UNIVERSIDAD NACIONAL DE SAN AGUSTÍN

FACULTAD DE INGENIERÍA, PRODUCCIÓN Y SERVICIOS

ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

Examen Parcial II Buffer Manager

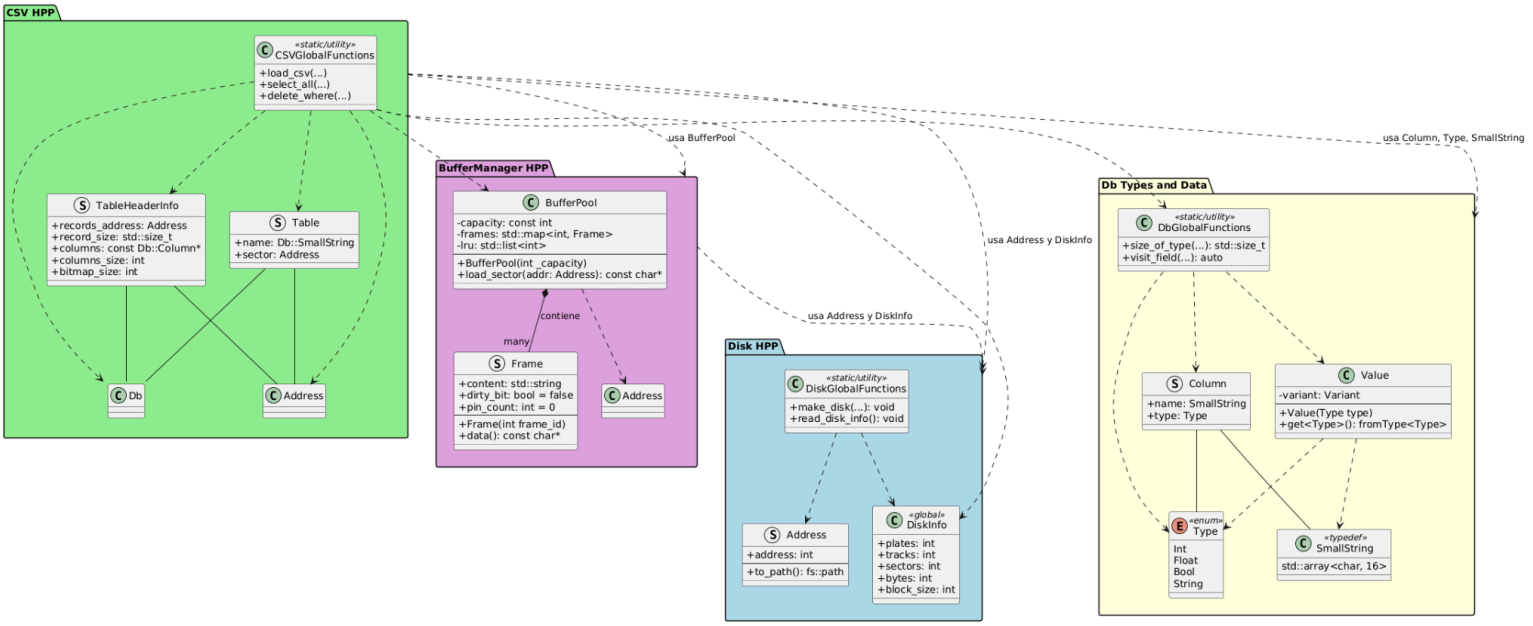
Integrantes

Alberto Gabriel Torres Ara CUI: 20233585

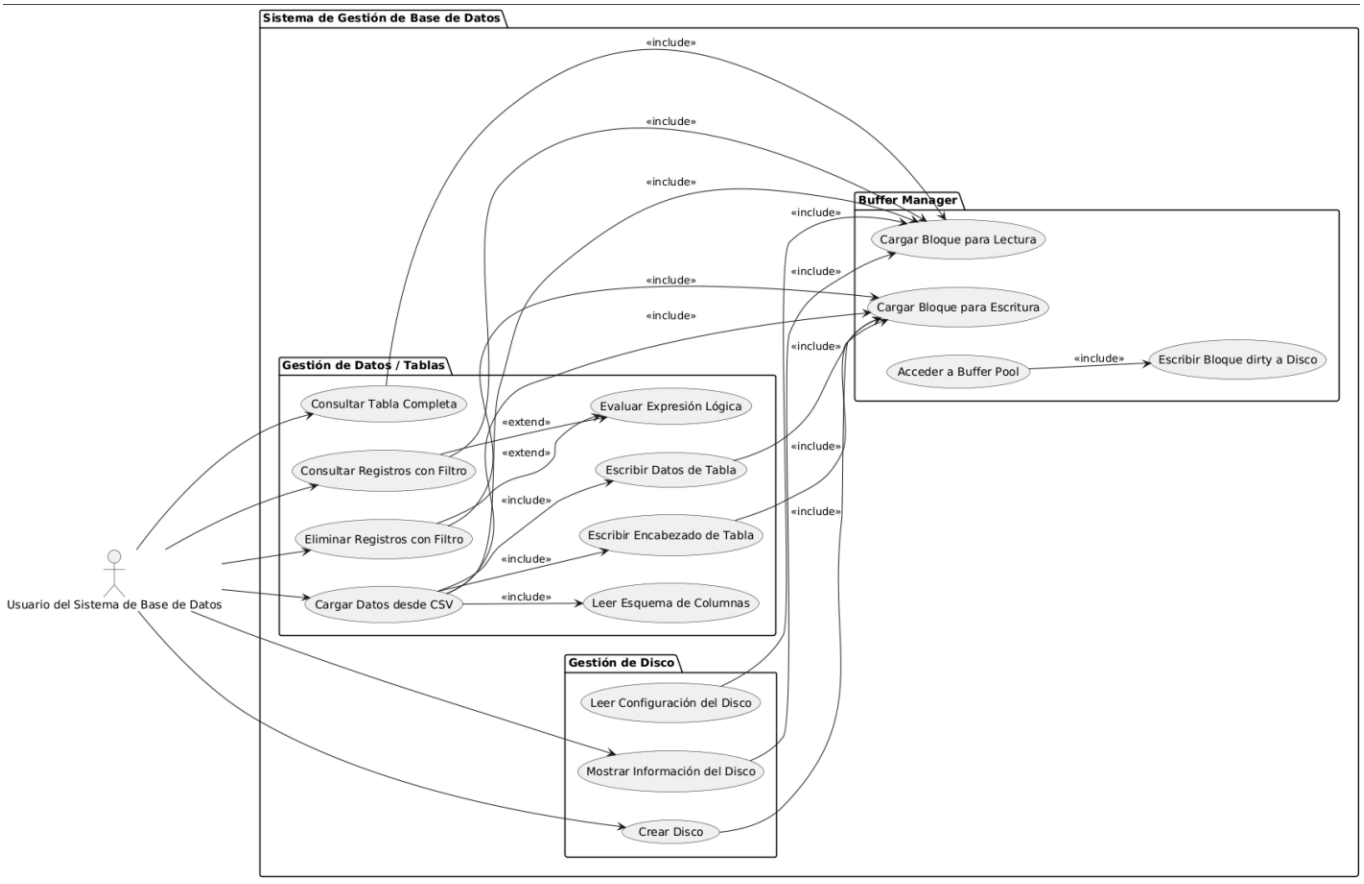
Arthur Patrick Meza Pareja CUI: 20231535

AREQUIPA-PERÚ 2025

1.- UML



2.-Diagrama de casos



3.- Describir cuál es la funcionalidad y objetivo de cada clase y sus funciones

3.1.- Buffer Manager

Clase Frame

La clase Frame representa una página de datos cargada en la memoria principal. Cada Frame es una copia de un bloque de datos del disco, y es la unidad básica que el BufferManager gestiona.

Funcionalidad y Objetivo:

- **Contener datos de una página: Su objetivo principal es almacenar el contenido de un bloque de disco para un acceso rápido.**
- **Seguimiento de modificaciones (dirty_bit): Permite saber si el contenido en memoria ha sido modificado y necesita ser escrito de vuelta al disco. Esto es vital para la persistencia de datos.**
- **Control de uso (pin_count): Evita que una página sea desalojada de la memoria mientras está siendo utilizada activamente por alguna parte de la aplicación.**

Funciones Clave:

- **Frame(int frame_id) (Constructor):**
    - **Funcionalidad: Inicializa un Frame cargando el contenido de un bloque específico del disco. Lee sector por sector el bloque correspondiente al frame_id y concatena su contenido en la cadena content.**
    - **Objetivo: Poblar el Frame con los datos actuales del disco cuando se carga por primera vez o se reemplaza un Frame existente.**
- **const char* data() const:**
    - **Funcionalidad: Retorna un puntero constante al inicio de los datos del Frame.**
    - **Objetivo: Proporcionar acceso de solo lectura al contenido del Frame.**

- **char* writeable_data():**
    - **Funcionalidad: Retorna un puntero no constante al inicio de los datos del Frame y simultáneamente establece dirty_bit en true.**
    - **Objetivo: Permitir el acceso de escritura al contenido del Frame y marcarlo como modificado para su posterior escritura a disco.**

Clase BufferManager

La clase BufferManager es el corazón del sistema de caché. Administra una colección de Frames en memoria, aplicando una política de reemplazo (LRU) para decidir qué páginas mantener y cuáles desalojar.

Funcionalidad y Objetivo:

- **Gestionar el Buffer Pool: Mantiene un conjunto de Frames en memoria (el "buffer pool") para almacenar temporalmente los bloques de disco más accedidos.**
- **Optimizar Acceso a Disco: Reduce la latencia al servir solicitudes de datos directamente desde la memoria si ya están cacheados (cache hit), evitando costosas operaciones de I/O a disco.**
- **Implementar Política de Reemplazo LRU: Decide qué Frame desalojar cuando el buffer pool está lleno y se necesita espacio para una nueva página, priorizando las páginas menos recientemente usadas.**
- **Asegurar Persistencia de Datos: Se encarga de escribir los Frames modificados (dirty) de vuelta al disco cuando son desalojados o cuando el BufferManager se cierra.**
- **Mantener Estadísticas: Calcula el "hit rate" para evaluar la eficiencia de la caché.**

Funciones Clave:

- **BufferManager(int _capacity) (Constructor):**
    - **Funcionalidad: Inicializa el BufferManager con una capacidad fija para el número de Frames que puede albergar.**

    - **Objetivo: Configurar el tamaño máximo del buffer pool.**
- **~BufferManager() (Destructor):**
    - **Funcionalidad: Al finalizar la vida del BufferManager, itera sobre todos los Frames en el pool. Si un Frame tiene su dirty_bit en true, escribe su contenido de vuelta a los sectores correspondientes en el disco.**
    - **Objetivo: Garantizar que todas las modificaciones hechas en memoria se persistan en el disco antes de que el buffer se libere, previniendo la pérdida de datos.**
- **void print() const:**
    - **Funcionalidad: Imprime una tabla con el estado actual de cada Frame en el buffer pool, incluyendo su ID, estado (lectura/escritura), dirty_bit, pin_count y su posición en la lista LRU. También muestra las estadísticas de acceso y el hit rate.**
    - **Objetivo: Ayudar a la depuración y monitoreo del comportamiento del BufferManager, permitiendo visualizar cómo se gestionan los frames y cuál es la eficiencia de la caché.**
- **const char* load_sector(Address sector_address):**
    - **Funcionalidad: Proporciona acceso de solo lectura a un sector específico. Calcula el block_id (ID de página) a partir de la sector_address. Si el bloque ya está en el pool, lo marca como recientemente usado (actualiza en LRU) y retorna un puntero a los datos. Si no está y hay espacio, carga el bloque. Si no hay espacio, aplica la política LRU para desalojar un Frame no pineado, escribe si está dirty, lo elimina y carga el nuevo bloque.**
    - **Objetivo: Servir solicitudes de lectura de datos, priorizando la búsqueda en caché y gestionando el reemplazo de páginas según la política LRU.**
- **char* load_writeable_sector(Address sector_address):**
    - **Funcionalidad: Similar a load_sector, pero proporciona acceso de escritura al sector. Después de cargar o encontrar el bloque, asegura que el dirty_bit del Frame correspondiente sea true.**

    - **Objetivo: Servir solicitudes de escritura de datos, asegurando que cualquier modificación se marque para su eventual escritura a disco.**
- **void pin(Address sector_address):**
    - **Funcionalidad: Incrementa el pin_count del Frame al que pertenece el sector dado.**
    - **Objetivo: "Fijar" una página en memoria, impidiendo que sea desalojada por la política LRU mientras está siendo utilizada por una transacción o una operación, garantizando su disponibilidad.**
- **void unpin(Address sector_address):**
    - **Funcionalidad: Decrementa el pin_count del Frame al que pertenece el sector dado, sin que el contador baje de cero.**
    - **Objetivo: "Desfijar" una página, indicando que ya no está en uso activo y puede ser considerada para desalojo por la política LRU si es necesario.**

3.2.- Disco

Define las estructuras y funciones para simular un sistema de disco persistente utilizando el sistema de archivos local, donde cada sector del disco es representado por un archivo individual.

Estructuras Clave

- **DiskInfo:**
    - **Funcionalidad y Objetivo: Una estructura global (globalDiskInfo) que almacena los parámetros físicos del disco simulado: número de platos, pistas, sectores por pista, tamaño en bytes por sector y el tamaño de un bloque (número de sectores por bloque). Su objetivo es proporcionar una configuración global del disco.**
- **Address:**
    - **Funcionalidad y Objetivo: Representa una dirección lógica de un sector en el disco. Permite la conversión entre una dirección lineal (int address) y una dirección física jerárquica (plato,**

superficie, pista, sector), así como la generación del std::filesystem::path correspondiente al archivo del sector. Su objetivo es abstraer la complejidad del direccionamiento físico y la ubicación de archivos.

Funciones Esenciales

- **make_disk(const DiskInfo& diskInfo):**
  - **Funcionalidad y Objetivo: Crea la estructura de directorios y archivos que simulan el disco físico en el sistema de archivos. Para cada sector, crea un archivo de tamaño diskInfo.bytes. Además, persiste la DiskInfo de configuración en el primer sector (p0/f0/t0/s0) para que pueda ser recuperada más tarde. Su objetivo es inicializar el entorno del disco simulado.**
- **read_disk_info():**
  - **Funcionalidad y Objetivo: Lee la DiskInfo previamente guardada en el primer sector del disco. Esto permite que el sistema conozca las características del disco simulado sin necesidad de recrearlo. Su objetivo es cargar la configuración del disco al inicio de la aplicación.**
- **pun_cast (Funciones de reinterpret_cast):**
  - **Funcionalidad y Objetivo: Funciones auxiliares que realizan un reinterpret_cast para tratar una variable de cualquier tipo T como un std::array<char, sizeof(T)>. Esto es crucial para leer y escribir estructuras de datos directamente desde/hacia archivos a nivel de bytes, sin serialización explícita. Su objetivo es permitir la lectura y escritura binaria de estructuras complejas.**
  -

3.3.- Table

Cargar datos desde archivos CSV a un formato persistente en el disco simulado, y para realizar operaciones básicas de consulta y manipulación de datos (SELECT, DELETE) sobre estas tablas. Se integra con el BufferManager para interactuar con el disco y con el Interpreter para evaluar condiciones en las consultas.

Estructuras Clave

- **Table:**
    - **Funcionalidad y Objetivo: Una estructura "Plain Old Data" (POD) que almacena el nombre de una tabla (Db::SmallString name) y la dirección del sector de disco donde comienza su cabecera (Address sector). Su objetivo es mantener un registro de las tablas existentes y dónde encontrar su metadatos en el disco.**
- **TableHeaderInfo:**
    - **Funcionalidad y Objetivo: Una estructura auxiliar que agrupa toda la información necesaria para trabajar con los registros de una tabla: la dirección del primer sector de datos, el tamaño de cada registro, un puntero a la información de las columnas, el número de columnas, y el tamaño del bitmap de registros por sector. Su objetivo es encapsular los metadatos de una tabla para un acceso conveniente.**

Funciones de Bajo Nivel (Auxiliares)

- **read_columns(std::stringstream schema):**
    - **Funcionalidad y Objetivo: Parsea una cadena de texto que representa el esquema de columnas (ej. "Nombre#String,Edad#Int") y retorna un vector de Db::Columns y el tamaño total en bytes de un registro. Su objetivo es interpretar la definición de la estructura de una tabla.**

- **request_available_sector(int records_per_sector, BufferManager& buffer_manager):**
    - **Funcionalidad y Objetivo: Busca un sector de disco que ya contenga registros y que aún tenga espacio disponible para añadir más. Itera sobre todos los sectores, carga su cabecera y verifica el contador de registros. Su objetivo es encontrar un lugar donde appendear nuevos registros eficientemente.**
- **request_empty_sector(BufferManager& buffer_manager):**
    - **Funcionalidad y Objetivo: Busca un sector de disco que esté completamente vacío (indicado por un NullAddress en su primera posición). Su objetivo es encontrar un nuevo sector para almacenar datos, ya sea una cabecera de tabla o el inicio de una cadena de registros.**
- **disk_info(BufferManager& buffer_manager):**
    - **Funcionalidad y Objetivo: Imprime estadísticas detalladas sobre el espacio en el disco simulado (total, disponible, ocupado) listando los sectores vacíos. Su objetivo es proporcionar una visión general del uso del disco.**
- **search_table(const std::string& table_name, BufferManager& buffer_manager):**
    - **Funcionalidad y Objetivo: Busca una tabla por su nombre en el "sector maestro" del disco (el sector 0, justo después de la DiskInfo). Retorna la Address del sector de la cabecera de la tabla si la encuentra, o NullAddress si no. Su objetivo es localizar los metadatos de una tabla específica.**
- **write_table_header(...):**
    - **Funcionalidad y Objetivo: Escribe la información de la cabecera de una nueva tabla. Esto incluye añadir la tabla al listado en el sector 0, y luego escribir el número de columnas y la definición de cada columna en un nuevo sector de cabecera dedicado a la tabla. Su objetivo es registrar una nueva tabla en el sistema.**
- **write_sector_header(...):**
    - **Funcionalidad y Objetivo: Inicializa la cabecera de un sector de datos que contendrá registros. Establece el puntero al "siguiente**

sector" (inicialmente NullAddress), el contador de registros a cero y el bitmap de registros como vacío. Su objetivo es preparar un sector para almacenar registros.

- **write_record(...):**
    - **Funcionalidad y Objetivo: Parsea una línea de un CSV (un registro) y escribe sus campos en el formato binario correspondiente en la memoria del BufferManager, moviendo el puntero record_data. Maneja la conversión de tipos (Int, Float, Bool, String). Su objetivo es serializar un registro en bytes para su almacenamiento en disco.**
- **write_table_data(...):**
    - **Funcionalidad y Objetivo: Lee línea por línea un archivo CSV, escribe cada registro en los sectores de datos disponibles (solicitando nuevos sectores cuando sea necesario) y actualiza el bitmap de ocupación dentro de cada sector. Su objetivo es cargar masivamente los datos de un CSV a una tabla en disco.**
- **read_table_header(...):**
    - **Funcionalidad y Objetivo: Lee la información de la cabecera de una tabla (número de columnas, definiciones de columnas, tamaño de registro, etc.) desde el disco, dado el nombre de la tabla. Calcula el tamaño del bitmap y los registros por sector. Su objetivo es recuperar los metadatos de una tabla existente.**
- **visit_records (Template):**
    - **Funcionalidad y Objetivo: Un patrón de diseño que itera sobre todos los registros de una tabla (siguiendo la cadena de sectores) y aplica un Visitor (una función lambda o un objeto función) a cada registro válido. Utiliza load_sector para acceso de solo lectura. Su objetivo es abstraer la iteración sobre los registros almacenados.**
- **visit_writeable_records (Template):**
    - **Funcionalidad y Objetivo: Similar a visit_records, pero utiliza load_writeable_sector para permitir que el Visitor modifique los datos del registro directamente en el buffer, marcando los frames**

como dirty. Su objetivo es permitir la modificación o eliminación de registros.

Funciones de Alto Nivel (Operaciones con Tablas)

- **load_csv(const std::string& csv_name, BufferManager& buffer_manager):**
    - **Funcionalidad y Objetivo: La función principal para importar datos. Si la tabla ya existe, abre el CSV, omite la cabecera y añade nuevos registros. Si la tabla no existe, lee el esquema del CSV, crea la cabecera de la tabla en disco y luego escribe todos los registros del CSV. Su objetivo es poblar o añadir datos a una tabla a partir de un archivo CSV.**
- **select_all(const std::string& table_name, BufferManager& buffer_manager):**
    - **Funcionalidad y Objetivo: Recupera todos los registros válidos de una tabla y los imprime en la consola. Utiliza read_table_header y visit_records para iterar y mostrar los datos. Su objetivo es implementar la funcionalidad SELECT * FROM table.**
- **select_all_where(const std::string& table_name, const std::string& expression, BufferManager& buffer_manager):**
    - **Funcionalidad y Objetivo: Recupera y imprime solo los registros que satisfacen una condición dada por una expression. Utiliza el Interpreter (parseExpression) para construir un árbol de evaluación y luego visit_records para iterar, evaluando la expresión en cada registro. Su objetivo es implementar la funcionalidad SELECT * FROM table WHERE condition.**
- **delete_where(const std::string& table_name, const std::string& expression, BufferManager& buffer_manager):**
    - **Funcionalidad y Objetivo: "Elimina" lógicamente registros de una tabla que cumplen con una expression. No borra físicamente los datos, sino que actualiza el bitmap de cada sector para marcar el registro como no válido. Utiliza visit_writeable_records y el**

Interpreter para identificar y marcar los registros a eliminar. Su objetivo es implementar la funcionalidad DELETE FROM table WHERE condition.

3.4.- Type

representación y el manejo de los tipos de datos dentro de la base de datos. Define los tipos de datos que el sistema puede almacenar, la estructura de una columna y un sistema de valores genéricos (Value) que permite trabajar con diferentes tipos de datos de manera uniforme, especialmente útil para el intérprete de expresiones.

Namespace Db

Contiene todas las definiciones relacionadas con la gestión de información.

Utilidades (Namespace Anónimo)

- **runtimeVariant (Función Template):**
  - **Funcionalidad y Objetivo: Esta función template recursiva permite construir un std::variant (un tipo que puede contener uno de varios tipos diferentes) en tiempo de ejecución, basándose en un índice numérico.**
  - **Objetivo: Proporciona una manera de crear una instancia de Value::Variant a partir de un Db::Type conocido solo en tiempo de ejecución (que se convierte a un índice). Esto es crucial para la flexibilidad del sistema de tipos.**

Tipos Fundamentales

- **SmallString:**
  - **Funcionalidad y Objetivo: Un alias para std::array<char, 16>. Se utiliza para almacenar cadenas de texto cortas de tamaño fijo, como los nombres de las columnas.**
  - **Objetivo: Optimizar el almacenamiento de cadenas cortas en disco al usar un tamaño predefinido, lo que simplifica la gestión de memoria y el pun_cast.**
- **enum class Type:**
  - **Funcionalidad y Objetivo: Define los tipos de datos primitivos que la base de datos soporta: Int, Float, Bool, String. Utiliza un std::size_t subyacente para facilitar la conversión a índices para std::variant.**
  - **Objetivo: Estandarizar y tipificar los datos que pueden ser almacenados y manipulados en la base de datos.**

Estructura de Datos del Esquema

- **struct Column:**
  - **Funcionalidad y Objetivo: Representa la definición de una columna de una tabla tal como se almacena en el disco. Contiene el nombre de la columna (SmallString name) y su tipo de dato (Type type).**
  - **Objetivo: Describir la estructura de una tabla y sus campos.**
- **operator>>(std::istream& is, Db::Type& type) (Sobrecarga):**
  - **Funcionalidad y Objetivo: Permite leer un Db::Type desde un flujo de entrada (como std::cin o std::stringstream). Convierte nombres de tipo como "INT", "FLOAT", etc., a su enum class Type correspondiente.**
  - **Objetivo: Facilitar la lectura de esquemas de tablas desde archivos o entradas de usuario.**

Sistema de Valores Genéricos

- **struct Value:**
  - **Funcionalidad y Objetivo: Una clase crucial que encapsula un std::variant para representar un valor de cualquier tipo de dato soportado por la base de datos. Es la abstracción principal para manejar los campos de los registros y los resultados de las expresiones.**
  - **using Variant = std::variant<std::int64_t, double, bool, std::array<char, 64>>: Define los tipos concretos que el Value puede contener. Nota que String se representa como std::array<char, 64> (tamaño fijo para cadenas).**
  - **template <Type type> using fromType = ...: Un alias de tipo para obtener el tipo subyacente de Variant dado un Db::Type.**
  - **Constructores: Permiten crear un Value a partir de un Db::Type (inicializando el variant con un valor por defecto del tipo correspondiente) o directamente desde un valor concreto (ej. Value(10), Value(true)).**
  - **friend decltype(auto) visit(Visitor&& v, Values&&... values): Un "friend function" que expone la funcionalidad std::visit del std::variant interno. Permite aplicar un functor (visitor) al valor contenido en uno o más objetos Value.**
  - **template <Type type> fromType<type> get(): Un método miembro para acceder al valor subyacente del Value de forma segura por tipo (Db::Type).**
  - **Objetivo: Proporcionar una interfaz uniforme y segura para manejar valores de diferentes tipos de datos, especialmente en el contexto de la evaluación de expresiones y la lectura/escritura de registros.**

Funciones de Acceso y Tamaño de Datos

- **template <Db::Type Type = Db::Type::Int> constexpr std::size_t size_of_type(Db::Type type):**
    - **Funcionalidad y Objetivo: Una función template recursiva que calcula el tamaño en bytes que ocupa un Db::Type dado en memoria/disco.**
    - **Objetivo: Determinar el offset y el espacio requerido para cada campo en un registro binario.**
- **template <class Visitor, Db::Type Type = Db::Type::Int> auto visit_type(const void* pointer, Db::Type type, Visitor&& v):**
    - **Funcionalidad y Objetivo: Una función template recursiva que "visita" un dato binario en una dirección de memoria (void* pointer) con un tipo Db::Type conocido en tiempo de ejecución. Convierte el void* al puntero de tipo correcto y aplica el Visitor.**
    - **Objetivo: Permitir el acceso tipado a datos brutos almacenados en memoria (obtenidos del disco) sin necesidad de múltiples if-else o switch-case para cada tipo.**
- **template <class Visitor> auto visit_field(const char* record, std::size_t index, const Column* columns, Visitor&& v):**
    - **Funcionalidad y Objetivo: La función clave para acceder a un campo específico dentro de un registro. Calcula el offset de un campo dado su index y el columns (esquema de la tabla), y luego llama a visit_type para aplicar el Visitor al valor real del campo.**
    - **Objetivo: Proporcionar un mecanismo genérico y robusto para extraer y operar sobre campos individuales de registros almacenados de forma contigua en memoria.**