

Aplicación de alto nivel para
administrar registros de longitud
fija con una llave en memoria
secundaria con C++ y Python

Introducción

Objetivo: implementar una aplicación que facilite la administración eficiente de registros (de longitud fija con una llave primaria) en memoria secundaria

No es un objetivo:
recrear SQL

Funcionalidades:

1. **Crear** una tabla bajo un método de indexación
2. **Añadir** un elemento a una tabla
3. **Eliminar** un elemento de una tabla
4. **Buscar** un elemento de una tabla por llave
5. **Importar** datos de un CSV a una tabla

Dominio de datos: **(casi) genérico**

Cualquiera conformado por: ints, floats y/o texto fijo

No soporta: fechas (al menos que estén como texto fijo), arreglos numéricos, datos de longitud variable.

Métodos de administración que soportará en esta versión:

- 1. Ordered sequential file**
- 2. Extendible hash**

Metodología:


1. Implementar las **clases de administración de data** de memoria secundaria
2. Implementar una **interfaz gráfica y pseudo-lenguaje** para abstraer su uso (concurrente)
3. Medir **performance** con un grupo de datos

Clases de administración de data

Ordered sequential file

Argumentos de template

```
template<  
    typename Tipo del registro,  
    typename Tipo de la llave del registro  
    std::uint32_t Tamaño de página en bytes  
>
```



tamaño del registro < PG < 1600
(hay errores para valores fuera de
ese rango)

Esqueleto y estructuras internas principales

```
struct Pointer{  
    bool in_aux;  
    std::uint32_t index;  
};  
  
struct Record{  
    T data;  
    bool dirty;  
    Pointer next;  
};  
  
const Pointer End{false, 999999999};
```

```
public:  
  
bool search(const Key_t& key, T& empty){ ...  
}  
  
void erase(const Key_t& key){ ...  
}  
  
void add(const T& data){ ...  
}
```

Algoritmo general para AÑADIR registros

*El archivo aux es del tamaño de una página por diseño

1. Búsqueda binaria en el archivo ordenado $O(\log(n))$
2. Si está sucio o es la misma llave, se reemplaza y termina $O(1)$
3. Se itera por la lista ordenada del archivo auxiliar hasta que sea mayor, igual o el final de la lista, y se añade. $O(1)^*$
4. Si la cantidad de elementos en el archivo auxiliar sobrepasa el máximo, se recrea el archivo ordenado y sin sucios $O(n)$

Accesos a memoria secundaria en el peor caso:

$$T = \log(\text{main size}/PG) + 1 + 2 + 2 * (\text{main size}/PG) + 1$$

Algoritmo general para ELIMINAR registros

*El archivo aux es del tamaño de una página por diseño

1. Búsqueda binaria en el archivo ordenado $O(\log(n))$
2. Si es, se marca sucio y termina $O(1)$
3. Se itera por la lista ordenada del archivo auxiliar hasta encontrarlo o hasta el final de la lista, y se marca sucio. $O(1)^*$

Accesos a memoria secundaria en el peor caso:

$$T = \log(\text{main size}/PG) + 1 + 2$$

Algoritmo general para BUSCAR registros

*El archivo aux es del tamaño de una página por diseño

1. Búsqueda binaria en el archivo ordenado
2. Si es, se devuelve y termina
3. Se itera por la lista ordenada del archivo auxiliar hasta encontrar o hasta el final de la lista, y se devuelve.

$O(\log(n))$

$O(1)^*$


Accesos a memoria secundaria en el peor caso:

$$T = \log(\text{main size}/PG) + 1$$

Extendible hash

Argumentos de template

```
template<  
    typename Tipo del registro,  
    typename Tipo de la llave del registro  
    std::uint32_t Tamaño de página en bytes  
>
```



tamaño del registro < PG < 1600
(hay errores para valores fuera de
ese rango)

Esqueleto y estructuras internas principales

```
std::uint32_t actual_depth = 2;

struct Bucket {
    std::uint32_t current_size = 0;
    T arr[Registers_per_Bucket];
};
```

```
public:

bool search(const Key_t& key, T& empty){ ...
}

void erase(const Key_t& key){ ...
}

void add(const T& data){ ...
}
```

Algoritmo general para AÑADIR registros

1. Calcular $\text{llave} \% \text{profundidad global}$ (módulo)
2. Acceder al índice correspondiente en index file $O(1)$
3. Acceder al bucket correspondiente en el data file $O(1)$
4. Si hay aún espacio en el bucket, se inserta y termina $O(1)$
5. Se cuentan los índices apuntando al mismo bucket $O(n)$
6. Si hay más de 1, se hace split y se reinsertan los datos (1) $O(1)$
7. Si hay solo 1, se duplican (clonan) los índices, se duplica la profundidad global y se repite desde (5) $O(n)$

Accesos a memoria secundaria en el peor caso:

$$T = 1 + 1 + 1 + \text{index file size}/PG + 2 * \text{index file size}/PG + \text{index file size}/PG + 3 * \text{bucket size}/PG$$

Algoritmo general para ELIMINAR registros

1. Calcular $\text{llave} \% \text{profundidad global}$ (módulo)
2. Acceder al índice correspondiente en index file
3. Acceder al bucket correspondiente en el data file
4. Se busca en el bucket, se elimina y se guarda

$O(1)$

$O(1)$

$O(1)$

Accesos a memoria secundaria en el peor caso:

$$T = 1 + 1 + 1$$

Algoritmo general para BUSCAR registros

1. Calcular $\text{llave} \% \text{profundidad global}$ (módulo)
2. Acceder al índice correspondiente en index file
3. Acceder al bucket correspondiente en el data file
4. Se busca en el bucket, se devuelve

$O(1)$

$O(1)$

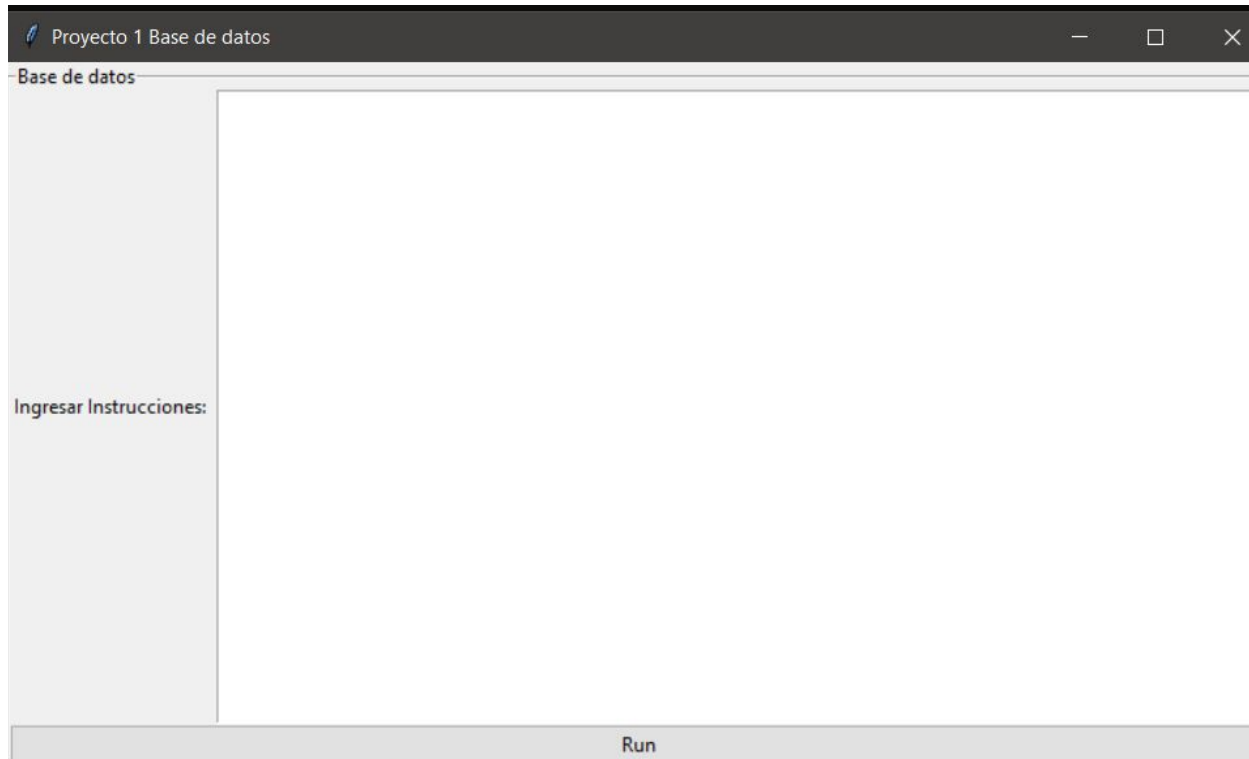
Accesos a memoria secundaria en el peor caso:

$$T = 1 + 1$$

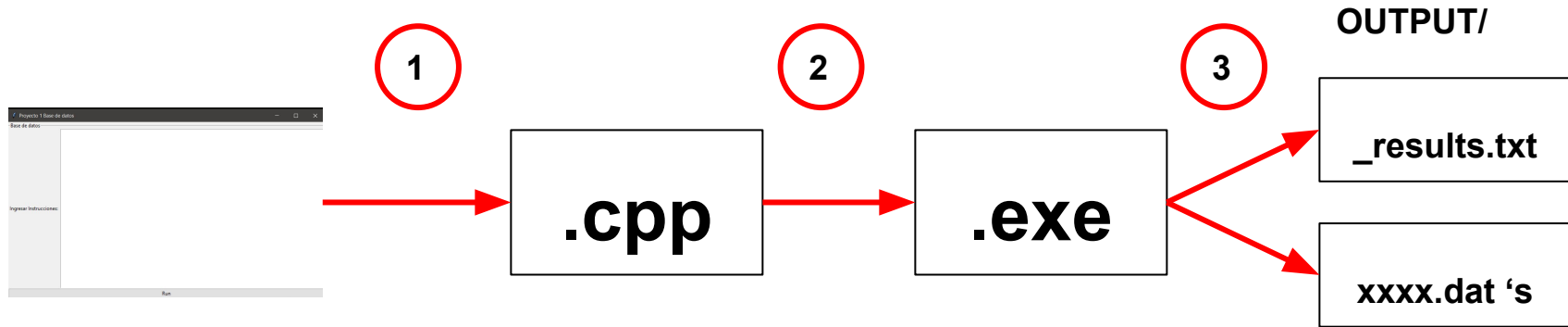
Interfaz gráfica y pseudolenguaje

Funcionamiento

Diseño de la aplicación



Lo que sucede al presionar Run en nuestra aplicación



1. Se parsea el pseudocódigo y se traduce a C++
2. Se compila y se ejecuta el código de C++ generado
3. El nuevo programa de C++ modifica **_results.txt** y binarios

¿Por qué hace todo eso el programa?

Pregunta: Digamos que se quiere crear una tabla con [int, int, bool, etc]. ¿Cómo se crearía un Struct/Clase que tenga esos tipos de elementos en C++ en tiempo de ejecución?

Respuesta: No se puede. Todos los tipos de datos se tienen que poder inferir en tiempo de compilación en C++. Por ello, una solución es escribir un archivo de C++ desde otro programa, compilarlo y ejecutarlo.

Concurrencia en el .cpp generado

1. Se crea un hilo por cada tabla a utilizar: las operaciones de **diferentes tablas** se ejecutan de manera **concurrente**
2. La ejecución de cada hilo de tabla se divide en dos momentos
 - a. Se ejecutan todos los **reads** de manera **concurrente** (más hilos)
 - b. Se ejecutan todos los **writes** de manera **secuencial**

Documentación

Documentación del pseudolenguaje

- **Tipos de elementos permitidos en una tabla**
 - **INumber:** 32 bits integer number
 - **UINumber:** 32 bits unsigned integer number
 - **FNumber:** float number
 - **Text<size>:** char array con un tamaño fijo (tiene sobrecarga de <, > y ==)

Documentación del pseudolenguaje

- **Reglas de redacción**
 - Cada instrucción es una línea
 - Cada parte de la instrucción está separada por un espacio
 - Siempre se tiene que definir las tablas a usar antes.
 - El primer atributo de un tipo de dato en una tabla es el key
 - Los dato tipo Text<> utilizan “ ” y no pueden contener espacios
 - (No se pone [], solo está en la documentación por orden)

Documentación del pseudolenguaje

- **Definir una tabla**
 - `table [tabla_name] [SF/EH] [PG_sz] [tipo1] [nombre1] [tipo2] [...]`
- **Añadir a una tabla**
 - `add [tabla_name] [atributo1] [atributo2] ...`
- **Eliminar de una tabla**
 - `erase [tabla_name] [key (es decir, atributo 1)]`
- **Buscar en una tabla**
 - `search [tabla_name] [key (es decir, atributo 1)]`
- **Importar CSV a una tabla**
 - `import [tabla_name] ["path/to/file.csv"]`

Performance

Cuanto más espacio ocupan comparado con el tamaño de la data sola

*Tamaño de página definida por el usuario: 1000 bytes

*Tamaño de la data de un registro: 67 bytes

veces más



Va a volver a subir cuando haya más splits (oscila)

Cuánto demora importar un conjunto grande de datos

*Tamaño de página definida por el usuario: 1000 bytes



Cuánto demora insertar un elemento individual al haber X registros

*Tamaño de página definida por el usuario: 1000 bytes

*IMPORTANTE: cuenta con el overhead de los threads y no los usa en este caso



Cuánto demora en buscar un registro individual al haber X registros

*Tamaño de página definida por el usuario: 1000 bytes

*IMPORTANTE: cuenta con el overhead de los threads y no los usa en este caso

milisegundos (empate)



Cuánto demora en buscar 10 registros a la vez al haber X registros

*Tamaño de página definida por el usuario: 1000 bytes

*En este caso, sí le saca provecho al multithreading



Conclusiones

Conclusiones

1. **Toda abstracción tiene un costo**
 - a. **Costo de abstraer las clases (dejarlas puramente genéricas):** no se pueden hacer optimizaciones alrededor de un específico tipo de data
 - b. **Costo de abstraer C++:** tiempo de escritura y compilación de un cpp durante la ejecución del programa principal
 - c. **Costo de abstraer multithread:** empeora los casos de accesos individuales

Conclusiones

2. **Prioriza el EH sobre SF, a no ser que la memoria sea escasa**
 - a. Tienen un desempeño parecido en los casos comunes
 - b. El SF cada cierta cantidad de inserts, tiene una caída considerable de performance (picos de tiempo)
 - c. **El SF es más constante en memoria, el EH es más constante en tiempo de escritura y lectura.**