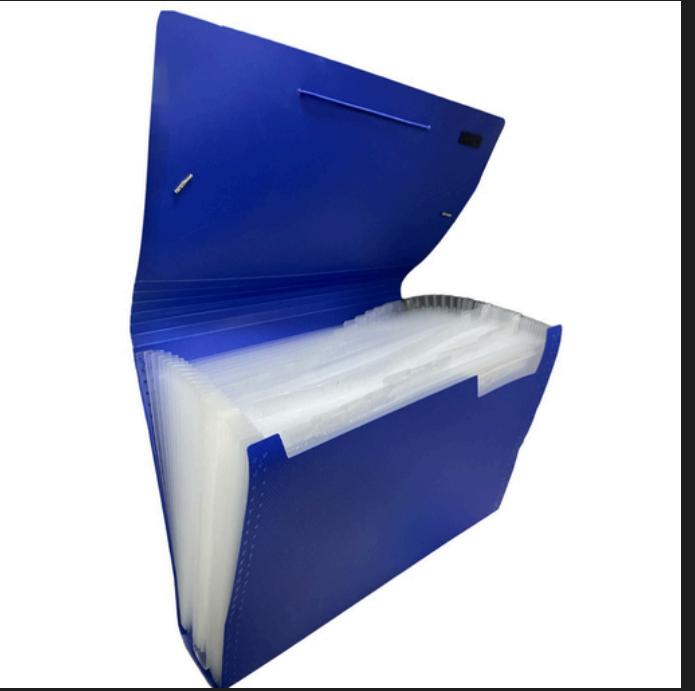
# BUFFER MANAGER

## Integrantes

- Arthur Patrick Meza Pareja
- Alberto Gabriel Torres Ara



# Disco



Lento

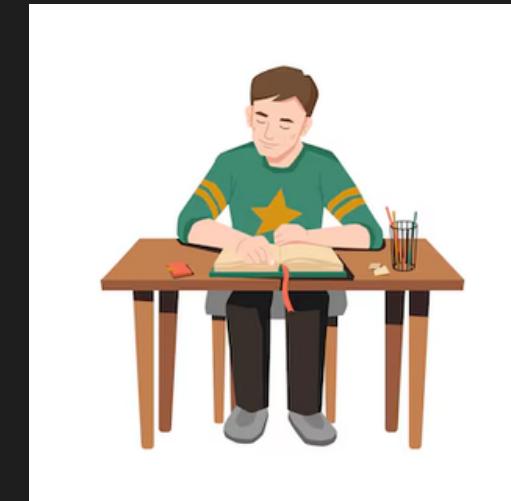


105785433

# M. Ram



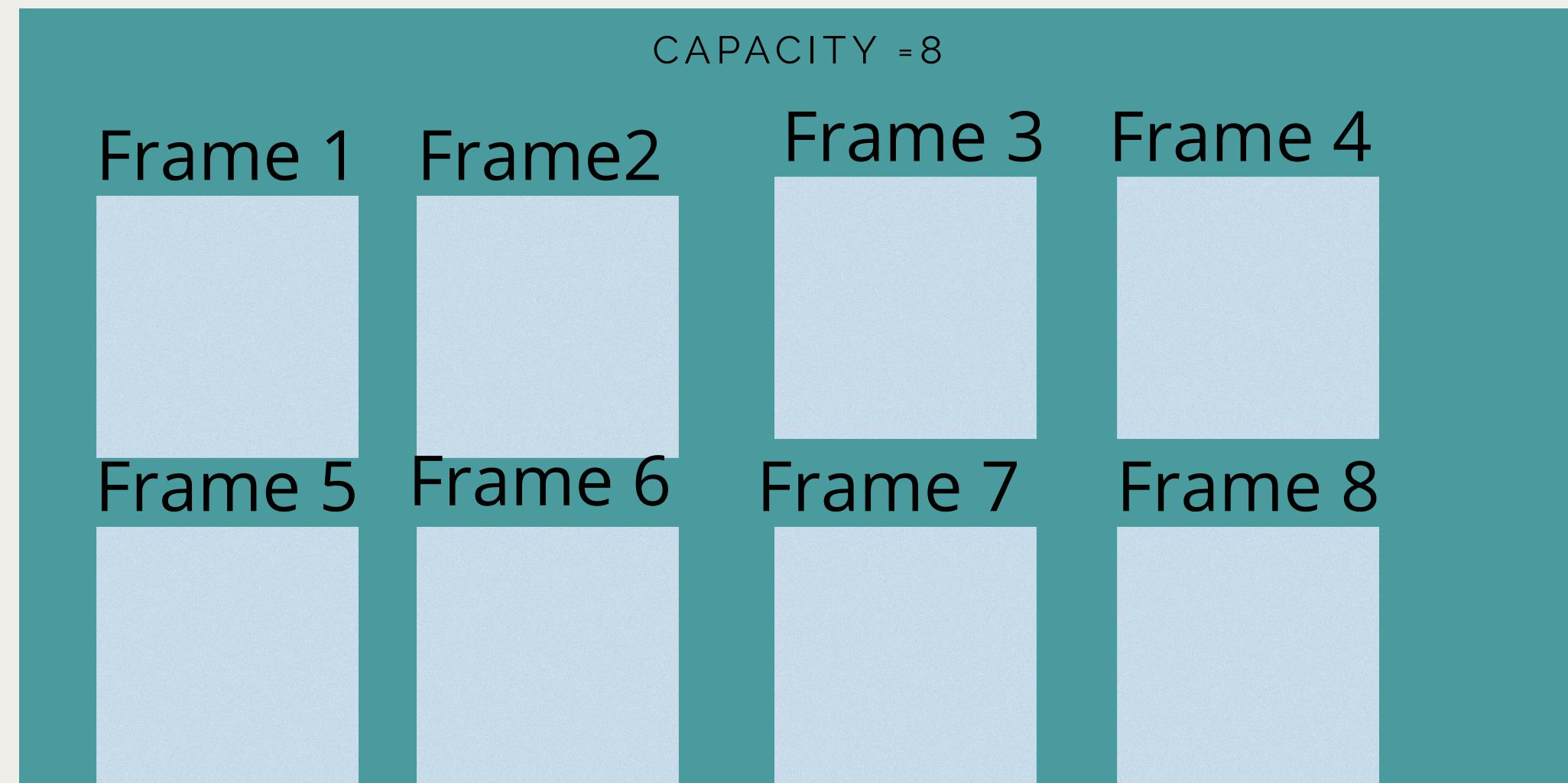
Rapido





# Estructura del Buffer Manager

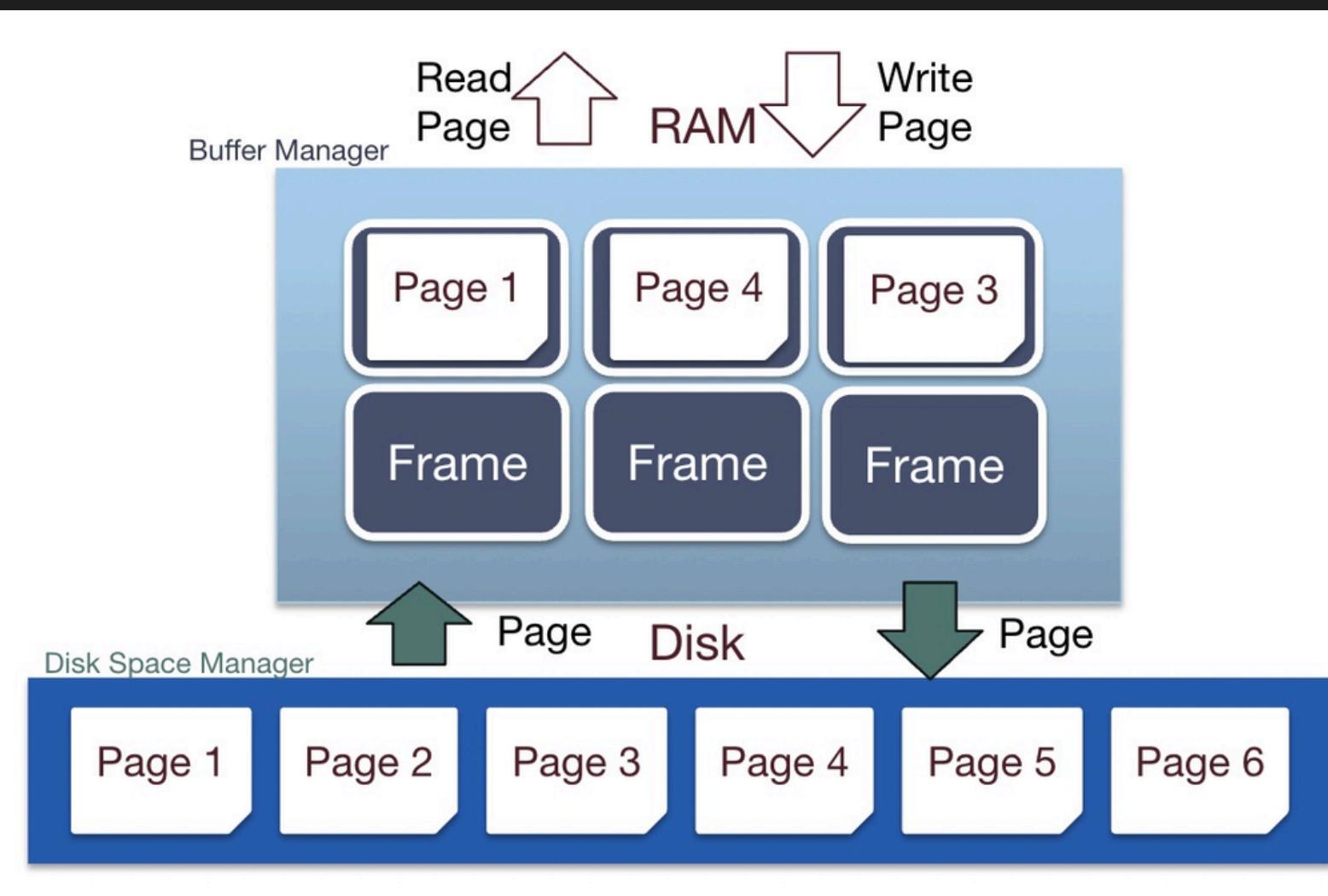
```
const int capacity;  
std::map<int, Frame> frames;  
std::list<int> lru;
```





&gt;&gt;&gt;

# Frame



Frame: Representa un bloque de datos (página) cargado en la memoria del buffer.

- content: El contenido real de la página.
- dirty\_bit: Indica si el contenido del frame ha sido modificado y necesita ser escrito de vuelta al disco.
- pin\_count: Un contador que evita que un frame sea desalojado mientras está siendo utilizado activamente

# Carga de Sectores

>>>

**load\_sector y**

**load\_writeable\_sector**

- Traduce la dirección del sector a un ID de bloque.
- Verifica si el bloque ya está en el pool (cache hit).
- Si es un hit, se actualiza su posición en la lista LRU y se retorna el puntero a los datos.
- Si no es un hit (cache miss):
  - Si hay espacio, se carga el bloque del disco y se agrega al pool y LRU.
  - Si no hay espacio, se busca el frame menos recientemente usado (LRU) que no esté "pineado".
    - Si el frame LRU está dirty, se escribe de vuelta al disco.
    - Se reemplaza el frame LRU con el nuevo bloque.
- `load_writeable_sector` adicionalmente setea el `dirty_bit` a true.

```
> > REQUEST 1 L  
Adding 1  
ID    L/W    DIRTY   PINS   LRU  
1      L       0        0       0
```

```
Total access 1 Hits 0  
Hit rate 0%  
> REQUEST 2 L
```

```
Adding 2  
ID    L/W    DIRTY   PINS   LRU  
1      L       0        0       0  
2      L       0        0       1
```

```
Total access 2 Hits 0  
Hit rate 0%  
> REQUEST 3 L
```

```
Adding 3  
ID    L/W    DIRTY   PINS   LRU  
1      L       0        0       0  
2      L       0        0       1  
3      L       0        0       2
```

```
Total access 3 Hits 0  
Hit rate 0%  
> REQUEST 4 W  
Adding 4
```



- Traduce la dirección del sector a un ID de bloque.
- Verifica si el bloque ya está en el pool (cache hit).
  - Si es un hit, se actualiza su posición en la lista LRU y se retorna el puntero a los datos.
- Si no es un hit (cache miss):
  - Si hay espacio, se carga el bloque del disco y se agrega al pool y LRU.
  - Si no hay espacio, se busca el frame menos recientemente usado (LRU) que no esté "pineado".
    - Si el frame LRU está dirty, se escribe de vuelta al disco.
    - Se reemplaza el frame LRU con el nuevo bloque.
- load\_writeable\_sector adicionalmente setea el dirty\_bit a true.

Total access 4 Hits 1  
Hit rate 25%  
> REQUEST 4 L

Adding 4				
ID	L/W	DIRTY	PINS	LRU
1	L	0	0	0
2	L	0	0	1
3	L	0	0	2
4	L	0	0	3

Total access 5 Hits 1  
Hit rate 20%  
> REQUEST 5 L

Erasing 1				
Replacing with 5				
ID	L/W	DIRTY	PINS	LRU
2	L	0	0	0
3	L	0	0	1
4	L	0	0	2
5	L	0	0	3

Total access 6 Hits 1  
Hit rate 16.6667%

# Pinning (pin and unpin) >>>



- `pin(Address sector_address)`: Incrementa el `pin_count` de un frame, evitando su desalojo.
- `unpin(Address sector_address)`: Decrementa el `pin_count`, permitiendo su posible desalojo.

```
> REQUEST 5 W
Ignoring 1
Ignoring 2
Erasing 3
Replacing with 5
ID    L/W    DIRTY   PINS   LRU
1     L      0        2       0
2     L      0        1       1
4     L      0        0       2
5     L      0        0       3
```

Total access 5 Hits 0

Hit rate 0%

```
> UNPIN 1
Unpinning 1
ID    L/W    DIRTY   PINS   LRU
1     L      0        1       0
2     L      0        1       1
4     L      0        0       2
5     W      1        0       3
```

Total access 5 Hits 0

Hit rate 0%

```
> UNPIN 1
Unpinning 1
ID    L/W    DIRTY   PINS   LRU
1     L      0        0       0
2     L      0        1       1
4     L      0        0       2
5     W      1        0       3
```

Total access 5 Hits 0

Hit rate 0%



# Ciclo de vida del Buffer Manager»

- Inicialización: Se define la capacity del buffer pool.
- Operación: Las aplicaciones solicitan sectores (lectura/escritura) y pinean/despinean frames según sea necesario.
- Destrucción (~BufferManager()): Antes de que el BufferManager sea destruido, todos los frames con el dirty\_bit seteado son escritos de vuelta al disco para asegurar la persistencia de los datos modificados.

```
52 class BufferManager {  
53     int hits = 0;  
54     int total_access = 0;  
55     const int capacity;  
56     std::unordered_map<int, Frame> pool;  
57     std::list<int> lru;  
58  
59 public:  
60     // Inicializamos con la capacidad (que se mantendrá constante)  
61     BufferManager(int _capacity) : capacity{_capacity} {}  
62     // Al momento de dejar de utilizar memoria, escribimos todas los frames  
63     // con el dirty bit seteado.  
64     ~BufferManager() {  
65         for (auto& [frame_id, frame] : pool) {  
66             if (frame.dirty_bit) {  
67                 for (int sector = 0; sector < globalDiskInfo.block_size; sector++) {  
68                     auto sector_data = frame.data() + sector * globalDiskInfo.bytes;  
69                     Address sector_address = {frame_id * globalDiskInfo.block_size +  
70                                         sector};  
71                     std::ofstream sector_file = sector_address.to_path();  
72                     sector_file.write(sector_data, globalDiskInfo.bytes);  
73                 }  
74             }  
75         }  
76     }
```

# Uso del buffer manager

>>>

```
151 Address* write_table_header(const std::string& table_name,
152                             const std::vector<Db::Column>& columns,
153                             BufferManager& buffer_manager) {
154     auto first_data =
155         buffer_manager.load_writeable_sector({0}) + sizeof(DiskInfo);
156     auto tables = reinterpret_cast<Table*>(first_data);
157     int table_idx = 0;
158
159     while (tables[table_idx].name[0] != '\0')
160         table_idx++;
161
162     auto& table = tables[table_idx];
163     for (int i = 0; i < table_name.size(); i++)
164         table.name[i] = table_name[i];
165     for (int i = table_name.size(); i < table.name.size(); i++)
166         table.name[i] = '\0';
167
168     auto header_sector = request_empty_sector(buffer_manager);
169     auto header_data = buffer_manager.load_writeable_sector(header_sector);
170     buffer_manager.pin(header_sector);
171     table.sector = header_sector;
172 }
```

```
399     write_table_data(file, next_address, header_info.columns,
400                      header_info.columns_size, records_per_sector,
401                      buffer_manager);
402 } else {
403     std::string schema_str;
404     std::getline(file, schema_str);
405     auto [columns, record_size] =
406         read_columns(std::stringstream(std::move(schema_str)));
407     int records_per_sector =
408         8 * (globalDiskInfo.bytes - sizeof(Address) - sizeof(int)) /
409             (8 * record_size + 1);
410
411     auto records_start = write_table_header(csv_name, columns, buffer_manager);
412     write_table_data(file, records_start, columns.data(), columns.size(),
413                      records_per_sector, buffer_manager);
414 }
415
416 buffer_manager.unpin(header_sector);
417 }
```